Міністерство освіти і науки України
ДВНЗ “Прикарпатський національний університет імені Василя Стефаника”
Кафедра комп’ютерної інженерії та електроніки

Курс лекцій з дисципліни
“СИСТЕМНЕ ПРОГРАМУВАННЯ”

Затверджено
на засіданні кафедри радіофізики і електроніки
протокол № 4 від 8 листопада 2017 р

Розроблено:
доц. Голота В.І.

Івано-Франківськ 2017

Зміст

1. СНОВНІ ПОНЯТТЯ 3
2. РЕЖИМИ РОБОТИ І МОДЕЛІ ПАМ’ЯТІ ПРОЦЕСОРІВ INTEL x86-32/64 26
3. ОСНОВИ АСЕМБЛЕРА...39
4. КЛАСИФІКАЦІЯ КОМАНД, КОМАНДИ ПЕРЕСИЛАННЯ ДАНИХ, АРИФМЕТИЧНІ КОМАНДИ, ЛОГІЧНІ КОМАНДИ І ОПЕРАЦІЇ, ЛАНЦЮГОВІ КОМАНДИ 52
5. КОМАНДИ ПЕРЕДАЧІ КЕРУВАННЯ 72
6. МАКРОПРОЦЕСОР І МАКРОДИРЕКТИВИ...80
7. ПРОЦЕСИ, СИСТЕМНІ ВИКЛИКИ І ПІДПРОГРАМИ 94
8. СПІВПРОЦЕСОР 111
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 136

1. СНОВНІ ПОНЯТТЯ

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

Вступ. Системне програмування зв'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 режим 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 архітектурні особливості процесорів та формати машинного подання чисел.

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. Границі 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. 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 апаратної частини. 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 функціонування програми;

• використовуються низькорівневі мови програмування які можуть працювати у ресурсо-обмеженому 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 несанкціонованого доступу та шкідливих програм.

Кожен із вказаних задач є самостійним напрямом в системному програмуванні, що змушує фахівців програмістів спеціалізуватися у одній з цих галузей.

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. Термін **little-endian** використовується для позначення послідовності байтів в якій молодші розрядові байти розміщуються в молодших адресах пам'яті. Термін **big-endian** позначає послідовності байтів у яких молодші розрядові байти розміщуються в старших адресах пам'яті. В Intel процесорах використовується little-endian послідовність зберігання байтів (II), а в процесорах фірми Motorola – big-endian послідовність (MM). Так для послідовностей чисел

```
m1 db 0,0,0x43,0x21
```

порядок розміщення байтів у пам’яті може буде наступним:

	little-endian				**big-endian**			
Адреси пам'яті	0	1	2	3	0	1	2	3
Розрядові байти	21	43	00	00	43	21	00	00

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

	little-endian				**big-endian**			
Адреси пам'яті	0	1	2	3	0	1	2	3
Розрядові байти	00	21	00	43	43	00	21	00

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 розглядувати як стрічку з двійкових розрядів, ціле число, число з плаваючою крапкою, 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 “словом, word” називають два байти (16 біт), а “подвійним словом, dword” – чотири байти (32 біт).

Програма, 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.

Процесор працює, раз за разом виконуючи **цикл оброблення команди**. На початку цього циклу, з комірок пам’яті, на які вказує лічильник команд, зчитується код чергової команди. Зразу після цього лічильник команд міняє своє значення так, щоб вказувати на наступну команду у пам’яті. Схеми процесора дешифрують код і виконують дії зумовлені цим кодом. Коли дії, зумовлені командою, будуть виконані, процесор знов повертається до початку циклу обробки команд. Наступний прохід цього циклу буде виконувати наступну команду і так поки не будуть виконані всі команди.

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, які використовуються 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 програм, 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 півня. 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 команд 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 число 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 центрального процесора. 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 приводить до появи "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
```

Для подання знакових цілих чисел використовується доповнюючий (доповняльний) код. Доповнюючий код можна уявити як відображення деякого діапазону, який включає позитивні і негативні цілі числа на інший діапазон, який містить тільки позитивні числа, рис. 2.

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.

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

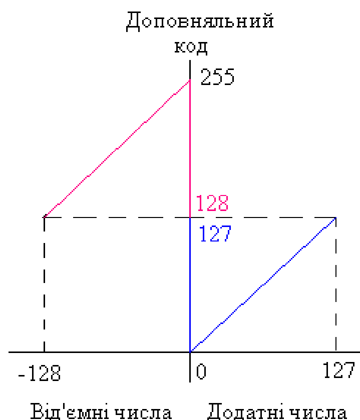


Рисунок 2 – Схема відображення чисел в доповнюючий код

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

0000.0001 1111.1110	0000.0010 1111.1101	0000.0011 1111.1100

+ 1 ---------------- 1111.1111	+ 1 ---------------- 1111.1110	+ 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, в не 255-1=254.

Таким чином прийнята наступна домовленість: якщо набір двійкових чисел розглядується як подання знакового числа, то **від'ємними вважаються числа, старший біт яких дорівнює одиниці**. Діапазон значень знакових цілих чисел

-128	-127	…	-2	-1	…	0	1	127
1000.0000	1000.0001	…	1111.1110	1111.1111	…	0000.0000	0000.00001	0111.1111

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

$5_{10} \rightarrow 0000.0101_2 \xrightarrow{inv} 1111.1010 \xrightarrow{+1} 1111.1011 \rightarrow -5$

$-5_{10} \rightarrow 1111.1011_2 \xrightarrow{inv} 0000.0100 \xrightarrow{+1} 0000.0101 \rightarrow 5$

При зміні знаку нуля нуль залишається нулем:

$0_{10} \rightarrow 0000.0000_2 \xrightarrow{inv} 1111.1111 \xrightarrow{+1} 0000.0000 \rightarrow 0$

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

$-128_{10} \rightarrow 1000.0000_2 \xrightarrow{inv} 0111.1111 \xrightarrow{+1} 1000.0000 \rightarrow -128_{10}$

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 → $0{,}155625 \cdot 10^{+3} = 0{,}155625 \cdot exp_{10}^{+3}$.

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

$155{,}625 = 1\cdot2^7 +0\cdot2^6+0\cdot2^5+1\cdot2^4+1\cdot2^3+0\cdot2^2+1\cdot2^1+1\cdot2^0+1\cdot2^{-1}+0\cdot2^{-2}+1\cdot2^{-3}$

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

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.

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)

Це цілі числа які 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:

$F =(-1)^0\cdot2^{(134-127)}\cdot(1+ 1810432 / 2^{23})= 2^7\cdot(1+0{,}2158203125)=128\cdot1{,}2158203125=155{,}625$,

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

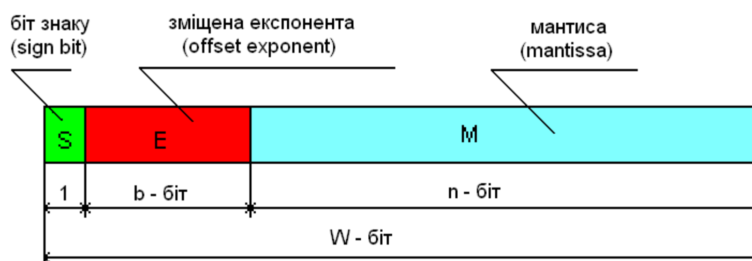


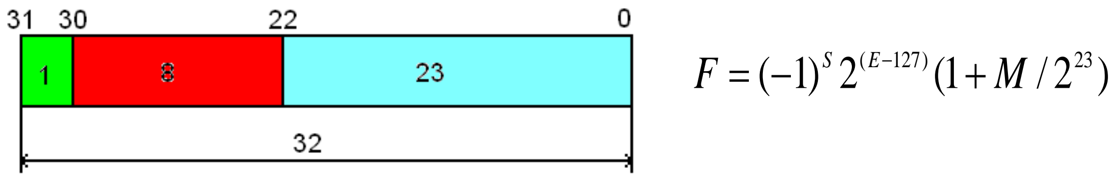
Рисунок 3 – Подання чисел в стандарті IEEE-754:

- 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-бітному IEEE-754 воно дорівнює +127, а в 64-бітному +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

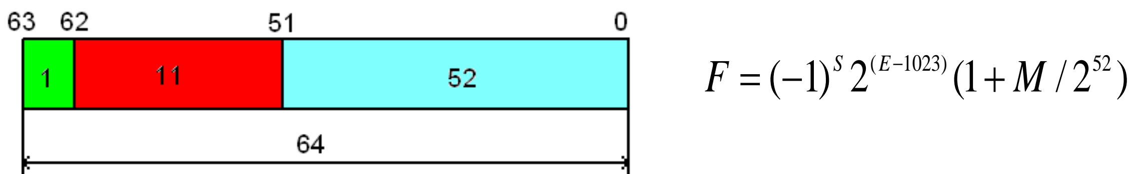
$$F=(-1)^S 2^{(E-2^{(b-1)}+1)}(1+M/2^n) \quad (1)$$

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



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

Рисунок 5 – Формат числа подвійної точності (double-precision) 64 біти

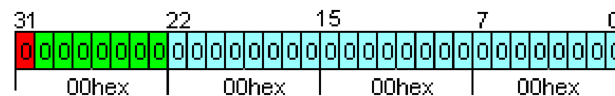
5.4. Подання денормалізованого числа та інших чисел у форматі IEEE-754

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

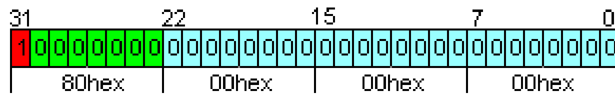
- 00 00 00 00 hex= 5,87747175411144e-39 (мінімальне позитивне число)
- 80 00 00 00 hex=-5,87747175411144e-39 (мінімальне негативне число)
- 7f ff ff ff hex= 6,80564693277058e+38 (максимальне позитивне число)
- ff ff ff ff hex=-6,80564693277058e+38 (максимальне негативне число)

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

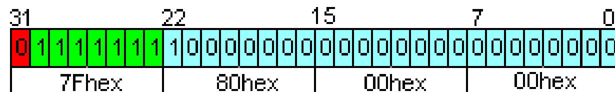
1. Число IEEE-754=00 00 00 00hex вважається числом +0:



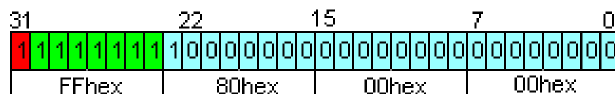
число IEEE-754=80 00 00 00hex вважається числом -0:



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

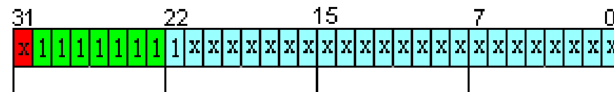


число IEEE-754=FF 80 00 00hex вважається числом -∞:

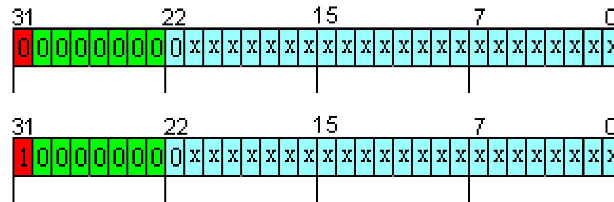


3. Числа IEEE-754=FF (1xxx)X XX XXhex не вважаються числами (NAN), крім випадку п.2, числа IEEE7- 54=7F (1xxx)X XX XXhex не вважаються числами (NAN), крім випадку п.2

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

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

Пояснення до виняткових чисел:

• Наявність двох нулів вносить симетричність.

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

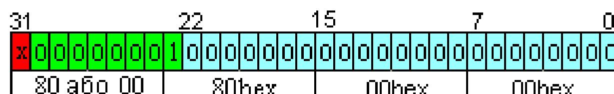
• Не числа NAN (No a Numbers) – до них відносяться символи, або результати недопустимих операцій.

• 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. Границі діапазону для чисел одинарної та подвійної точності IEEE-754

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

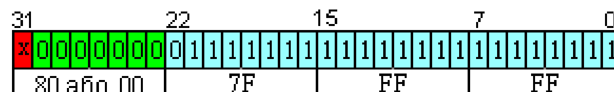
Мінімальне нормалізоване число (абсолютне):



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):

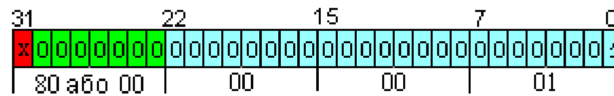


00 7F FF FF = $2^{-126}\cdot(1-2^{-23}) \approx 1{,}17549421\cdot e^{-38}$

80 7F FF FF = $-2^{-126}\cdot(1-2^{-23}) \approx -1{,}17549421\cdot e^{-38}$

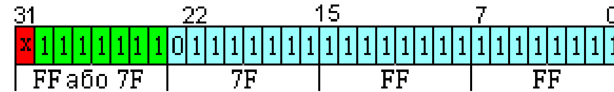
Як видно, мінімальне нормалізоване число межує з максимальним денормалізованим.

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

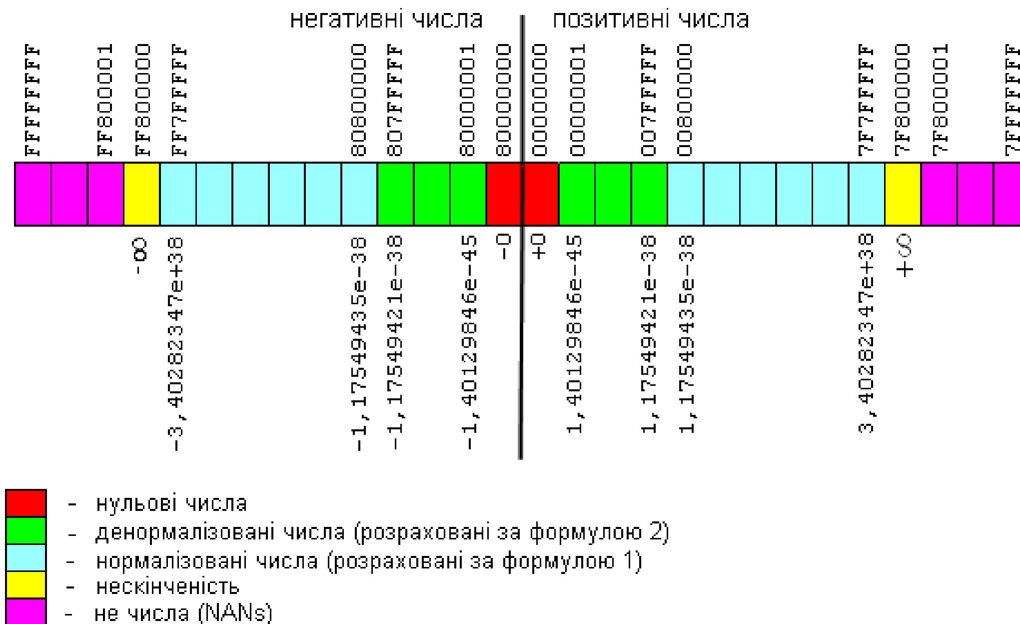


Рисунок 6 – Діапазон чисел формату одинарної точності (32 біти) за стандартом IEEE-754

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

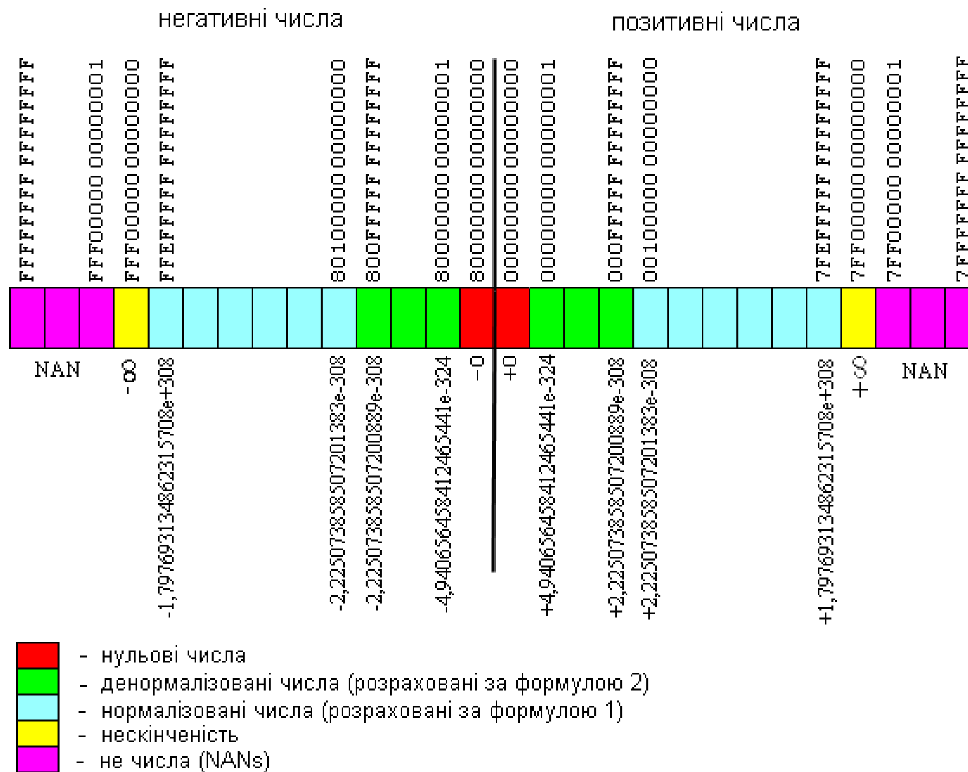


Рисунок 7 – Діапазон чисел формату подвійної точності (64 біт) за стандартом IEEE-754

5.6. Точність подання дійсних чисел у форматі IEEE7-54

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.

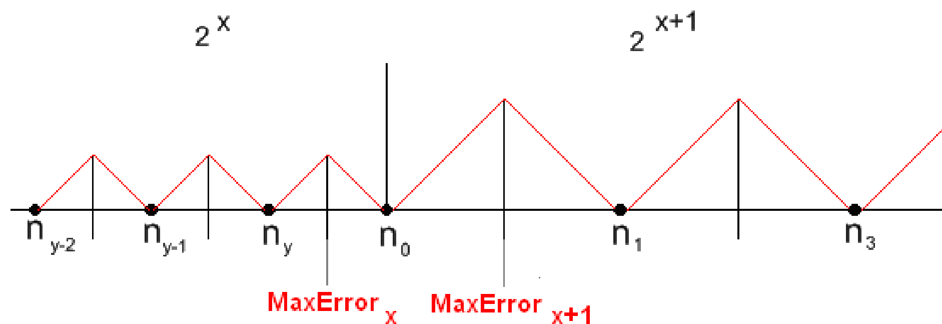


Рисунок 8 – Функція похибки точності подання числа у IEEE-754

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):

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

Максимальна відносна похибка нормалізованого числа (single):

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

Максимальна відносна похибка нормалізованого числа (double):

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

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 сскладає 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 денормілізованого 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 конвейєрі 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).

Архітектури VLIW (англ. very long instruction word – дуже довге слово команди) 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 субкоманд 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 були реалізовані 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-бітного 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:

• 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;

• добавлено APIC для підтримки багатьох процесорів;

• добавлено 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, три-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 який функціонували 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 пластикових 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 основувалися 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 основувалися 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 призначена 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 чотириядерну 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 чотириядерну 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 ІС;
- 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

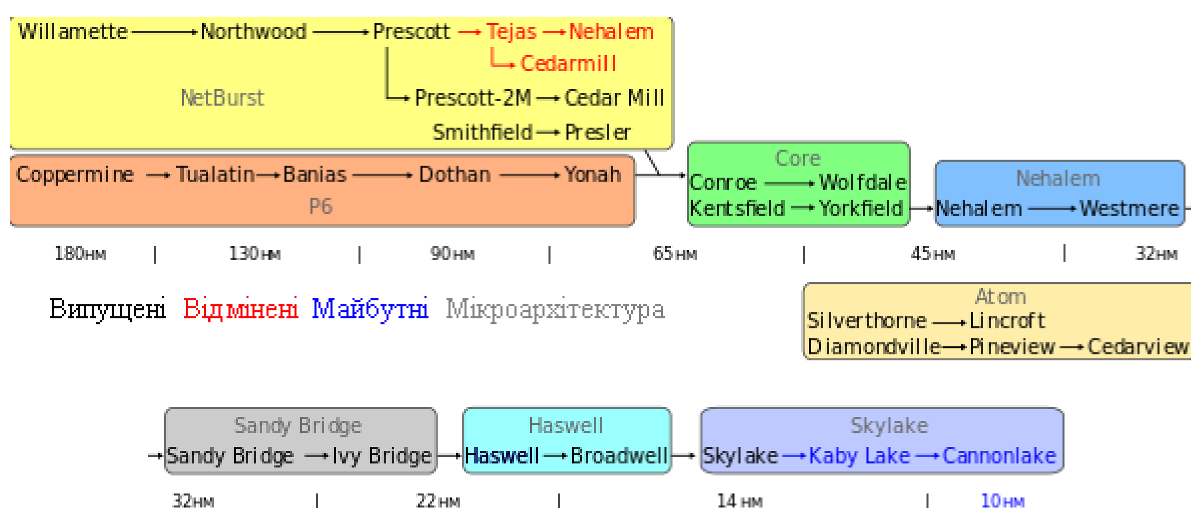


Рисунок 9 – Лінійка процесорів Intel

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 операційний режим процесора і характеристики виконуваної задачі.

• Регістри керування пам’яттю GDTR (регістр глобальної дескрипторної таблиці), IDTR (регістр таблиці переривань), TR (регістр задачі), LDTR (регістр 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 функціонування процесора.

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

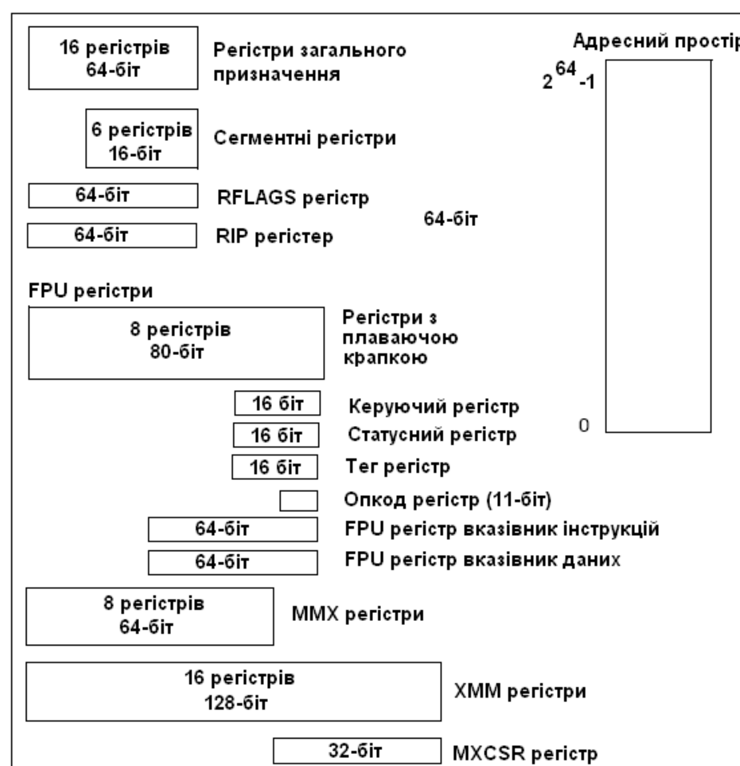


Рисунок 10 – Середовище виконання процесора x86-64

Висновки

• Системне програмування пов’язане з розробленням програм, які взаємодіють з системним програмним забезпеченням (операційною системою), або апаратним забезпеченням обчислювальної системи.

• Асемблер – це програм, яка приймає на вхід текст, який містить мнемоніки машинних команд, зручних для людини, і переводить їх у послідовність відповідних кодів машинних команд, зрозумілих процесору.

• Процесор може працювати у привілейованому режимі (режим суперкористувача) або обмеженому режимі (режим користувача).

• У Intel процесорах послідовності байтів у пам’яті зберігаються починаючи з самого молодшого байту, а у Motorola процесорах – починаючи з самого старшого байту.

• У Intel процесорах обробляються як цілі числа, так і числа з плаваючою крапкою, подані у форматі стандарту IEEE-754. Цілі знакові числа зберігаються у доповнюючому коді.

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 доповнюючий 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.

Література

1. Харт Джонсон М. Системное программирование в среде Windows / Джонсон М. Харт ; пер. с англ. — М. : Издательский дом «Вильямс», 2005.

2. Електронний ресурс: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

2. РЕЖИМИ РОБОТИ І МОДЕЛІ ПАМ'ЯТІ ПРОЦЕСОРІВ INTEL x86-32/64

Мета. Ознайомлення з режимами роботи і моделями пам'яті процесорів Intel.

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 режим плоскої моделі пам'яті
5. Довгий режим плоскої моделі пам'яті
6. Регістри процесорів x86-64

6.1. Регістри загального призначення
6.2. Сегментні регістри
6.3. Регістр вказівник команд і регістр прапорів

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-е значення	16-е 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
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-бітними.

• **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- бітний, а розмір операнду не може бути 8-байтовим.

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

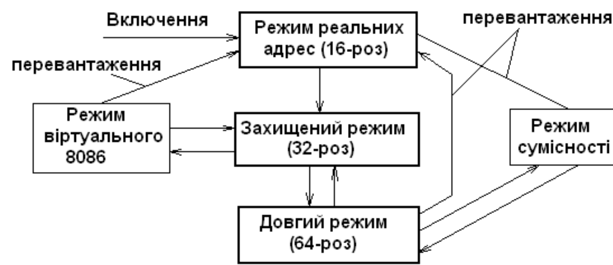


Рисунок 1 – Діаграма режимів роботи процесора x86-64

3. Реальний режим

Перші процесори 8080, 8086 і 80286 були 16-розрядними і мали *16-розрядні регістри* та забезпечували доступ до *1 Мбайта* 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 мегабайтної пам'яті. 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, як показано на рис. 2.

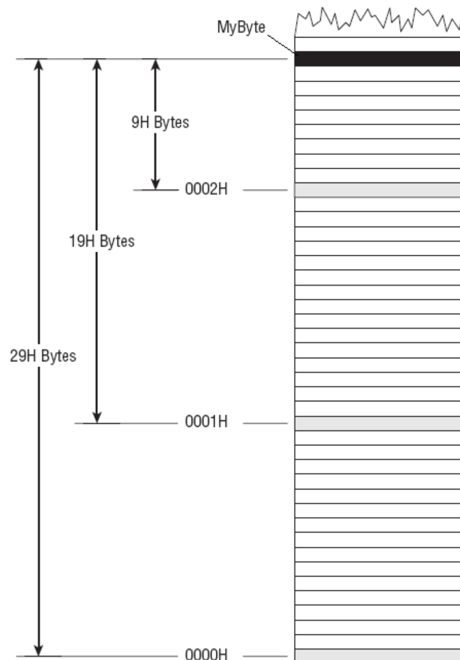


Рисунок 2 – Адресація комірки MyByte з використанням різних сегментів і зміщень: 0000h:0028h, 0001h:0019h, 0002h:0009h

Таким чином, для адресації мегабайтної 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 – сегмент коду. Машинні інструкції розміщуються з деяким зміщенням у сегмент коду. Сегмент коду містить адресу сегменту, інструкції якого виконуються в даний момент часу.

• DS – 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 сегмент даних.

• SS – сегмент стеку. Стек використовується для тимчасового зберігання даних і адрес. Стек займає сегмент і його адреса зберігається сегменті стеку.

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

• FS і GS – є копіями FS (позначення взяті з послідовності букв E, F, G). Регістри 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. 2.3. 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. Схема плоскої моделі пам'яті для реального режиму показана на рис. 3.

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. 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.

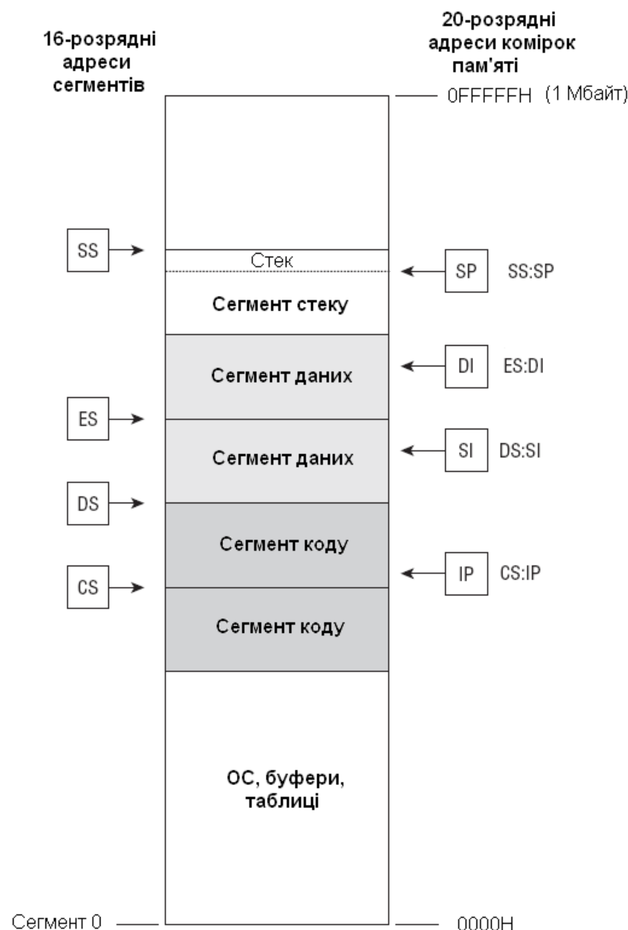


Рисунок 4 – Схема сегментованої моделі пам’яті для реального режиму:

На рис. 2.4 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 поміщенні значень у стек, їх адреси зменшуються.

Необхідно зазначити, що в реальному режимі в 1 Мбайті 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 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 застосування.

Схема захищеного режиму плоскої моделі пам'яті показана на рис.2.5. Всі регістри загального призначення і регістр вказівник команд EIP (в позначеннях 32-розрядних регістрів добавлена буква E) є 32-розрядними, тому вони потенційно мають доступ до всього адресного простору від 0 до 4 ГБайт пам'яті. 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.

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

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 80H.

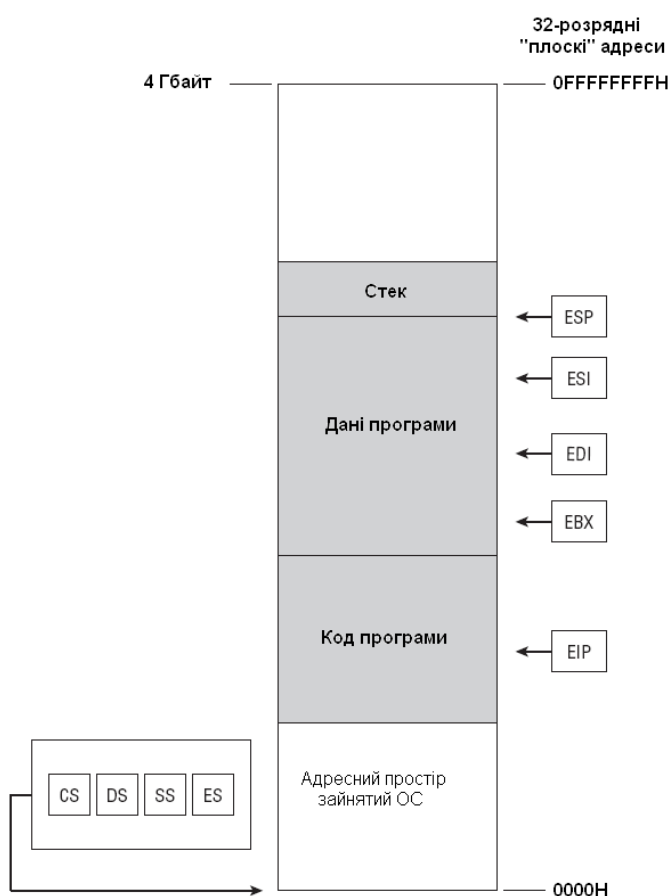


Рисунок 5 – Схема плоскої моделі пам'яті для захищеного режиму

5. Довгий режим плоскої моделі 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 режимі процесор працює як дійсно 64-розрядний процесор. Всі регістри процесора є 64-розрядними і назви їх починаються з букви R, так EAX став RAX, EBX став RBX і т. д. Додатково, порівняно з IA-32

процесорами добавлено регістри загального призначення R8-R15 і 8-ім 128-розрядних регістрів SSE.

64-розрядний регістр може 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**;
- **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 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 (IP в реальному режимі) – вказівник інструкції;

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

• 32-розрядний регістр прапорів – EFLAGS;

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

• 64-бітні 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 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-бітні регістри математичного співпроцесора ST0-ST7;

• 64-бітні MMX-регістри (MM0-MM7);

• 128-розрядні XMM-регістри XMM0-XMM15 і 32-бітний MXCSR;
• 64-розрядні регістри керування CR0-CR4 і CR8; регістри-вказівники системних таблиць GDTR, LDTR, IDTR і регістр задачі TR;
• 64-розрядні регістри налагодження DR0-DR3, DR6, DR7;
• 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.

32-розрядні регістри загального призначення використовуються як:

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

До цих регістрів можна звертатися по "частинах". Наприклад до молодших 16-біт регістра `eax` можна звернутися як до `ax.ax`. Вони в свою чергу містять дві однобайтні половинки, до яких можна звернутися як до `ah` (старшого), `al` (молодшого) байту.

Інструкції, в яких використовуються регістри з префіксом "`r`" дають доступ до байтових регістрів. Для цього старші частини байтів `ah, dh, ch, bh` замінюються молодшими частинами наступних регістрів `sil, dil, spl, bpl`.

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)

6.2. Сегментні регістри

До сегментних регістрів відносяться наступні:
• **cs** (Code segment) – сегмент коду, містить адресу сегменту з машинними командами.

• **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 ESP (RSP) 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, EBP (RBP) звичайно вказує на початок фрейму у стеку, де зберігаються локальні дані підпрограм. Вказівник інструкції EIP (RIP) напряму використовувати неможна, так як він використовується процесором. Регістри STn, MMn, XMMn використовуються математичним співпроцесором при роботі з числами з плаваючою крапкою. Регістри CRn, DRn, регістри-вказівники системних таблиць, MSR-регістри є системними і керуються ключовими механізмами роботи процесора.

6.3. Регістри стану - вказівник команд і регістр прапорів

До регістрів стану відносяться регістр вказівник інструкцій і регістр прапорів.

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

Регістр прапорів 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 адресів до молодших;

0 – напрямок “вперед”, від молодших адресів 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.

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-бітну 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 мегабітними сторінками (PSE);
• 5-біт, включає 36-бітний 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, 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.

Висновки.

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

• 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).

• Всі регістри процесора x86-32/64 поділяються на три групи: регістри загального призначення, сегментні, стану і керуючі.

• Регістри загального призначення також поділяються на три групи: регістри даних, регістри вказівники і індексні регістри.

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.

Література.

1. Аблязов Р.З. Программирование на ассемблера на платформе x86-64. – М.: ДМК Пресс, 2011. – 304 с.

2. Столяров А.В. Программирование на языке асемблера NASM для ОС UNIX. Уч. пособие. – 2-е изд. – М.: Макс-пресс, 2011. – 188 с.

3. Магда Ю.С. Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006. — 410 с.

3. ОСНОВИ АСЕМБЛЕРА

Мета. Ознайомлення з засобами компіляції і налагодження асемблерних програм, створенням виконуваних файлів, типами даних.

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 шістнадцяткові 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

Асемблер, як мова програмування, використовується для:
- написання інтерфейсних програм до ОС, процесора і BIOS;
- падання даних у пам'яті і зовнішніх пристроях;
- отримання програм менших за розміром і швидших у виконанні;
- розв'язання критичних за часом задач;
- генерації машинного коду в трансляторах.

Для розроблення програм на асемблері для процесорів x86-64 використовуються різні асемблери NASM, 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 виснити де знаходиться `nasm` командою консолі:

```
> whereis nasm
```

/usr/bin/nasm

Якщо `nasm` відсутній то потрібно його інсталювати із сайту розробників. 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`. При асемблюванні найчастіше використовуються наступні ключі:

`-h` – довідка про ключі `nasm`
`-hf` – довідка про вихідні формати файлів
`-g` – генерувати відлагоджувальну інформацію у вибраному форматі
`-e`– виконати тільки препроцесорну обробку
`-a`– тільки асемблювати (без препроцесорної обробки)
`-o` – задання імені вихідного файлу;
`-f` – задання формату вихідного файлу
`-F` – вибір формату відлагоджувальної інформації
`-l` – генерація файлу роздруку;
`-I` – добавити каталоги для пошуку файлів, які підключаються
`-v` – версія програми

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

– загальний формат аргументів nasm
```
>nasm -f <format> <filename> [-o <output>]
```

– асемблювання програми в об'єктний файл формату ELF
```
>nasm -f elf program.asm
```

– 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 об'єктний файл формату ELF з вставленням налагоджувальної інформації для зневадника (опція `-g`), формат генерування налагоджувальної інформації (`-F stabs`)
```
>nasm -f elf -g -F stabs  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
```
>nasm -f bin program.asm -o myfile.com
```

– асемблювання програми з отриманням файлу роздруку з hex-кодами
```
>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
```

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-бітні процесори, а x86_64, amd64 – на 64-бітні .

Запуск програми на виконання

```
>./program
```

Запуск програми на виконання у консольному налагоджувачі KDBG

```
>gdb program
```

Команди для асемблювання і компонування можна оформити як сценарії. Приклад сценаріїв для асемблювання (`debug.sh`) і компонування 32-розрядного бінарного файлу для роботи із графічним зневадником KDbg (`load.sh`). Після асемблювання створюється об'єктний файл з розширенням *.o.

Сценарій асемблювання `debug.sh`:

```
#!/bin/bash
echo "Програма " $1
nasm -f elf32 -g -F stabs $1
```

Запуск сценарію на асемблювання

```
>./debug.sh 1.asm
```

Сценарій компонування `load.sh`:

```
#!/bin/bash
ld -m elf_i386 $1 -o test
```

Запуск сценарію на компонування

```
>./load.sh 1.o
```

Після компонування створюється бінарний файл `test`, який можна завантажити у налагоджувач. Для налагодження програм потрібно запустити графічне середовище KDbg, рис. 1.

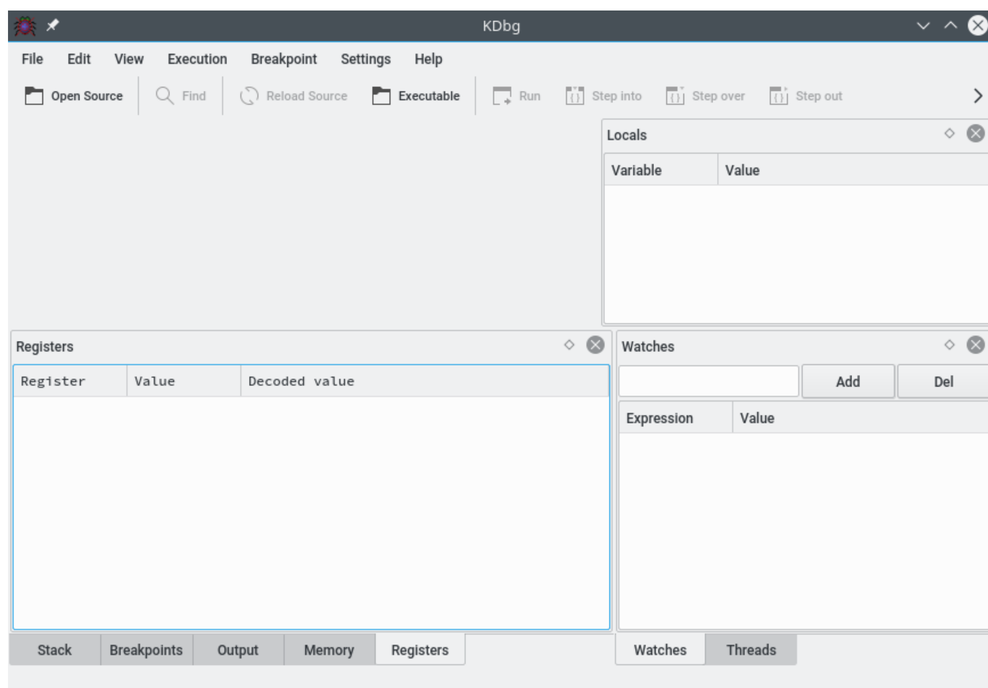


Рисунок 1 – Графічне середовище налагоджувача KDbg

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**.

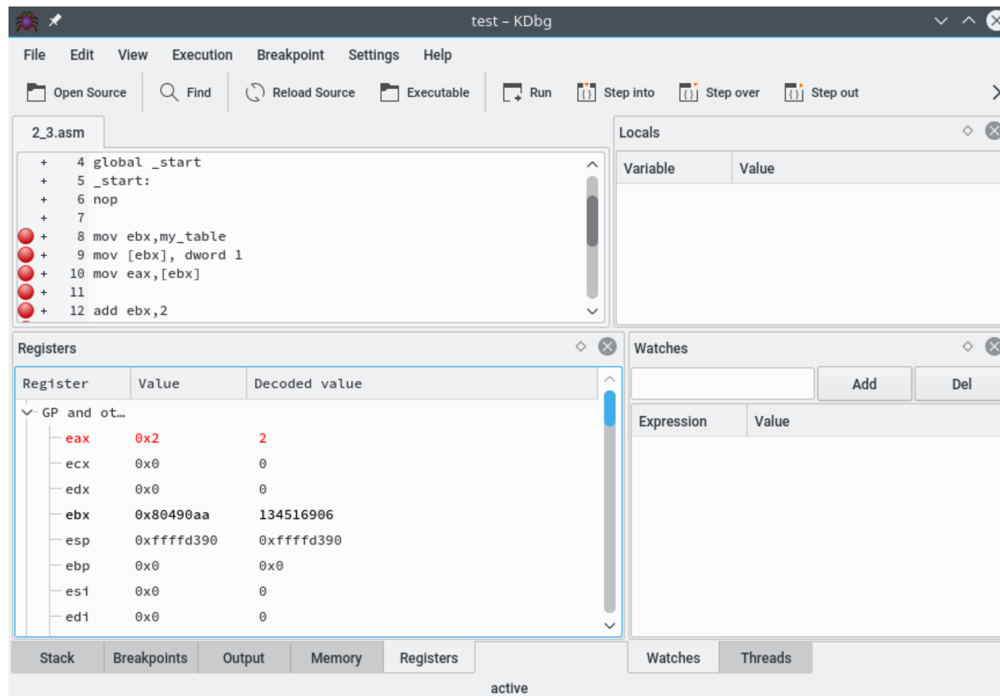


Рисунок 2 – Налагодження програми

2. Послідовність створення виконуваних файлів

Послідовність створення виконуваних файлів показана на рис. 1.

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

- початкові 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.

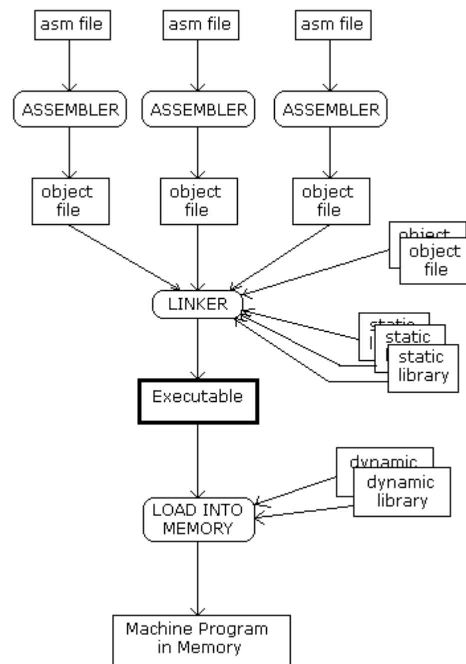


Рисунок 3 – Послідовність створення виконуваних файлів асемблера

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 речення програми, які замінюються під час асемблювання іншими реченнями.

• 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.

4. Константні типи даних

Асемблер розпізнає наступні типи констант: числові, символьні, стрічкові, з плаваючою крапкою, стрічки UNICODE, запаковані BCD числа.

4.1. Числа

Числові константи є просто числа. Числа в 2-й, 8-й, 10-й і 16-й системах можна задати з використанням префіксів або суфіксів:

`0b11110001, 0b111_1000, 0y111_1000, 111110001b, 1111_10001b,` – двійкове ціле

`0o1234567, 01234567o, 01234567q` – вісімкове ціле

`200, 0200, 0200d, 0d200` – десяткове ціле

`0x1A2B3C4E5F, 1A2B3C4E5Fh, $0A2B3C4E5F` – шістнадцяткове ціле (при використанні символу `$` потрібно слідкувати, щоб зразу після нього стояла цифра, а не буква, наприклад,

`$0f5` замість `$f5`. Аналогічно необхідно слідкувати і за першим символом при використанні символу `h`, наприклад, `0a51h`, а не `a51h`). Приклади пересилання різних чисел у регістр `ax`:

```
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 4-х (i386) або до 8-ми (x86-64) символів взятих у одинарні (‘…’), подвійні (“…”) або лівонахилені лапки (`…`).

У стрічки, які знаходяться у лівонахилених лапках, можна поміщати esc-послідовності у C-стилі:

```
\' одинарні лапки (')
\" подвійні лапки (")
\` одинарні лівонахилені 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 як `\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 стрічку довжиною до 8 байт. Символьна константа довжиною більшою як один байт розміщується у регістрі і пам’яті з порядком байтів “little-endian”:

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

4.4. Стрічкові константи (string constant)

Стрічкові константи використовуються у директивах виділення і ініціалізації пам'яті:

```
db 'hello' ; стрічкова константа
db 'h','e','l','l','o' ; еквівалентна стрічкова константа
dd 'ninechars' ; стрічкова константа doubleword
dd 'nine','char','s' ; стрічкова константа з трьох doublewords
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 директив `DB`, `DW`, `DD`, `DQ`, `DT` і `DO`:

```
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
```

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 +0p33
dt 33p
```

5. Псевдоінструкції і директиви

Псевдоінструкції і директиви не виконуються процесором. Вони самі виконують якусь дію, яка не траслюється 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 оголошення, ініціалізації і резервування пам’яті

До псевдоінструкцій відносяться:

- `DB, DW, DD, DQ, DT` для оголошення і ініціалізації пам’яті в секції `.data`;
- `RESB, RESW, RESD, RESQ, REST` для резервування пам'яті в секції `.bss`;
- `INCBIN` створення області пам’яті заповненої даними із зовнішнього (графічного або акустичного) файлу;
- `EQU` визначає символ для заданого константного значення;
- префікс `TIMES` повторює інструкцію, яка асемблюється, задане число разів.

Псевдоінструкції оголошення ініціалізованих даних 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, чотирне слово, десять байтів. Прийнято позначати виділені області пам’яті позначками (змінними), які будуть асоційовані з адресою першого байта.

```
section    .data
L1 db 0       ; byte позначений L1 і ініціалізований значенням 0
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 (character константа)
L9 dd 0x12345678  ; 0x78 0x56 0x34 0x12
L10 dd 1.234567e20 ; floating-point константа
L12 dd 1A92h   ; double word ініціалізований шістнадцятковим значенням 1A92
L13 dq 0x123456789abcdef0 ; 8-ми byte константа
L14 dq 1.234567e20 ; double-точності float
```

```
L15 dt 1.234567e20 ; extended-точності float
```

Для задання символу, його потрібно взяти в одинарні або подвійні лапки. Аналогічно можна задати символьний рядок. Всередині подвійних лапок, одинарні лапки розглядуються 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.

Для довгих послідовностей використовується псевдоінструкція **times**. Префікс **times** повторює інструкцію при асемблюванні задане число разів:

```
L12 times 100 db 0 ; виділенн 100 байтів ініціалізованих нулями
zerobuf times 64 db 0
```

Причому TIMES не є константою, а виразом:

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

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

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

Псевдоінструкції резервування пам'яті повідомляють асемблеру про потреби в оперативній пам'яті. Вони поділяються на два види: псевдоінструкції резервування неініціалізованої пам'яті і псевдоінструкції задання початкових даних.

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

```
section   .bss
bufflen    equ 16   ; задання розміру буфера
buff resb, bufflen  ; за адресою buff буде розміщено буфера розміром 16 байт
L13       resw 100  ; за адресою L13 буде виділено 100 слів
x         resd      ; за адресою x буде розміщене подвійне слово
 buffer    resb 64   ; резервувати 64 bytes
 wordvar   resw 1    ; резервувати word
 realarray resq 10   ; масив 10 значень подвійної точності
 ymmval    resy 1    ; один YMM регістр
```

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
```

6. Використання позначок

Позначки використовуються в секції коду для галуження програми. Позначка **`label:`** є змінною (показчиком, вказівником, посиланням) і **задає адресу пам'яті**. Позначка **`.label:`** є локальною. Якщо позначка взята в квадратні дужки **`[label]`**, то вона задає **значення за адресою** `label`.

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).*

Приклад використання позначок в 32-бітному 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
    mov eax, [L6] ; копіювати подвійне слово за адресою L6 у регістр EAX
    add eax, [L6] ; EAX = EAX + подвійне слово за адресою L6
    add [L6], eax ; до подвійного слова за адресою L6 додати регістр EAX
    mov al, [L6];копіювати перший байт подвійного слова за адресою L6 у регістр
AL
```

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
```

Окрема спеціальна псевдопозначка **`$`**, містить поточну адресу.

7. Види адресації в командах асемблера

Простір пам'яті призначений для зберігання кодів команд і даних, для доступу до яких використовуються різні способи адресації.

Дані, які обробляються командами називаються операндами. Операнди записуються після команди через кому. 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. Місцезнаходження 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.

7.1 Безпосередня адресація

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

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

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  eax, ecx    ; скопіювати значення з регістра ecx в регістр eax
mov dx, tax_rate ; скопіювати адресу комірки пам’яті у регістр
mov count, cx    ; 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 dd 0
section .text
mov eax,addr1    ; скопіювати в регістр eax адресу addr1
add eax,2        ; збільшити адресу на 2
```

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 (EBX, BX, EBP, BP) і індексні регістри (DI, SI) для адресації комірок пам'яті. Для доступу до значень комірок регістри записуються в квадратних дужка, наприклад **[EBX].**

Операнд команди може містити адресу комірки пам'яті або регістр з адресою комірки пам'яті. Адреса комірки пам'яті задається змінною (позначкою). Для доступу до значень комірки пам'яті змінну записують у квадратних дужках, наприклад **,[num]**.

```
segment .data
num dw 1,2,3,4,5
segment .text
global _start
_start:
mov eax,[num]     ; завантажити в регістр eax значення за адресою num
mov ebx,[num+2]   ; nasm - завантажити в регістр ebx значення,
                  ; яке знаходиться за адресою num+2
mov ecx,[eax]     ; завантажити в регістр ecx значення, яке знаходиться
                 ; за адресою eax
```

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

```
my_table times 10 dw 0 ; виділення 10 word з ініціалізацією 0
mov ebx, my_table      ; ефективна адреса my_table в ebx
mov [ebx], dword 1     ; my_table[0] = 1
add ebx, 2             ; адреса ebx = адреса ebx+2
mov [ebx],dword 3      ; my_table[1] = 3
```

Використання непрямої адресації операнду в оперативної пам’яті, при зберіганні його адреси в регістровій пам’яті, суттєво скорочує довжину поля адреси, одночасно зберігаючи можливість використання для фізичної адреси повної розрядності регістра. Недолік цього способу – необхідність додаткового часу для читання адреси операнду. 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, які зберігаються в різних адресах.

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 сегментами. 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*, *із 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 =  { EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP}
INDEX = { EAX, EBX, ECX, EDX, ESI, EDI, EBP }
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 `eax/R0D, ecx/R1D, edx/R2D, ebx/R3D, esp/R4D, ebp/R5D, esi/R6D, edi/R7D, ebp, esp`.

`INDEX` – один з регістрів загального призначення, за винятком `esp/R4D`.

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

`OFFSET` – любе 32-бітне число.

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

```
segment .data
mas dw 1,2,3,4
    dw 5,6,7,8
segment .text
global _start
_start:
nop

mov ebx,mas  ; адреса початку мпсиву
mov esi,1    ; рядок 1, 2-елементи,  адреса  2*4=8
             ; рядок 2, 2-й елемент, зміщення 2
mov [ds:ebx], word 0xaa_bb
mov [ds:ebx+esi*4+4], word 0xcc_dd
```

```
mov eax,1  ; повернення коду завершення в ОС
mov ebx,0
int 0x80
```

8.1. Команда lea

Команда `lea` обчислює виконавчу адресу джерела без звернення до пам'яті і поміщає отриману адресу в *отримувач*. *Джерело* має знаходитися в пам'яті (не може бути безпосереднім значенням або регістром).

```
segment .data
var dd 0x00000072
segment .text
        lea eax,var    ; аналогічно mov eax, var
        lea eax, esp+4 ; помістить в eax адресу попереднього елемента в стеку
```

Висновки.

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

• Модуль, готовий до виконання, отримується за три кроки – асемблювання, компонування та завантажування.

• Асемблер підтримує наступні типи даних: числові, символьні, стрічкові, з плаваючою крапкою, стрічки UNICODE, запаковані BCD числа.

• Крім інструкцій, асемблер використовує псевдоінструкції і директиви, які виконують якусь дію, яка не транслюється в машинний код.

• Найбільш поширеними способами адресації пам'яті є безпосередня, регістрова, пряма, непряма.

• У загальному випадку повна (виконавча, фізична) адреса у пам'яті обчислюється з використанням регістрів загального призначення, сегментних регістрів, індексних регістрів та зміщення.

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; RESB, RESW, RESD, RESQ, REST; 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.

Література.

1. Юров В.И. Assembler. Учебник для вузов. 2-е изд. – СПб.: Питер, 2003. – 637 с.
2. Аблязов Р.З. Программирование на ассемблера на платформе x86-64. – М.: ДМК Пресс, 2011. – 304 с.
3. Магда Ю.С. Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006. — 410 с.

4. КЛАСИФІКАЦІЯ КОМАНД, КОМАНДИ ПЕРЕСИЛАННЯ ДАНИХ, АРИФМЕТИЧНІ КОМАНДИ, ЛОГІЧНІ КОМАНДИ І ОПЕРАЦІЇ, ЛАНЦЮГОВІ КОМАНДИ

Мета. Вивчення функціональної класифікації команд, команд пересилання даних, арифметичних команд, логічних команд і операцій

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 і логічні команди і операції.

План.

1. Функціональна класифікація цілочисельних 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. Ланцюжкові команди

1. Функціональна класифікація цілочисельних 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 можна структурувати за наступними групами:

- команди процесора;
- команди співпроцесора;
- MMX цілочисельне розширення (мультимедійні застосування);
- XMM розширення з плаваючою крапкою.

Функціональна класифікація цілочисельних команд показана на рис. 1.

Рисунок 1 – Класифікація цілочисельних 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-байтну область пам’яті, яка знаходиться за адресою `x`, можна так:

```
mov [x], dword 15
; або так
mov dword [x], 15
```

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 eax,ebx          ; обмін вмісту регістрів eax, ebx

xchg al,ah            ; обмін місцями значень al, ah
xchg ax, word [xsi]   ; 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, доступ до яких здійснюється 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 порта 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 зворотньому 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:

- `lea отримувач, джерело` – завантаження ефективної адреси;
- `lds отримувач, джерело` – завантаження вказівника у регістр сегмента даних `ds`;
- `les отримувач, джерело` – завантаження вказівника у регістр додаткового сегмента даних `es`;
- `lfs отримувач, джерело` – завантаження вказівника у регістр додаткового сегмента даних `fs`;
- `lgs отримувач, джерело` – завантаження вказівника у регістр додаткового сегмента даних `gs`;
- `lss отримувач, джерело` – завантаження вказівника у регістр додаткового сегмента даних `ss`;

`Команда lea` (мнемоніка від англ. Load Effective Address) обчислює 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 eax,[Msg]

lea eax,[1000+ebx+8*ecx]
```

Команда множить значення регістра ecx на 8, додає до добутку значення з регістра ebx та число 1000 і отриманий результат пересилає у регістр eax.

2.4. Перекодування даних

Команда xlat замінює значення в регістрі al іншим байтом з таблиці перекодування 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-байтної таблиці перекодування.

```
hex_table times 256 db 'a'  ; таблиця перекодування
symbol db 'x'               ; символ для перекодування
mov ebx,hex_table
mov al,[symbol]
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:sp (ss:esp).

Для роботи зі стеком призначені наступні регістри:

- ss – сегментний регістр стеку;
- sp/esp – регістр вказівник верхівки стеку;
- bp/ebp – 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 esp/sp, ebp/bp для адресації пам'яті, асемблер автоматично приймає їх вміст як зміщення відносно регістра ss.
- регістр esp/sp завжди вказує на верхівку стеку, тобто містить зміщення за яким був записаний останній елемент;
- регістр 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. Це адреса повернення, параметри і локальні змінні однієї підпрограми з адресою верхівки стеку, яка записується у регістр ebp. Значення з ebp використовується для доступу до любих елементів кадру стека.

Команди роботи зі стеком:

- занесення у стек регістрів загального призначення у наступному порядку (e)ax, (e)cx, (e)dx, (e)bx, (e)sp, (e)bp, (e)si, (e)di:

• push reg ; де reg := 16-,32-,64-біт,ds,cs,ss,es,fs,gs регістри

• push mem ; де mem := 8-,16-,32-,64-біт комірки пам'яті

- занесення у стек 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 (у вказаному порядку, значення esp у стані до виконання команди):

• pusha – всі регістри 16-бітові
pushaw - всі регістри 32-бітові (e)
pushad - всі регістри 64-бітові (r)

• pushf, pushfw, pusfd ; занесення у стек регістра прапорів 16-, 32- і 64-бітного

- 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

• popof, popfw,popfd ; відновлення вмісту регістра прапорів 16-, 32- і 64-бітного

Команду push eax можна реалізувати за допомогою пари команд:

```
sub esp,4   ; eax 4-байтний регістр
mov [ss:esp],eax  ; запис eax в стек
```

Команду pop eax можна реалізувати за допомогою пари команд:

```
mov eax,[ss:esp]  ; помістити в eax верхівку стек
add esp,4         ; вилучити останнє значення типу dword із стека
```

3. Арифметичні команди

Процесор може виконувати цілочисельні операції і операції з плаваючою крапкою. Для цього у його архітектурі є два окремих блоки:

- пристрій для виконання цілочисельних операцій;
- пристрій для виконання операцій з плаваючою крапкою.

Кожен з цих пристроїв моє свою систему команд. Пристрій для виконання цілочисельних команд підтримує команди для роботи з двома типами чисел:

- цілі двійкові числа з знаком або без знаку;
- цілі десяткові числа.

Арифметичні команди асемблера поділяються на наступні групи:

• цілочисельної 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 (neg, not, bswap, cmp, cmpxchg, 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 fecimal). 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.

3.1. Команди перетворення типу

Операнди команд арифметичних операцій можуть мати різні розміри. Для розв'язування проблем, які виникають у таких випадках використовуються команди перетворення типу. Ці команди розширюють байти у слова, слова – у подвійні слова, подвійні слова – в почотирні слова. Команди перетворення типу особливо корисні при перетворенні цілих із знаком, так як вони автоматично заповнюють старші біти ново формованого 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 – перетворення байта у регістрі al у слово, шляхом поширення значення старшого біта al на всі біти регістра ah;

• cwd – перетворення слова у регістрі ax у подвійне слово в регістрі dx:ax, шляхом поширення значення старшого біта ax на всі біти регістра dx;

• cwde – перетворення слова у регістрі ax у подвійне слово в регістрі eax, шляхом поширення значення старшого біта ax на всі біти регістра eax;

• cdq – перетворення подвійного слова у регістрі eax у почотирне слово в регістрі edx:eax, шляхом поширення значення старшого біта eax на всі біти регістра edx;

2. Команди movsx, novzx, які відносяться до команд оброблення стрічок. Ці команди мають корисні властивості, що використовуються при перетворенні типів:

• movsx 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, reg8/mem8      ; 8-й біт розширюється  до 16-го біту
movsx reg32, reg8/mem8      ; 8-й біт розширюється  до 32-го біту
movsx reg32, reg16/mem16    ; 16-й біт розширюється до 32-го біту
```

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;

`add операнд1, операнд2` – команда додавання за принципом дії `операнд1 = операнд1 + операнд2;`

`adc операнд1, операнд2` – команда додавання з врахуванням прапора перенесення `cf` `операнд1 = операнд1 + операнд2 + значення_cf;`

Команди віднімання цілих чисел без знаку:

`dec операнд1` – операція декременту, тобто зменшення значення операнда на 1;

`sub операнд1, операнд2` – команда віднімання за принципом дії `операнд1 = операнд1 - операнд2;`

`sbb операнд1, операнд2` – команда віднімання з врахуванням прапора позики `cf` `операнд1 = операнд1 - операнд2 - значення_cf;`

Приклад:

```
add edx,12         ; збільшити на 12 вміст регістра edx
add eax,ebx        ; до значення регістра eax додати значення регістра ebx
add dword [x],12   ; до значення 4-х байтної комірки з адресою x додати 12
sub [x],ecx        ; із значення 4-х байтної комірки з адресою x відняти значення
                   ; регістра ecx
```

В результаті виконання команд `add`, `sub` виставляються прапори `ZF`, `SF`, `OF`, `CF`.

`ZF=1 (нуля)`, 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 64-бітні числа, перше записано в пару регістрів `edx:eax`, а друге – в `ebx:ecx`. Тоді додати ці два числа можна командами

```
add eax, ecx ; додавання молодших частин
adc edx, ebx ; додавання старших частин з врахуванням перенесення
```

якщо потрібно їх відняти то використовуються команди

```
sub eax, ecx ; віднімання молодших частин
sbb edx, ebx ; 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 eax              ; зменшити значення регістра eax на 1
inc dword [count]    ; необхідно вказати розмір операнда
```

Приклад організації циклу:

```
mov eax,5
DoMore: dec eax
...
jnz DoMore
```

Команди цілочисельного множення і ділення.

Команди цілочисельного 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 `eax/ax/al`, а при необхідності і регістрові пари `dx:ax`, `edx:eax`. Необхідно зауважити, що результат множення двох n-розрядних чисел може поміститися тільки в $2n$-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`

Розряди явного операнда	Множення			Ділення			
	неявний множник2	явний множник1	результат множення	неявне ділене	явний дільник	Част-ка	Зали-шок
8	al	reg8/mem8	ax	ax	reg8/mem8	al	ah
16	ax	reg18/mem16	dx:ax	dx:ax	reg18/mem16	ax	dx
32	eax	reg32/mem32	edx:eax	edx:eax	reg32/mem32	eax	edx

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.

Для ділення беззнакових чисел використовується команда `div`, а для ділення знакових – `idiv`. Значення прапорів після операцій цілочисельного ділення не визначені. Приклади:

```
mov ax,888   ; неявний множник2
mov bx,77    ; явний множник1
mul bx       ; 68376=0x10b18  (результат) => edx:eax (1:0b18)
mov dx,0x1   ; неявне ділене dx:ax
mov ax,0xb18 ; неявне ділене dx:ax
mov bx,77    ; явний дільник
div bx       ; 888 (результат) => ax
```

3.3. Арифметичні операції над двійково-десятковими числами

Двійково-десяткові числа (BCD-числа) використовуються у застосуваннях, 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 не вірний
	+ 0110	Корекція – додати в молодшу тетраду 6
	= 0001_0011	

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 пропор `cf` скидається у 0, інакше – у 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	Результат не вірний. Має бути 16-7=9
	– 0110	Корекція – відняти від молодшої тетради 6
	= 1111_1001	

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) – керекція результату додавання

Команда `daa` перетворює вміст регістра `al` у дві запаковані десяткові цифри за алгоритмом, який наводиться у [1]. 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

Алгоритмом, за яким команда `das` корегує результат, наводиться у [1].

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
```

Фрагмент програми з результатом виконання команд `neg` і `not`:

```
  mov ecx,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 eax,42
neg eax
add eax,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	80000000h	2147483647	7FFFFFFFh

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 eax      ; 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 агентів.

```
; блокування
wait:
mov eax,-1
mov ecx,5            ; номер процесора
cmpxchg [mem],ecx    ; порівняння з [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`.

4. Класифікація логічних команд і операцій

До засобів процесора для організації роботи з даними за правилами формальної логіки відносяться наступні команди і операції:

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:
    - зсуву логічного, арифметичного (`sar, sal, shl, shr, shld, shrd`);
    - зсуву циклічного (`rcl, rcr, rol, ror`);

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

- 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

- 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

• 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

Синтаксис логічних команд:

and операнд1, операнд2 – команда логічного множення виконує порозрядну операцію І над бітами операндів операнд1, операнд2. Результат записується в операнд1.

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 end. Початковий стан регістрів невідомий.

```
and eax,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.

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 eax,10b       ; встановити 1-й біт в регістрі eax
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.

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 eax,10b         ; інвертувати 1-й біт в регістрі eax
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 eax, eax  ;  розмір команди 2 байти
mov eax, 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 eax,eax  ; 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 eax,eax
je is_zero
   ; eax != 0 */
is_zero:
   ; eax == 0
```

Для порівняння операнда з нулем рекомендується використовувати команду test eax,eax замість команди cmp eax,0

not операнд – команда логічного заперечення операнд виконує порозрядне інвертування бітів операнда. Результат записується в операнд.

```
not eax   ; інвертувати біти регістра eax
```

4.2. Бітові операції над виразами

Над виразами асемблера можна виконувати:

- 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 **$**, **$$**.

Вирази 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 реалізовуються 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.

`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.

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 "висунутий" 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 висовуваним бітом.

`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.
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 переносу 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`.

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.

Команди циклічного зсуву через прапор переносу cf відрізняються від команд простого циклічного зсуву тим, що зсовуваний біт не зразу попадає в операнд з іншого кінця, а записується спочатку у прапор переносу cf.

Існує ще один вид зсувів – циклічний зсув через прапор cf. Ці команди використовують прапор cf як продовження операнда.

rcl операнд, лічильник_зсувів (rotate through carry left) – циклічний зсув вліво через перенос. 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 перенос. 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.

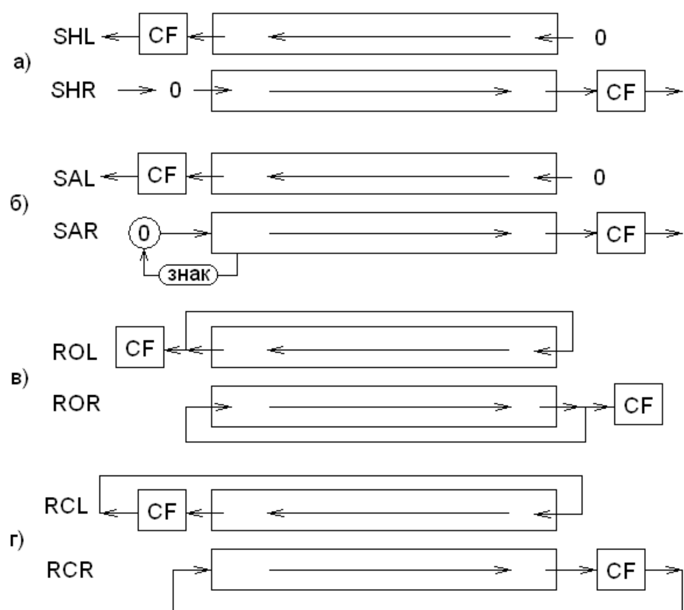


Рисунок 2 – Команди зсуву: а) логічного лінійного; б) арифметичного лінійного; в) простого циклічного; г) циклічного через прапор CF

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. Ланцюжкові команди

Ланцюжкові 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. Ланцюжкові команди використовують регістри esi і edi.

Загальна ідея ланцюжкових команд полягає у тому, що читання з пам'яті здійснюється за адресою з регістра esi, а записування – за адресою з регістра edi, а потім значення цих регістрів одночасно збільшуються (або зменшуються) в залежності від команди на 1, 2, 4. Напрямок зміни адрес визначається прапором напрямку df. Встановити прапор напрямку df можна командою std, а скинути – командою cld.

Всі ланцюжкові команди можна розділити на групи.

Команди пересилання. Формат команд:

movsb, movsw, movsd, movsq – копіювання ланцюжка байтів, слів, подвійних слів, почотирних слів, які адресуються регістрами esi і edi, [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 esi і edi 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 esi і edi. Після виконання команди вміст регістрів esi і edi 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 esi з одним з регістрів al, ax, eax 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 **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 esi в регістри al, ax, eax 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 edi елементи з al, ax, eax в залежності від команди. Після виконання команди вміст регістра edi 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 repe – до першого неспівпадіння елементів.

Рішення про циклічне виконання ланцюжкової команди приймається на основі стану регістра `ecx/cx` (`repe/repne`) або прапора `zf` (`repz/repnz`).

Так команда `rep stow` заміняє блок команд:

```
Clear:  mov es:[di],ax ; Copy AX to ES:DI
inc di                 ; збільшення індекса на 2 для word уп буфері,
inc di
dec cx                 ; зменшення cx на 1
jnz Clear              ; цикл поки cx не 0
```

Висновки.

Основна команда копіювання даних – `mov`. Операнди цієї команди мають бути узгодженими за розрядністю.

Керування периферійними пристроями здійснюється за допомого двох команд введення і виведення – `in` і `out`.

В процесі роботи можна отримати як повну, так і ефективну адресу пам'яті.

Архітектура процесора підтримує специфічну, але ефективну структуру – стек, а система команд підтримує всі необхідні операції зі стеком.

Мінімально адресована одиниця даних у процесорі – байт. Логічні команди дозволяють маніпулювати окремими бітами комірок пам'яті і регістрів, чим і пояснюється їх важливість. Можливість роботи на бітовому рівні дозволяє в окремих випадках суттєво зекономити пам'ять.

Команди зсуву дозволяють виконати швидке множення і ділення операндів на степені двійки, а також ефективно перетворювати дані.

Застосування команд циклічного зсуву і зсуву подвійної точності дозволяє реалізувати максимально швидкі операції по переміщенню, вставлянню і видобуванню бітових підрядків.

Ланцюжкові команди призначені для роботи з послідовностями байтів в пам'яті.

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. Ланцюжкові команди.

Література.

1. Юров В.И. Assembler. Учебник для вузов. 2-е изд. – СПб.: Питер, 2003. – 637 с.
2. Аблязов Р.З. Программирование на ассемблера на платформе x86-64. – М.: ДМК Пресс, 2011. – 304 с.
3. Магда Ю.С. Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006. — 410 с.

5. КОМАНДИ ПЕРЕДАЧІ КЕРУВАННЯ

Мета. Вивчення групи команд передачі керування

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.

План.
1. Засоби передачі керування
2. Безумовні переходи
2.1. Виклики підпрограм і повернення
2.2. Організація стекового фрейму
3. Умовні переходи
3.1 Прості умовні переходи
3.2. Переходи за результатами порівнянь
3.3. Команди циклів

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 деякому 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:eip`, де `cs` – сегментний регістр коду, `eip` – регістр вказівник команд.

За принципом дії команди переходів можна розділити на три групи:

1. *Команди безумовної передачі керування*:
   • команда безумовного переходу (`jmp`);
   • виклик процедури і повернення з процедури (`call, ret`);
   • виклик програмних переривань і повернення з програмних переривань.
2. *Команди умовної передачі керування*:
   • команди переходу `jxx` за результатами команди порівняння `cmp`;
   • команда переходу за станом визначеного прапора `eflags/flags`;
   • команди переходу за вмістом регістра `ecx`.
3. *Команди керування циклом*:
   • команди організації циклу `loop` з лічильником `ecx`;
   • команди організації циклу `loope/loopz, loopne/loopnz` з лічильником `ecx` і можливістю термінового виходу з циклу за додатковою умовою.

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 сегмента, в якому описана позначка;

• тип позначки або атрибут віддалі.

2. Безумовні переходи

Синтаксис команди безумовного переходу:

```
jmp [атрибут_віддалі] адреса_переходу (задається як позначка)
```

Приклади переходів:

```
jmp cycl     ; перехід на позначку cycl
jmp eax      ; перехід за адресою, яка міститься в регістрі eax
jmp [addr]   ; перехід за адресою, яка міститься в пам'яті за адресою addr
jmp [eax]    ; перехід за адресою, яка знаходиться а регістрі eax
```

При переходах всередині сегменту атрибут віддалі позначки може бути `short` або `near`. Переходи `short` можуть бути в межах -128÷127 (діапазон знакового 8-розрядного числа). Переходи `near` можуть мати 2-байтне зміщення в 16-розрядних сегментах і 4-байтне зміщення в 32-розрядних сегментах. Команда переходу всередині сегменту виконує наступну дію:

```
eip = eip +зміщення
```

При міжсегментних переходах атрибут віддалі позначки `far`. У цьому випадку команда переходу виконує дію:

```
cs = адреса сегменту в який здійснюється перехід
eip = 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*. 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
mes db "Привіт світ",0
len_mes =$-mes   ; довжина стрічки mes
```

2.1. Виклики підпрограм і повернення

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`.

Команди повернення керування викликаючій програмі:

```
ret
ret [число]
```

У своїй найпростішій формі вона не має аргументів. Виконуючи цю команду, процесор видобуває 4 байти з верхівки стеку і записує їх в регістр `eip`, в результаті чого керування передається за адресою, яка знаходилася в пам'яті на верхівці стеку.

У формі з параметром команда зчитає адресу повернення із стеку і записує її в регістр `cs:eip`. Параметр `[число]` задає кількість елементів, які вилучаються зі стеку.

2.2. Організація стекового фрейму

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 передача параметрів через загальну область пам'яті або регістри унеможливлює використання рекурсії.

Тому звичайно у складних програмах (особливо при трансляції з мов програмування високого рівня) параметри передаються через стек і в стеку розміщуються локальні змінні.

Параметри (які розміщує в стеку викликаючи програма), адреса повернення (яку розміщує команда `call`) і локальні змінні (які розміщує підпрограма) утворюють *стековий фрейм*. Як реперну точку, при зверненні до елементів стекового фрейму, можна використати адресу повернення. Якщо в стек занести три 4-байтні 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 може використовуватися у підпрограмі (для виклику інших підпрограм), то його значення зберігають в іншому регістрі, переважно в `ebp` (регістрі вказівника бази). Тоді регістр `ebp` буде містити адресу повернення, а регістр `esp` буде використовуватися як вказівник верхівки стеку.

В програмі можуть викликатися декілька підпрограм, які також використовують регістр `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 ebp і відновлює його перед поверненням управління.*

Для зберігання `ebp` також використовується стек, причому збереження здійснюється командою `push ebp` зразу після отримання управління підпрограмою. Таким чином, старе значення `ebp` розміщується в стеку після адреси повернення з підпрограми. Саме це старе значення `ebp` використовується як реперна точка для адресації стекового фрейму:

```
[ebp+16]
[ebp+12]    параметри
[ebp+8]
[ebp+4]     адреса повернення
[ebp]       збережене старе значення ebp
[ebp-4]
[ebp-8]     локальні змінні
```

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 обслуговування стеку. На початку підпрограми замість трьох команд може використати одну команда `enter 16,0`, а 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 функції в стек у зворотному порядку (справа наліво);

- викликає команду `call`.

Команда ***`call` виконує наступні дії:***

- записує у стек адресу наступної команди після `call` (адрес повернення);

- модифікує регістр команд `eip` так, щоб він містив адресу підпрограми виклику.

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 `ebp` командою `push ebp`;

- копіює вказівник верхівки стеку `esp` у вказівник бази `ebp` командою `mov ebp,esp`. Це дозволяє отримати доступ як до параметрів підпрограми, так і локальних змінних шляхом індексування вказівника бази. Вказівник бази `ebp` завжди вказуватиме на значення вказівника стеку `esp` на початку виконання підпрограми. Вказівник бази стеку є константою і дозволяє звертатися до всіх значень стекового фрейму (параметри, адрес повернення, `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+34]
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 `eax`;

- повертає стек в стан коли викликалася підпрограма (відновлює `esp`, `ebp`);

- повертає керування в точку виклику командою `ret`, яка знімає значення з верхівки стеку (адресу повернення) і записує його в регістр вказівник команд `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`.

3. Умовні переходи

3.1. Прості умовні переходи

Прості команди умовного переходу переходять 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 рівності `zf=0`.

Такі команди умовного переходу ставлять зразу після арифметичних операцій або команди `cmp`, наприклад

```
cmp eax, ebx    ; порівняти значення регістрів
jz are_equal    ; якщо вони однакові перейти на позначку are_equal
```

Таблиця 1. Найпростіші команди умовних переходів

Команда	Умова переходу	Команда	Умова переходу
jz	zf=1	jnz	zf=0
js	sf=1	jns	sf=0
jc	cf=1	jnc	cf=0
jo	of=1	jno	of=0
jp	pf=1	jnp	pf=0

3.2. Переходи за результатами порівнянь

За результатами команди порівняння `cmp` можуть встановлюватися два прапори, наприклад `sf=1, of=0` (переповнення не було, число отримане негативне) 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

Команда	Перехід якщо	Вираз	Умова переходу	Синонім
	нерівність для знакових чисел			
je	equal	a=b	zf=1	jz
jne	not equal	a≠b	zf=0	jnz
	нерівності для знакових чисел			
jl jnge	less not greater or equal	a<b	sf≠of	
jle jng	less or equal not greater	a≤b	sf≠of або zf=1	
jg jnle	greater not less or equal	a>b	sf=0 і zf=0	
jge jnl	greater or equal not less	a≥b	sf=of	
	нерівності для беззнакових чисел			
jb jnae	below not above or equal	a<b	cf=1	jc
jbe jna	below or equal not above	a≤b	cf=1 або zf=1	

```
    ja          above            a>b       cf=0 i zf=0
    jnbe        not below or
                equal
    jae         above or equal   a≥b       cf=0               jnc
    jnb         not below
```

Приклад використання команд:

```
.text
    mov eax,15
    cmp eax,15     ; порівняння
    jne not_equal  ; якщо операнди не рівні, перейти на позначку not_equal
        ; команди для eax = 15
        jmp out
 not_equal:
        ; команди для eax != 15
 out:
```

Крім команд переходів за результатами порівнянь `jcc`, існує родина команд `setcc`. Вони перевіряють стан прапорів так як `jcc`. За значенням прапорів операнд встановлюється в 1, якщо перевірювана умова `cc` істинна, і в 0, якщо умова фальшива. Команди `setcc` працюють тільки з операндами, які зберігаються в пам'яті і мають розмір один байт. Синтаксис команд `setcc`: `setcc операнд`.

Приклади команд `setcc`:

```
        cmp   a,5
        seta  al      ; a>5
        cmp   b,10
        setb  bl      ; a<5
        and   al,bl
        cmp   c,0
        sete  bl      ; a==5
        or    al,bl
        jz    is_false
is_true:
        ...
is_false:
```

3.3. Команда організації циклів

Команда `loop` призначена для організації циклів з наперед відомою кількістю ітерацій.

Синтаксис команди `loop`:

```
позначка:
..
loop  позначка
```

Принцип роботи:

- зменшити значення регістра `ecx` на 1;
- якщо `ecx` = 0, передати керування наступній за `loop` команді;
- якщо `ecx` ≠ 0, передати керування на *позначка*.

Як лічильник ітерацій команда `loop` використовує регістр `ecx`, в який перед початком циклу записується потрібне число ітерацій. Сама команда `loop` виконує дві дії: зменшує на одиницю значення в регістрі `eax` і, якщо результат не нульовий, переходить на задану позначку. Команда `loop` здійснює тільки короткі переходи на позначки, які розміщені від самої команди не далі як на 128 байт. Приклад підрахунку суми елементів масиву із 10 подвійних слів із проходженням масиву 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 і якщо ecx не 0, перехід на lp
```

Замість команди `loop` можна використати дві команди:

```
dec ecx
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 в регістрі `ECX` не нуль і прапор `zf` встановлений. Команда `loopne` (синонім `loopnz`) здійснює перехід, якщо в регістрі `ecx` не нуль і прапор `zf` скинутий.

Для аналізу стану регістра `ecx` є дві додаткові команди умовного короткого переходу:

`jcxz` – умовний перехід, якщо в регістрі `cx` нуль.

`jecxz` - умовний перехід, якщо в регістрі `ecx` нуль.

Команди не враховують прапори. Вони використовуються для запобігання виконання циклу у випадку, коли значення регістра `ecx` є нульовим на початку циклу. Якщо на момент входу у цикл `ecx=0`, то виконується тіло циклу, а потім від лічильника віднімається 1. В результаті лічильник отримає значення максимально можливого цілого числа 2^{32}. Для запобігання таких ситуацій перед циклом потрібно поставити команду `jecxz`:

```
; присвоєння значення ecx
jecxz lpq
lp: ; тіло циклу
; ...
loop lp
lpq:
```

На асемблері можна створювати довільні цикли з післяумовою, подібні до `do{} while();` в мові програмування С. Фрагмент коду такого циклу на асемблері:

```
loop_start:                /* початок циклу                    */

        /* тіло циклу */
         cmpl  ...          /* порівняння */
        je    loop_end     /* команда умовного переходу для виходу з
                              циклу                            */
        jmp   loop_start   /* інакше повторити цикл спочатку     */
loop_end:
```

Висновки.

Передача керування у програмі може здійснюватися безумовно або умовно. Безумовна передача керування здійснюється командою безумовного переходу `jmp`, викликом процедури і поверненням з процедури (`call, ret`), викликом і поверненням з програмних переривань.

Для організації виклику і повернення з процедури використовується стек. Якщо в процедуру передаються параметри і використовуються локальні змінні то то утворюється степовий фрейм. У стековий фрейм записуються параметри, адреса повернення (`esp`), старе значення вказівника бази (`ebp`) та локальні змінні.

Команди умовної передачі керування `jxx` аналізують результат виконання команди `cmp` за станом прапора `eflags` або вмістом регістра `ecx`.

Для організації циклів використовується команда `loop` з регістром лічильником `ecx`.

Запитання.

1. Розподіл команд переходів за принципом дії.
2. Команди безумовних переходів.
3. Команди виклику і повернення з підпрограм.
4. Організація стекового фрейму.
5. Умовні переходи.
6. Команда організації циклів.

Література.

1. Юров В.И. Assembler. Учебник для вузов. 2-е изд. – СПб.: Питер, 2003. – 637 с.
2. Аблязов Р.З. Программирование на ассемблера на платформе x86-64. – М.: ДМК Пресс, 2011. – 304 с.
3. Магда Ю.С. Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006. — 410 с.

6. 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 може здійснювати перетворення тексту програми виконуючи як макровиклики, так і безпосередні вказівки макродиректив.

План.

1. Основні поняття
2. Однорядкові макровизначення
2.1. Директиви `%define, %undef, %xdefine`
2.2. Макрозмінна `%assign (%iassign)`
2.3. Директива зчеплення виразів макровизначень `%+`
2.4. Спеціальні символи `%?, %??`
2.5. Директиви `%strcat, %strlen, %substr`
3. Багаторядкові макровизначення
3.1. Перевантаження макровизначень
3.2. Локальні позначки у макровизначеннях
3.3. Скупі макропараметри
3.4. Змінне число і діапазон параметрів
3.5. Лічильник параметрів макровизначення `%0`
3.6. Прокручування параметрів макровизначення
3.7. Відміна макровизначень
3.8. Коди умов як макропараметри
3.9. Зчеплення макровизначень для утворення таблиць кодів
4. Найпростіші приклади макровизначень
5. Препроцесорний стек контекстів
6. Директиви асемблювання з умовами
6.1. Директиви з логічними умовами
7. Стандартні макровизначення
8. Директиви асемблера

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 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 макропроцессора

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

Макросом називають правило перетворення фрагмента програми з іменем. Ім'я фрагмента програми називають **іменем макроса** (хоча часто замість терміну "ім'я макроса" використовують просто слово "макрос").

Перед використанням макроса його потрібно визначити, тобто вказати макропроцесору, що деякий ідентифікатор є іменем макроса, а також задати правило, згідно якого буде діяти макропроцесор, коли зустріне це ім'я. Фрагмент програми, який визначає макрос називається **макровизначенням**.

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

Приклад макровизначення для виклику підпрограми з одним аргументом

```
%macro pcall 2  ; к-ть параметрів макроса
push %2
call %1
add esp 4       ; звільнення стеку від аргумента
%endmacro
```

Макрос з іменем `pcall` 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 макроса. Якщо в тексті асемблерної програми зустрінеться рядок `pcall myproc, acx` то макропроцесор сприйме його як макровиклик і виконає макророзширення згідно макровизначення. В результаті макрос буде замінено на наступний фрагмент

```
push eax
call myproc
add esp 4
```

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

```
%include "stud.inc"
```

2. Однорядкові макроси

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

Для описання однорядкового макросу використовується директива `%define`.

Директива **`%define`** замінює значення змінних або розширює вирази:

```
%define size 100
%define ctrl 0x1F &
%define param(a,b) ((a)+(a)*(b))
…
mov byte [param(2,ebx)], ctrl 'D'
```

буде розширено до

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

Коли однорядковий макрос містить інший макрос, то макророзширення відбувається під час *виконання* (*підставлення*), а не визначення:

```
%define a(x) 1+b(x)
%define b(x) 2*x
mov ax,a(8)         ; макророзширення під час виконання 1+2*8
```

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 циклічних розширень у макросах використовується механізм розширення тільки для першого появлення макросу:

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

Макроси можна перевантажувати:

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

Макроси без параметрів можна перевизначати в одному файлі:

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

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

```
%define foo bar
%undef foo
mov eax, foo
```

Для того, щоб макроси розширювалися під час *визначення* (*негайно*), а не під час виконання, використовується директива **`%xdefine`**.

```
%define isTrue 1
%define isFalse isTrue
%define isTrue 0
  val1: db isFalse
%define isTrue 1
  val2: db isFalse
```

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 –` матиме 1 .

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

```
%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 і` і `val2` матимуть значення 1.

2.2. Директива `%assign (%iassign)`

`%assign` (`%iassign` – регістро нечутлива версія) використовується для визначення макрозмінних, які не мають параметрів, а мають числове значення. Значення можна задати як вираз, який оцінюється один раз при обробці макрозмінної:

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

З макрозмінною `%assign` часто використовуються директива макроповторень `%rep … %endrep`. В тілі макроповторення може 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
```

2.3. Директива зчеплення виразів макровизначень `%+`

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          ; структура
.COM1addr RESW 1
.COM2addr RESW 1
; ..
endstruc
```

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
```

Доступ до різних елементів структури з використанням макросу

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

2.4. Спеціальні символи `%?`, `%??`

Спеціальні символи `%?`, `%??` можуть використовуватися для посилання на свої імена макросів всередині макророзширень. Директива `%?` посилається на ім’я макросу при виклику, а `%??` – при оголошенні макросу, наприклад:

```
%idefine Foo mov %?,%??
foo
FOO
```

буде розширена до:

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

2.5. Директиви `%strcat`, `%strlen`, `%substr`

Директива `%strcat` об'єднує символьні рядки і назначає їх однорядковому макросу:

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

Директива `%strlen` назначає довжину стрічки макросу

```
%strlen charcnt ’my string’
```

В результаті `charcnt` буде присвоєно значення 9.
Директива `%substr` видобуває рядки з підрядків:

```
%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

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 ebp
    mov ebp,esp
    sub esp,%1
%endmacro
```

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

Виклик макросу:

```
myfunc: prolog 12
```

буде розширений до наступних інструкцій:

```
myfunc: push ebp
mov ebp,esp
sub esp,12
```

Приклад макросу з двома параметрами:

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

який буде розширений до наступних інструкцій (якщо параметр задається списком з комою, то він 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 eax, 4    ; номер системного виклику (sys_write)
mov ebx, 1
mov ecx, %1
mov edx, %2
int 80h
%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 eax,1   ; номер системного виклику(sys_exit)
int 0x80    ; звернення до ядра
```

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 макровизначеннях" – позначки 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 ebp
   mov ebp,esp
%endmacro
```

і макрос з двома параметрами:

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

Тоді виклики:

```
push ebx     ; це не виклик макросу
push eax,ecx ; 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 dx,%%str
   mov cx,%%endstr-%%str
   mov bx,%1
   mov ah,0x40
   int 0x21
%endmacro
```

При виклику, макросу з параметрами `writefile [filehandle],"hello, world",13,10` приймає як перший параметр `[filehandle]`, а другий доповнює рештою параметрів – `"hello, world",13,10` (розміщується після `db`).

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 ax,0x4c01
   int 0x21
%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 0-*
```

Для розширення параметрів у заданому діапазоні використовується спеціальна конструкція %{start:end}, наприклад виклик:

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

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

розшириться до діапазону 3,4,5, а виклик

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

розшириться до діапазону 5,4,3.

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 по колу справа наліво на 1 позицію (%rotate -1 – зліва на право).

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

3.7. Відміна макросів

Для відміни багаторядкових макросів використовується директива %unmacro:

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

3.8. Коди умов як макропараметри

Для використання кодів умов як макропараметрів використовується синтаксис %+1, %-1. Синтаксис %+1 вказує, що він містить код умови і буде впливати на препроцесор так, що він буде видавати повідомлення про помилки, якщо макрос викликається з параметром, який не є дійсним кодом умови (%-1 – параметр, який є інверсним кодом умови).

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

Цей макрос може бути викликаний як `retc ne` і в результаті спричинить інструкцію умовного переходу `je` у макророзширенні. Вираз `j%+1` спричинить інструкцію умовного переходу `jne` у макророзширенні.

3.9. Зчеплення макросів для утворення таблиць кодів

Макроси можна зчеплювати і створювати, наприклад, таблиці кодів:

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

Макровиклик

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

розшириться до:

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

4. Найпростіші приклади макросів

Використовуючи макроси можна значно скоротити виклики підпрограм. Розглянемо виклики підпрограм з двома і трьома параметрами. Макровизначення 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
    call %1
    add esp,8
%endmacro
```

```
%macro call_3 4
    push %4
    push %3
    push %2
    call %1
    add esp,12
%endmacro
```

Макровиклики матимуть вид:

```
call_2 myproc eax, 1, 2        call_3 myproc eax, 1, 2, 3
```

Саме макророзширення буде наступним: макропроцесор замінить входження `%1, %2, %3` на відповідні параметри. В результаті макророзширення буде отримано наступний текст:

push eax push 1 push 2 call myproc add esp 8	push eax call myproc add esp 4

```
push eax
call myproc
add esp 4
```

Аналогічним чином можна записати макровизначення для виклику підпрограм з двома і трьома параметрами:

%macro call_2 3
    push %3
    push %2
    call %1
    add esp,8
%endmacro
	%macro call_2 4
	    push %4
	    push %3
	    push %2
	    call %1
	    add esp,8
	%endmacro

5. Препроцесорний стек контекстів

Асемблер `Nasm` підтримує препроцесорний стек контекстів, кожен з яких задається своїм іменем. 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 макровизначенні 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 потрібно визначення (або доступ до) локальних позначок контексту нижче від першої верхівки стеку можна використати `%$$label`, або `%$$$label` і так далі.

6. 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.

6.1. Директиви з логічними умовами

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* макровизначення задається директивою `%ifdef`:

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

Якщо визначення `DEBUG` закоментувати:

```
; %define DEBUG
```

то макровизначення макропроцесором ігнорується і не виконується, тому його можна залишити у тексті програми. Розкоментовуючи або закоментуючи ім'я макророзширення можна керувати відлагоджувальним 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 визнечені 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 макророзширення можна перевіряти і факт його відсутності директивами `%ifndef, elifndef`.

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
```

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

Для тестування довільних числових виразів використовується директиви `%if, %elif, %ifn, %elifn вираз`. `%if` 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 нечутлива).

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

Для *визначення чи у макрос, як перший параметр, передається ідентифікатор, число або стрічка* 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 bx,%1
                mov ah,0x40
                int 0x21
%endmacro
writefile [file], strpointer, length
writefile [file], "hello", 13, 10
```

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

`%ifempy` – немає параметрів

`%iftoken 1` – один параметр

`%iftoken -1` – більше одного параметра

7. Стандартні макровизначення

Макровизначення **`struc`** і **`endstruc`** використовуються для визначення даних типу структура. Макровизначення `struc` може мати один або два параметри. Перший параметр задає

ім'я типу даних, а другий – зміщення структури від бази. Всередині структури необхідно визначити поля з використанням псевдоінструкцій `resb`.

```
struc mytype
   mt_long: resd 1
   mt_word: resw 1
   mt_byte: resb 1
   mt_str:  resb 32
endstruc
```

У структурі визначено наступні символи, які мають зміщення: `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 елементу 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
```

Маючи визначений тип структури можна оголосити екземпляр структури у сегменті даних використовуючи механізм `istruc`:

```
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.

Макровизначення `align` або `alignb` вирівнює 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-байтну границю
align 16       ; вирівняти на 16-байтну границю
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
```

8. Директиви асемблера

Дирекиви асемблера – це макровизначення, визначені самим асемблером.

Директива **BITS N** – задає режим роботи процесора (16- або 32-розрядний).

Директива **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, під час компіляції їм відводиться тільки пам'ять, а значення не присвоюються.

Директива **EXTERN** – визначає зовнішні ідентифікатори для "імпорту" у програму.

Директива **GLOBAL** – визначає ідентифікатори для "експорту", вони позначаються як глобальні і можуть використовуватися іншими модулями (програмами).

Директива **COMMON** – використовується замість **GLOBAL** для "експорту" ідентифікаторів, оголошених у секції **.bss**.

Директива **CPU** – заставляє процесор генерувати команди тільки для вказаного типу процесора.

Директива **ORG** – встановлює початкову адресу для завантаження програми.

Висновки.

Асемблери мають недостатні можливості введення і виведення даних (порівняно з мовами високого рівня), тому їх доповнюють потужними макропроцесорами. Макропроцесор – це програмний засіб, який отримує на вхід деякий текст і, використовуючи вказівки, що містяться у ньому, частково перетворює його, формуючи на виході текст, але вже без вказівок на перетворення. Результатом роботи макропроцесора є текст на асемблері, який вже асемблюється згідно з правилами мови. Правило, у відповідності з яким фрагмент програми, що містить визначено слово, повинен бути перетворений, називається макросом. Фрагмент програми, який визначає макрос називається макровизначенням.

Макровизначення бувають одно- і багаторядкові. Для написання макровизначень використовуються директиви.

Запитання.

1. Що таке макропроцесор, макрос, ім'я макроса і макророзширення.
2. Директиви однорядкових макровизначень **%define, %xdefine і** макрозмінна **%assign.**
3. Багаторядкове макровизначення %macro … %endmacro.
4. Директиви асемблювання з умовами.
5. Макровизначення **struc** і **endstruc** для описання структур.
6. Призначення директив асемблера.

Література.

1. Юров В.И. Assembler. Учебник для вузов. 2-е изд. – СПб.: Питер, 2003. – 637 с.

2. Аблязов Р.З. Программирование на ассемблера на платформе x86-64. – М.: ДМК Пресс, 2011. – 304 с.

3. Магда Ю.С. Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006. — 410 с.

4. Столяров А.В. Программирование на языке асемблера NASM для ОС UNIX. Уч. пособие. – 2-е изд. – М.: Макс-пресс, 2011. – 188 с.

7. ПРОЦЕСИ, СИСТЕМНІ ВИКЛИКИ І ПІДПРОГРАМИ

Мета. Вивчення процесів захищеного режиму, системних викликів, 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 `int`) з номером `0x80`. 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 реєнтерабельними, що дозволяє їх використовувати у багатозадачному режимі. Рекурсивні підпрограми дозволяють самовиклик.

План.
1. Процеси у захищеному режимі
2. Системні виклики
3. Підпрограми
3.1. Передача параметрів у підпрограму
3.2. Локальні змінні у стеку
4. Виклик підпрограм асемблера з Сі програми
5. Домовленості, щодо виклику підпрограм асемблера з Сі програми
6. Реєнтерабельні 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 (і 32-розрядним регістрам) любий процес може адресувати 4 ГБайт адресного простору. Процес розміщує у цьому адресному просторі чотири секції: коду (`.txt`), статичних даних (`.data`), динамічних даних (`.bss`, куча) 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 Мб). 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 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:

ESP після запуску програми	Вільна пам'ять
argc	Кількість параметрів
argv[0]	Вказівник на імя програми
argv[1]	Вказівник на аргументи програми
argv[argc-1]	
NULL	Кінець аргументів командного рядка
env[0]	Вказівники на змінні оточення
env[1]	
env[n]	
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",15,0
segment .text
global main
main:
  mov ecx,[esp+4]   ; argc
  mov edx,[esp+8]   ; argc
top:
  push ecx    ; збереження регістрів, які використовує printf
  push edx
  push dword [edx]
  push dword fmt
  call printf
  add esp,8   ;  вилучення 2-х параметрів

  pop edx     ; відновлення регістрів для printf
  pop ecx
  add edx,4   ; вказівник на наступний аргумент
  dec ecx     ; лічильник аргументів
  jnz top
  ret
```

2. Системні виклики

Системний виклик – це звернення задачі користувача за послугами до ядра операційної системи. В Linux звернутися до ядра операційної системи можна через програмне переривання (команда int) з номером 0x80. Завдяки захищеному режиму, при виклику переривання 0x80 змінюється контекст програми (значення сегментного регістра) і процесор починає виконувати код ядра. Після завершення оброблення переривання буде продовжено виконання програми.

Аргументи окремих системних викликів передаються через регістри процесора, що забезпечує найбільшу швидкодію. Номер системного виклику поміщається у регістр eax. Для передачі аргументів використовується набір регістрів у фіксованому порядку: ebx, ecx, edx,

esi, edi, ebp. Значення, яке повертає системний виклик, заноситься у регістр eax. У випадку помилки повертається негативне значення.

Список системних викликів бібліотеки libc мови Сі можна отримати наступною командою:

```
>ls /usr/share/man/man2 | sed –e s/.2.gz//g | xargs man –s 2 –k |
     sort | grep –v 'unimplemented system calls'
```

Кожний системний виклик має свій номер. Всі вони перераховані в файлі /usr/include/asm-86/unistd_32.h (32-розрядна ОС) або /usr/include/asm-86/unistd_64.h (64-розрядна ОС).

restart_ syscall 0	setregid 71	newselect 142	setuid32 213	keyctl 288
sys_exit 1	sigsuspend 72	flock 143	setgid32 214	ioprio_set 289
fork 2	sigpending 73	msync 144	setfsuid32 215	ioprio_get 290
read 3	sethostname 74	readv 145	setfsgid32 216	inotify_init 291
write 4	setrlimit 75	writev 146	pivot_root 217	inotify_add_watch 292
open 5	getrlimit 76	getsid 147	mincore 218	inotify_rm_watch 293
close 6	getrusage 77	fdatasync 148	madvise 219	migrate_pages 294
waitpid 7	gettimeofday 78	sysctl 149	getdents64 220	openat 295
creat 8	settimeofday 79	mlock 150	fcntl64 221	mkdirat 296
link 9	getgroups 80	munlock 151	gettid 224	mknodat 297
unlink 10	setgroups 81	mlockall 152	readahead 225	fchownat 298
execve 11	select 82	munlockall 153	setxattr 226	futimesat 299
chdir 12	symlink 83	sched_setparam 154	lsetxattr 227	fstatat64 300
time 13	oldlstat 84	sched_getparam 155	fsetxattr 228	unlinkat 301
mknod 14	readlink 85	sched_setscheduler 156	getxattr 229	renameat 302
chmod 15	uselib 86	sched_getscheduler 157	lgetxattr 230	linkat 303
lchown 16	swapon 87	sched_yield 158	fgetxattr 231	symlinkat 304
break 17	reboot 88	sched_get_priority_ max 159	listxattr 232	readlinkat 305
oldstat 18	readdir 89	sched_get_priority_ min 160	llistxattr 233	fchmodat 306
lseek 19	mmap 90	sched_rr_get_interv al 161	flistxattr 234	faccessat 307
getpid 20	munmap 91	nanosleep 162	removexattr 235	pselect6 308
mount 21	truncate 92	mremap 163	lremovexattr 236	ppoll 309
umount 22	ftruncate 93	setresuid 164	fremovexattr 237	unshare 310
setuid 23	fchmod 94	getresuid 165	tkill 238	set_robust_list 311
getuid 24	fchown 95	vm86 166	sendfile64 239	get_robust_list 312
stime 25	getpriority 96	query_module 167	futex 240	splice 313
ptrace 26	setpriority 97	poll 168	sched_setaffinity 241	sync_file_range 314
alarm 27	NR_profil 98	nfsservctl 169	sched_getaffinity 242	tee 315
oldfstat 28	NR_statfs 99	setresgid 170	set_thread_area 243	vmsplice 316
pause 29	fstatfs 100	getresgid 171	get_thread_area 244	move_pages 317
utime 30	NR_ioperm 101	prctl 172	io_setup 245	getcpu 318
stty 31	NR_socketcall 102	rt_sigreturn 173	io_destroy 246	epoll_pwait 319
gtty 32	NR_syslog 103	rt_sigaction 174	io_getevents 247	utimensat 320

access 33	NR_setitimer 104	rt_sigprocmask 175	io_submit 248	signalfd 321
nice 34	getitimer 105	rt_sigpending 176	io_cancel 249	timerfd_create 322
ftime 35	stat 106	rt_sigtimedwait 177	fadvise64 250	eventfd 323
sync 36	lstat 107	rt_sigqueueinfo 178	exit_group 252	fallocate 324
kill 37	fstat 108	rt_sigsuspend 179	lookup_dcookie 253	timerfd_settime 325
rename 38	olduname 109	pread64 180	epoll_create 254	timerfd_gettime 326
mkdir 39	iopl 110	pwrite64 181	epoll_ctl 255	signalfd4 327
rmdir 40	vhangup 111	chown 182	epoll_wait 256	eventfd2 328
dup 41	idle 112	getcwd 183	remap_file_pages 257	epoll_create1 329
pipe 42	vm86old 113	capget 184	set_tid_address 258	dup3 330
times 43	wait4 114	capset 185	timer_create 259	pipe2 331
prof 44	swapoff 115	sigaltstack 186	timer_settime 260	inotify_init1 332
brk 45	sysinfo 116	sendfile 187	timer_gettime 261	preadv 333
setgid 46	ipc 117	getpmsg 188	timer_getoverrun 262	pwritev 334
getgid 47	fsync 118	putpmsg 189	timer_delete 263	rt_tgsigqueueinfo 335
signal 48	sigreturn 119	vfork 190	clock_settime 264	perf_event_open 336
geteuid 49	clone 120	ugetrlimit 191	clock_gettime 265	recvmmsg 337
getegid 50	setdomainname 121	mmap2 192	clock_getres 266	fanotify_init 338
acct 51	uname 122	truncate64 193	clock_nanosleep 267	fanotify_mark 339
umount2 52	modify_ldt 123	ftruncate64 194	statfs64 268	prlimit64 340
lock 53	adjtimex 124	stat64 195	fstatfs64 269	name_to_handle_at 341
ioctl 54	mprotect 125	lstat64 196	tgkill 270	open_by_handle_at 342
fcntl 55	sigprocmask 126	fstat64 197	utimes 271	clock_adjtime 343
mpx 56	create_module 127	lchown32 198	fadvise64_64 272	syncfs 344
setpgid 57	init_module 128	getuid32 199	vserver 273	sendmmsg 345
ulimit 58	delete_module 129	getgid32 200	mbind 274	setns 346
oldolduname 59	get_kernel_syms 130	geteuid32 201	get_mempolicy 275	process_vm_readv 347
umask 60	quotactl 131	getegid32 202	set_mempolicy 276	process_vm_writev 348
chroot 61	getpgid 132	setreuid32 203	mq_open 277	kcmp 349
ustat 62	fchdir 133	setregid32 204	mq_unlink 278	finit_module 350
dup2 63	bdflush 134	getgroups32 205	mq_timedsend 279	
getppid 64	sysfs 135	setgroups32 206	mq_timedreceive 280	
getpgrp 65	NR_personality 136	fchown32 207	mq_notify 281	
setsid 66	NR_afs_syscall 137	setresuid32 208	mq_getsetattr 282	
sigaction 67	NR_setfsuid 138	getresuid32 209	kexec_load 283	
sgetmask 68	setfsgid 139	setresgid32 210	waitid 284	
ssetmask 69	llseek 140	getresgid32 211	add_key 286	
setreuid 70	getdents 141	chown32 212	request_key 287	

Приклади оголошення системних викликів у програмі:

```
LINUX_SYSCALL equ 0x80  #номер виклику звернення до ядра ОС
SYS_EXIT  equ 1         #номер виклику завершення програми
SYS_READ  equ 3         #номер виклику читання вхідного потоку
SYS_WRITE equ 4         #номер виклику записування у вихідний потік
SYS_OPEN  equ 5         #номер виклику відкриття файлу
```

```
SYS_CLOSE equ 6          #номер виклику закриття файлу
SYS_BRK   equ 45         #номер виклику зміни розміру сегмента даних
```

Стандартні номери (дескриптори) потоків введення/виведення:

```
STDIN  equ  0
STDOUT equ  1
STDERR equ  2
```

Приклад системного виклику, який завершує виконання програми:

```
mov eax,1 ;номер системного виклику — sys_exit
mov ebx,0 ;код повернення 0
int 0x80  ;виклик ядра і завершення поточної програми
```

Приклади системних викликів для роботи з файлами.

Системний виклик 8 – створення і відкриття нового файлу (`creat`).

Вхід: `eax = 8`
`ebx = ім’я_файлу`
`ecx = права_доступу_до_файлу`

Повернення: `eax = файловий дескриптор`

Помилки: `eax = код_помилки`

Системний виклик 5 – відкриття існуючого файлу (`open`).

Вхід: `eax = 5`
`ebx = ім’я_файлу`
`ecx = режим_використання_файлу`
`edx = права_доступу_до_файлу`

Повернення: `eax = файловий дескриптор`

Помилки: `eax = код_помилки`

Системний виклик 3 – читання з файлу (`read`).

Вхід: `eax = 3`
`ebx = файловий дескриптор`
`ecx = вказівник_на_вхідний_буфер`
`edx = розмір_буферу`

Повернення: `eax = число_прочитаних_файлів`

Помилки: `eax = код_помилки`

Системний виклик 4 – записування у файл (`write`).

Вхід: `eax = 4`
`ebx = файловий дескриптор`
`ecx = вказівник_на_вихідний_буфер`
`edx = розмір_буферу`

Повернення: `eax = число_записаних_файлів`

Помилки: `eax = код_помилки`

Системний виклик 6 – закриття файлу (`close`).

Вхід: `eax = 6`
`ebx = файловий дескриптор`

Повернення: `eax = -`

Помилки: `eax = код_помилки`

Системний виклик 19 – зміна поточної позиції у файлі (`lsek`).

Вхід: `eax = 19`
`ebx = файловий дескриптор`
`ecx = зміщення`
`edx = підрахунок зміщення (від початку_файла, від_поточної_позиції, від_кінця_файлу)`

Повернення: `eax = зміщення від початку файла`

Помилки: `eax = код_помилки`

Таким чином, використовуючи всі регістри загального призначення, можна передати до 6 параметрів. Такий спосіб виклику (з передачею параметрів через регістри) називається швидким `fastcall`. В інших системах (наприклад, BSD) можуть застосовуватися інші способи виклику.

Більш докладну інформації для кожного системного виклику можна отримати з довідкової системи Linux:

```
>man <ім'я_виклику>
```

Необхідно зауважити, що не потрібно використовувати системні виклики скрізь, де тільки можна, без особливої необхідності. В різних версіях ядра порядок аргументів у деяких системних викликах відрізняється, і це спричиняє помилки, які досить важко знайти. Тому варто використовувати функції стандартної бібліотеки Сі, так як їх сигнатури не змінюються, що забезпечує переносимість коду на Сі. Якщо ж пишеться невелика частина самого навантаженого коду і є недопустимими накладні витрати, які вносить виклик стандартної бібліотеки Сі, тоді використовуються системні виклики.

Розглянемо для прикладу системний виклик `write`, який дозволяє виводити дані через один з відкритих потоків введення-виведення (на екран або у файл). Системний виклик має номер 4 і приймає три параметри: номер (дескриптор) потоку виведення, адресу пам'яті даних, кількість цих даних в байтах.

Тоді програма, яка друкує повідомлення на екран і завершає роботу з поверненням коду у середовище Linux буде наступною:

```
SYS_EXIT  equ 1
SYS_WRITE equ 4
STDOUT    equ 1
section .data
msg db "Привіт світ", 0xA  ;  0xA -> '\0'
msg_len   equ $-msg
section .text
global _start       ; експорт точки входу в програму
_start:
mov eax,SYS_WRITE ; номер виклику write у eax
mov ebx,STDOUT    ; номер STDOUT у ebx
mov ecx,msg       ; адреса повідомлення в ecx
mov edx,msg_len   ; довжина повідомлення із символом 0xA у edx
int 80h           ; звернення до ядра і виконання системного виклику
mov eax,SYS_EXIT  ; номер SYS_EXIT в eax
mov ebx,0         ; код повернення в ebx
int 80h           ; звернення до ядра для завершення програми
```

Для введення даних (як з клавіатури, так із файлів) використовується виклик `read`, який має номер 3. Він має три параметри: номер (дескриптора) потоку введення, адресу пам'яті, де будуть розміщені вхідні дані, кількість цих даних в байтах. Після виклику `read` потрібно *проаналізувати* результат його роботи. Якщо читання пройшло успішно – виклик поверне кількість прочитаних символів. Виклик повертає 0 у ситуації "кінець файлу". Така ситуація виникає коли досягнуто кінець файлу і в ньому більше не має даних або натиснуто комбінацію клавіш `CTRL+D`. Від'ємне значення вказує на помилку при читанні. Приклад програми, яка читає дані з консолі і виводить їх на екран:

```
SYS_EXIT  equ 1
SYE_READ  equ 3
SYS_WRITE equ 4
STDIN     equ 0
STDOUT    equ 1
section .bss
msg resb 64
msg_len   equ $-msg
section .text
global _start      ; експорт точки входу в програму
```

```
_start:
nop                 ; порожня команда для зневадника
mov eax,SYS_READ    ; номер виклику read
mov ebx,STDIN       ; читання з консолі
mov ecx,msg         ; адреса буфера, коди поміщаються дані
mov edx, msg_len    ; довжина даних
int 0x80            ; звернення до ядра
mov edx,eax         ; результат системного виклику в eax
jl label            ; аналіз результату
mov eax, SYS_WRITE  ; системний виклик write
mov ebx, STDOUT     ; виведення на екран
mov ecx, msg        ; адреса даних
mov edx, msg_len    ; довжина даних
int 80h             ; звернення до ядра і виконання системного виклику
label:
mov eax,SYS_EXIT    ; системний виклик _exit
mov ebx,edx         ; код результату читання або msg_len
int 0x80            ; звернення до ядра
```

Для роботи з файлами потрібно створити додаткові потоки введення/виведення на додаток до стандартних потоків. Додаткові потоки створюються системним викликом `open` з номером 5. Виклик має три параметри: *адреса рядка тексту*, який містить ім'я файлу (ім'я має закінчуватися нульовим байтом, як відіграє роль обмежувача); число, яке задає *режим використання файлу* (читання, запис), значення якого задається бітовою стрічкою; *права доступу до файлу* (задається тільки при створенні файлу, переважно як `0666q`).

Деякі прапори режиму використання файлу:

O_RDONLY	Тільки читання	000h
O_WRONLY	Тільки запис	001h
O_RDWR	Читання і запис	002h
O_CREAT	Дозволити створення файлу	040h
O_EXCL	Вимагати створення файлу	080h
O_TRUNC	Якщо файл існує, знищити його вміст	200h
O_APPEND	Якщо файл існує добавити у кінець	400h

Наприклад комбінації прапорів `O_WRONLY|O_CREAT|O_TRUNC` мови Сі у Linux задається значенням `241h`.

Результат виклику `open` повертається в регістрі `eax`. Якщо виклик закінчився успішно, то `eax` містить дескриптор відкритого файлу (номер потоку введення або виведення). Саме цей дескриптор потрібно використовувати як перший параметр в командах `read` і `write` для роботи з файлами. Звичайно це значення копіюється у спеціально відведену область пам'яті. Якщо виклик закінчився не успішно, то `eax` містить негативне значення.

Коли робота з файлом закінчена його потрібно закрити системним викликом `close`, який має номер 6. Виклик має один параметр, який дорівнює дескриптору файлу, який потрібно закрити.

Приклад:

```
SYS_open  equ 5
SYS_close equ 6
SYS_w     equ 0101h   ; прапор запису
SYS_mod   equ 0666q   ; права доступу 8-ме число
SYS_write equ 4
SYS_exit  equ 1
```

```
segment .bss          ; сегмент неініціалізованих даних
   fd   resb  1
segment .data         ; сегмент ініціалізованих даних
   file db "7.dat\0"
   msg  db "Привіт світ",10
   len equ $-msg
segment .text
global _start         ; експорт точки входу в програму
_start:
   nop
   mov eax,SYS_open
   mov ebx,file
   mov ecx,SYS_w
   mov edx,SYS_mod
   int 0x80          ; відкриття файлу із заданим прапором і правами
   mov [fd],eax      ; запис файлового дескриптора
   jl label          ; аналіз результату

   mov eax,SYS_write
   mov ebx,[fd]
   mov ecx,msg
   mov edx,len
   int 0x80

   mov eax,SYS_close
   mov ebx,[fd]
   int 0x80        ; запис у файл повідомлення

label:
   mov eax,SYS_exit
   mov ebx,[fd]
   int 0x80          ; завершення програми з поверненням дескриптора файлу
ret
```

Для роботи із системними викликами з різним число параметрів можна використати макрос `my_syscall`, записавши його у файл `my_syscall.inc`:

```
%macro my_syscall 1-*
%rep %0
%rotate -1
  push dword %1
%endrep
  pop eax
%if %0 > 1
  pop ebx
  %if %0 > 2
    pop ecx
    %if %0 > 3
      pop edx
      %if %0 > 4
 pop esi
 %if %0 > 5
   pop edi
   %if %0 > 6
     %error "Забагато параметрів для Linux системних викликів"
   %endif
 %endif
      %endif
    %endif
  %endif
%endif
  int 80h
%endmacro
```

Використання макросу my_syscall значно скорочує текст програми виведення стрічки з пам'яті на екран:

```
%include "my_syscall.inc"
section .data
msg db "Hello world", 10
msg_len equ $-msg
section .text
global _start
_start:
   my_syscall 4, 1, msg, msg_len
   my_syscall 1, 0
```

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. Але це можна зробити при неявній формі виклику інструкції jmp і прийнятті домовленостей, що адреса позначки повернення заноситься у регістр ebx:

```
     mov ebx, return1   ;  ebx = &return1
     jmp subprog1
return1:
      ...
subprog1:
      ...
     jmp ebx           ;  повернення за адресою позначки return1
```

Для неявної адресації можуть бути використані регістри eax, ebx, ecx, edx, esi, edi. 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 подвійне слово у стек віднімаючи 4 від вмісту регістра ESP (вказівник верхівки стеку) і записуючи його значення як [ESP] . Інструкція pop читає подвійне слово як [ESP] і потім додає 4 до вмісту регістра ESP.

Приклад роботи команд push, pop в припущенні, що початкове значення ESP=1000h.

```
push dword 1 ; 1 записано за адресою ESP = 0FFCh
push dword 2 ; 2 записано за адресою ESP = 0FF8h
push dword 3 ; 3 записано за адресою ESP = 0FF4h
pop eax ; EAX = 3, ESP = 0FF8h
pop ebx ; EBX = 2, ESP = 0FFCh
pop ecx ; ECX = 1, ESP = 1000h
```

Команди роботи зі стеком pusha, popa використовуються для записування і зчитування регістрів EAX, EBX, ECX, EDX, ESI, EDI і EBP.

Для спрощення виклику підпрограм є дві інструкції call і ret. Інструкція call робить безумовний перехід на підпрограму і записує у стек адресу наступної інструкції після call. Інструкція ret зчитує зі стеку записану адресу повернення і здійснює на неї перехід. Виклик підпрограми у такому випадку буде наступний:

```
     call subprog1   ; запис у стек адреси return1
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 мінятися у підпрограмі, то передаються їхні *адреси*, а не *значення*. Якщо розмір параметра менший від подвійного слова, то він перетворюється у подвійне 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 4).

Регістри	Стек
esp+4	Параметр
esp	Адреса повернення

а)

Регістри	Стек	Записує в стек
esp+8	Параметр	← Осн. програма
esp+4	Адреса повернення	← Осн. програма
esp	Локальна змінна 1	← Підпрограма

б)

Рисунок 1 – Стан стеку при виклику підпрограми

У першому випадку значення параметра буде як [`esp+4`], а у другому – як [`esp+8`]. При використанні локальних змінних міняється адреса параметрів відносно регістра `esp`, що є незручним і може бути джерелом помилок. Для розв'язання цієї проблеми використовується регістр `ebp` для посилання на дані у стеку. За домовленостями викликів мови Сі прийнято, що *підпрограма* спочатку записує оригінальне значення `ebp` у стек, а потім встановлює нове значення `ebp` рівним значенню `esp`. Це дозволяє при поповненні стеку міняти значення `esp`, без зміни `ebp`, рис. 2.

Регістри		Стек	Записує в стек
esp+12	ebp+8	Параметр	← Осн. програма
esp+8	ebp+4	Адреса повернення	← Осн. програма
esp+4	ebp	Оригінальне ebp=esp	← Підпрограма
esp	ebp-4	Локальна змінна 1	← Підпрограма

Рисунок 2 – Стан стеку при виклику із регістром `ebp`

При завершенні підпрограма має відновити початкове значення `ebp`. Приклад коду підпрограми, яка зберігає і відновлює оригінальне значення `ebp`:

```
1  subprogram:
2  push ebp      ; збереження оригінального EBP у стеку
3  mov ebp,esp   ; нове значення EBP = ESP
4  ; subprogram code
5  pop ebp       ; відновлення оригінального значення EBP
6  ret
```

Рядки коду 2, 3 і 5, 6 є однаковими для всіх підпрограм. Рядки 2, 3 називаються епілогом, а 5, 6 – прологом підпрограми. Тепер параметри є доступними у підпрограмі як [ebp+8], 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. Ключове слово `pascal` у шаблонах і визначеннях функцій вказує, що компілятор використовує домовленості мови Pascal. Ключове слово `stdcall`, яке використовують функції Microsoft Windows API Сі, також вказує на домовленості виклику мови Pascal.

Приклад виклику підпрограми згідно домовленостей мови Сі. Параметри заноситься у стек інструкцією `push`, а вилучається інструкцією `add`. Можна було б використати інструкцію `pop ecx`, а але вона записує у регістр `ecx` непотрібне значення.

```
1 push dword 1 ; записування в стек 1 як параметра
2 call fun
3 add esp, 4   ; 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 *реєнтерабельною*. Реєнтерабельну 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 `ebp`. Місце під них виділяє пыдпрограма відніманням числа потрібних для них байтів від `esp`.

```
1 subprogram:
2 push ebp     ; збереження оригінального EBP у стеку
3 mov ebp,esp  ; нове значення EBP = ESP
4 sub esp,LOCAL_BYTES  ; число байт для локальних змінних
5 ; subprogram code
6 mov esp,ebp  ; звільненні стеку від локальних змінних
7 pop ebp      ; відновлення оригінального значення EBP
8 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. Вигляд підпрограми з використанням інструкцій `enter` і `leave`.

```
1 subprogram:
2 enter LOCAL_BYTES,0   ; число байт для локальних змінних
3 ; subprogram code
4 leave
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, наприклад підключенням файлу `asm_io.asm` (бібліотеки Сі 32-розрядні і асемблерний код використовує 32-розрядні інструкції).

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 driver.c
 2 | компіляція в об'єктний код:
 3 |   gcc -c driver.c
 4 | */
 5 | int main()
 6 | {
 7 |    int ret status ;
 8 |    ret status = asm_main();
 9 |    return ret status ;
10 | }
```

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
 2 | ; асемблювання і компонування:
 3 | ; nasm -f elf32 1.asm
 4 | ; gcc -o driver driver.o 1.o asm_io.o
 5 | ;
 6 | ; компіляція і компонування:
 7 | ; gcc -o driver driver.c 1.o asm_io.o
 8 |
 9 | %include "asm_io.inc"
10 | ; сегменти даних
11 | segment .data
12 | prompt1 db "Введіть число: ", 0
13 | prompt2 db "Введіть інше число: ", 0
14 | outmsg1 db "Введено ", 0
15 | outmsg2 db " і ", 0
16 | outmsg3 db ", сума цих чисел ", 0
17 | segment .bss
18 | input1 resd 1
19 | input2 resd 1
20 | ; сегмент коду
21 | segment .text
22 | global _asm_main  ; глобальна позначка
23 | _asm_main:
24 | enter 0,0         ; налаштування входу в підпрограму
25 | pusha             ; збереження регістрів
26 |
27 | mov eax, prompt1  ; виведення повідомлення
28 | call print_string
29 |
30 | call read_int     ; зчитування цілого і
```

```
31 mov [input1], eax ; запис у input1
32
33 mov eax, prompt2  ; виведення повідомлення
34 call print_string
35
36 call read_int     ; зчитування цілого і
37 mov [input2], eax ; запису у input2
38
39 mov eax, [input1] ; eax =  dword з input1
40 add eax, [input2] ; eax += dword з input2
41 mov ebx, eax      ; ebx = eax
42
43 dump_regs 1            ; виведення значення регістра
44 dump_mem 2, outmsg1, 1 ; виведення з пам'яті
45
46 ; виведення результату
47 mov eax, outmsg1
48 call print_string  ; виведення першого повідомлення
49 mov eax, [input1]
50 call print_int     ; виведення input1
51 mov eax, outmsg2
52 call print_string  ; виведення другого повідомлення
53 mov eax, [input2]
54 call print_int     ; виведення input2
55 mov eax, outmsg3
56 call print_string  ; виведення третього повідомлення
57 mov eax, ebx
58 call print_int     ; виведення суми (ebx)
59 call print_nl      ; виведення new-line
60
61 popa
62 mov eax, 0         ; повернення у програму Сі
63 leave
64 ret
```

У рядку 11 визначено сегмент ініціалізованих даних. В рядках 12-16 оголошені стрічки повідомлень, які завершуються `null` символом (ASCII код 0), а не символом '0', так як вони будуть друкуватися функціями бібліотеки Сі.

Сегмент неініціалізованих даних визначений у рядку 17. Позначення `.bss` означає "block started by symbol".

Сегмент коду визначений у рядку 21. У цьому сегменті розміщуються інструкції підпрограми. В рядку 22 визначено ім'я підпрограми як глобальне, а в рядку 23 задається позначка входу в підпрограму. Згідно домовленостей виклику Сі підпрограм під час компіляції імена функцій і глобальних змінних починаються із символу `"_"` . Це правило є специфічним для DOS/Windows, а компілятори Linux не використовують цей символ.

Директива `global` в рядку 22 визначає ім'я підпрограми видимим для зовнішньої області, тобто воно буде доступне для любого модуля програми.

Для спрощення написання асемблерних підпрограм, які будуть викликатися із основної Сі програми можна використати наступну структуру коду:

```
 1 %include "asm_io.inc"
 2 segment .data
 3 ;
 4 ; місце для розміщення ініціалізованих даних
 5 ;
 6
 7 segment .bss
 8 ;
 9 ; місце для розміщення неініціалізованих даних
10 ;
```

```
11
12 segment .text
13 global _asm_main
14 _asm_main:
15 enter 0,0      ; налаштування входу в підпрограму
16 pusha
17 ;
18 ; місце для розміщення коду
19 ;
20
21 popa
22 mov eax, 0     ; повернення у Сі програму
23 leave
24 ret
```

5. Домовленості, щодо виклику підпрограм асемблера з Сі програми

З мови Сі асемблер може викликатися як підпрограма або як вбудований (inline) асемблер. Виклик підпрограм асемблера є 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 Borland і Microsoft використовують формат MASM, а 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 `ebx, esi, edi, ebp, cs, ds, ss, es` використовуються підпрограмою. Однак, це не значить, що підпрограма їх не змінює. Підпрограма може їх змінити, але вона повинна відновити їх початковий стан перед поверненням з підпрограми. Значення регістрів `ebx`, `esi` і `edi` 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. В домовленостях виклику 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      dd 0
3  format db "x = %d\n", 0
4
5  segment .text
6  ...
7  push dword [x]     ; занесення у стек значення x
8  push dword format ; занесення у стек адреси формату стрічки
9  call _printf
10 add esp, 8         ; вилучення параметрів зі стеку
```

Регістри	Стек
esp+12	Значення параметру x
esp+8	Адреса формату стрічки
esp+4	Адреса повернення
esp	Збережене ebp

Мова Сі підтримує функції зі змінним числом параметрів, наприклад `printf()`, `scanf()`. Домовленості виклику 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 буде за адресою esp+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 eax,esp-8
```

Для обчислення адрес є спеціальна інструкція lea (load effective address – завантаження ефективної адреси). Адресу локальної змінної x можна обчислити так:

```
lea eax,[esp-8]
```

Виглядає так, ніби інструкція читає значення з [esp-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.

Повернення значень. Не void функції Сі повертають значення. Значення повертається через регістри. Всі внутрішні типи (char, int, enum і т.і.) повертаються через регістр eax. Якщо вони менші 32 біт, то вони розширюються до 32 біт при записуванні у eax. 64 бітні значення повертаються у парі регістрів edx:eax. Значення вказівників повертаються в eax. Значення з плаваючою крапкою повертаються в регістрі ST0 співпроцесора.

Підтримка декількох домовленостей про виклики підпрограм. Деякі компілятори підтримують декілька домовленостей про виклики підпрограм, наприклад компілятор GCC. Домовленості про виклики функції можуть бути явно задані 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 Pascal домовленостях). Тому stdcall може використовуватися 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 очистки стеку після інструкції call. Основний її недолік в тому, що вона не може використовувати функції зі змінним число аргументів.

6. Реєнтерабельні і рекурсивні підпрограми

Реєнтерабельна 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 ax, 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.

Реєнтерабельні підпрограми мають наступні переваги над звичайними:

• реентерабельні підпрограми можна викликати рекурсивно;

• реєнтерабельні 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);

• реєнтерабельні 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  ; факторіал n!
 2  segment .text
 3  global _fact
 4  _fact:
 5  enter 0,0
 6
 7  mov eax, [ebp+8]  ; eax = n
 8  cmp eax, 1
 9  jbe term_cond     ; умова завершення if n <= 1
10  dec eax
11  push eax
12  call _fact        ; eax = fact(n-1)
13  pop ecx           ; повернене значення в eax
14  mul dword [ebp+8] ; edx:eax = eax * [ebp+8]
15  jmp short end_fact
16  term_cond:
17  mov eax, 1
18  end_fact:
19  leave
20  ret
```

```
                Стек
              | n(3)
N=3, фрейм    | Адреса повернення
              | Збережене ebp
              | n(2)
N=2, фрейм    | Адреса повернення
              | Збережене ebp
              | n(1)
N=1, фрейм    | Адреса повернення
              | Збережене ebp
```

Висновки.

В програму асемблера параметри командного рядка передаються через стек.

В Linux асемблерні програми можуть звертатися до системних викликів через програмне переривання з номером `0x80`.

Аргументи окремих системних викликів передаються через регістри процесора, що забезпечує найбільшу швидкодію. Номер системного виклику поміщається у регістр `eax`. Для передачі аргументів використовується набір регістрів у фіксованому порядку: `ebx, ecx, edx, esi, edi, ebp`. Значення, яке повертає системний виклик, заноситься у регістр `eax`. У випадку помилки повертається негативне значення.

Переважно асемблер використовують для написання критичних модулів. При написанні програм для захищеного режиму основною є Сі програма, яка викликає асемблерні підпрограми. Це дозволяє ініціалізувати середовищем Сі всі сегменти і їх сегментні регістри.

Для спрощення написання асемблерних підпрограм можна використовувати однакову структуру коду.

Запитання.

1. Особливості процесу у захищеному режимі.
2. Передача параметрів командного рядка процесу.
3. Що таке системні виклики і як вони звертаються до ядра Linux.
4. Передача аргументів системним викликам.
5. Системні виклики читання/виведення через стандартні потоки.
6. Системні виклики для роботи з файлами.
7. Що таке підпрограма і як її можна викликати.
8. Призначення інструкцій `call, ret`.
9. Способи передачі параметрів у підпрограму.
10. Як змінюється стек при виклику і поверненні з підпрограми.
11. Стан стеку при виклику підпрограми зі змінним числом аргументів.
12. Збереження локальних змінних підпрограми у стеку. Інструкції `enter` і `leave`.
13. Переваги виклику асемблерних підпрограм з Сі програми.
14. Домовленості мови Сі по виклику підпрограм.
15. Особливості інтерфейсу мови Сі з підпрограмами асемблера.
16. Реєнтерабельні і рекурсивні підпрограми.

Література.

1. Юров В.И. Assembler. Учебник для вузов. 2-е изд. – СПб.: Питер, 2003. – 637 с.
2. Аблязов Р.З. Программирование на ассемблера на платформе x86-64. – М.: ДМК Пресс, 2011. – 304 с.
3. Магда Ю.С. Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006. — 410 с.
4. Столяров А.В. Программирование на языке асемблера NASM для ОС UNIX. Уч. пособие. – 2-е изд. – М.: Макс-пресс, 2011. – 188 с.

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 чисельних 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 1018.

• 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).

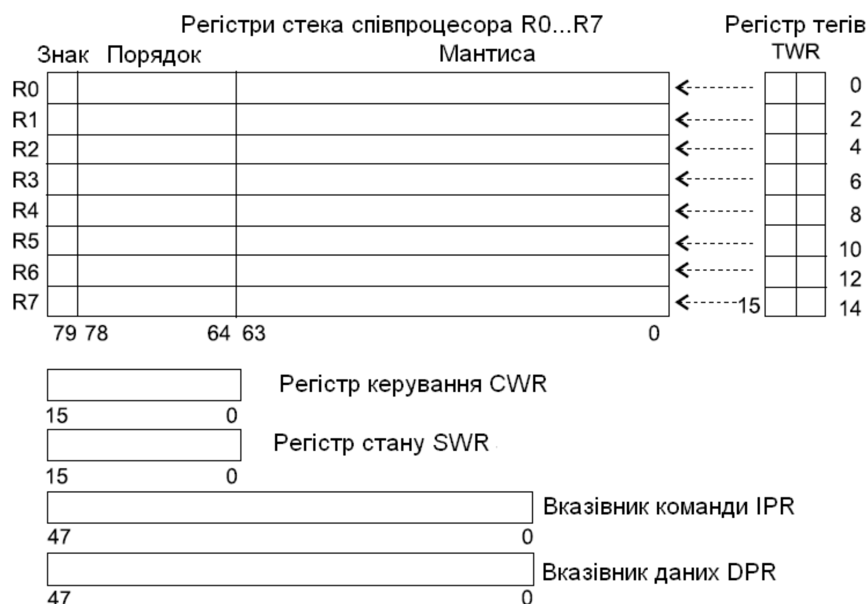


Рис. 1. Програмна модель співпроцесора

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 обробці обчислювальних алгоритмів. Реалізація чисельних алгоритмів на основі стека співпроцесора дозволяє отримати істотний виграш в 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 вершиною 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), що керує режимами роботи співпроцесора; за допомогою полів в цьому регістрі можна регулювати точність виконання чисельних 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 вказівників 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 обробці 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 охарактеризуємо 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 є 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

Рис. 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, 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 розробці і використанні) небажано прив'язувати їх за параметрами, що передаються до апаратних ресурсів (фізичних номерів регістрів співпроцесора). Набагато зручніше задавати порядок дотримання 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.

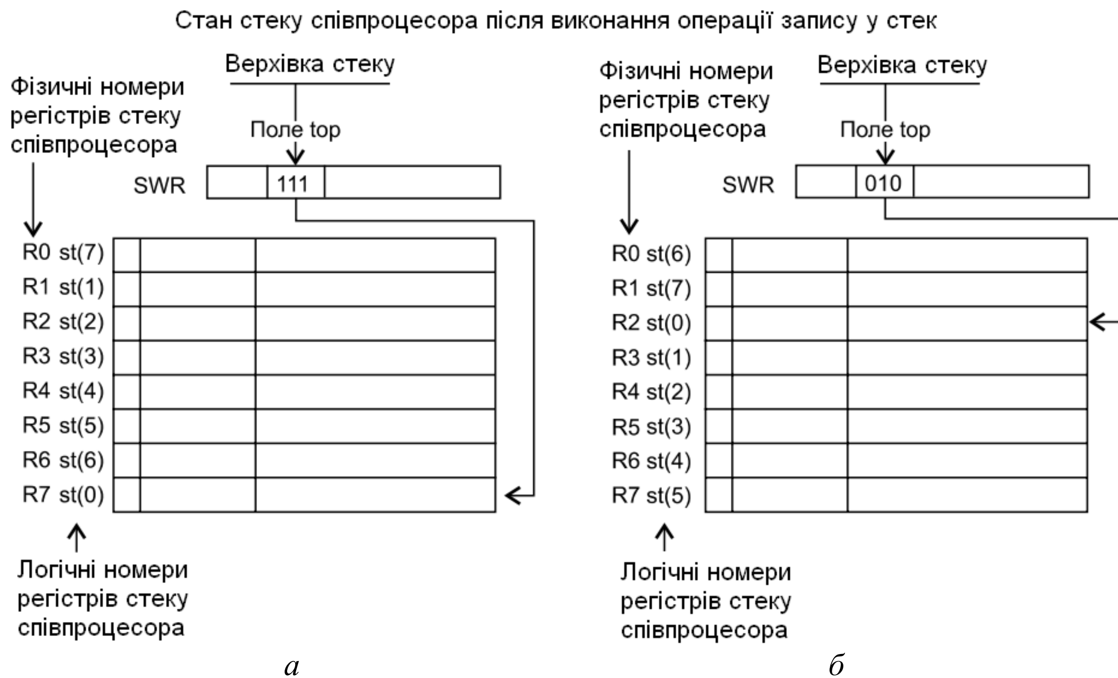


Рис. 3. Фізична і логічна нумерація регістрів стеку співпроцесора

Перш 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. Кожне 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.

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, яке до цього моменту має бути надане 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 перш 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 EFLAGS 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 порядка 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).

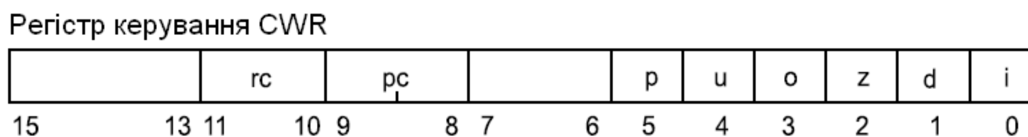


Рис. 4. Формат регістра керування співпроцесором CWR

Шість масок призначені для маскування винятковиx 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 в меншу сторону, тобто $m = a$;

• 10 – значення *m* округляється у велику 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 одно 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 абсолютними, а 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. Перерахуємо формати даних, з якими працює співпроцесор :

• двійкові цілі числа в трьох форматах – 16, 32 і 64 біти;

• упаковані цілі десяткові (BCD) числа – довжина максимального числа складає 18 упакованих 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 чисельні значення, до яких відносяться :

• денормалізовані дійсні числа – це числа, менші мінімального нормалізованого числа (див. нижче) для кожного речового 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 цілочисельними 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).

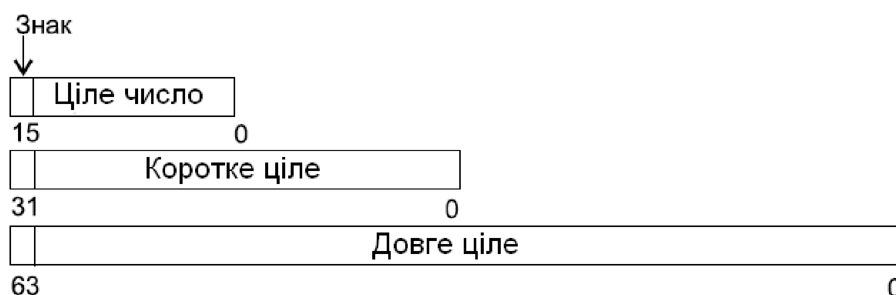


Рис. 5. Формати цілих чисел співпроцесора

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 обробка співпроцесором цілочисельних даних буде сповільнена 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). Для описання запакованого десяткового числа використовується директива DT. Ця директива 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.

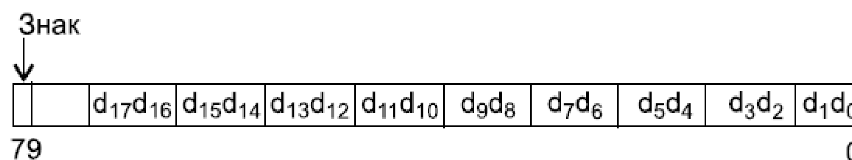


Рис. 6**.** Формат десяткового числа співпроцесора

Запаковані десяткові числа також подаються в стеку співпроцесора у розширеному форматі. Запаковані десяткові числа в програмі описуються директивою DT. Наприклад, ціле число 5 365 904 у форматі запакованого десяткового числа можна описати таким чином:

```
i dt 5365904
;подання в пам'яті: i dt=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 якими працює співпроцесор, – дійсний. Дані цього типу описуються трьома форматами: коротким, довгим і розширеним (рис. 7).

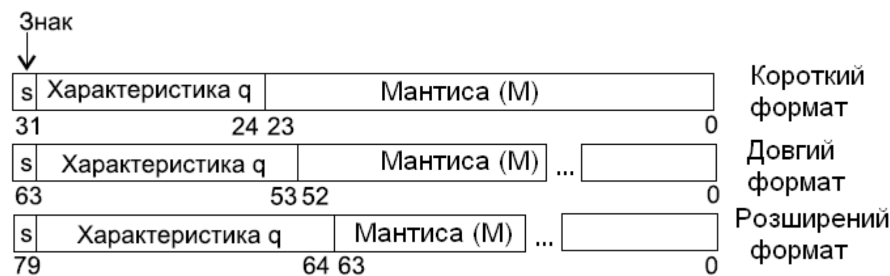


Рис. 7. Формати дійсних чисел співпроцесора

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 обробки в процесорі чисел з плаваючою крапкою його архітектурою накладаються деякі обмеження на компоненти формули (1).

Далі перераховані ці умови і обмеження для співпроцесорів, що застосовуються в архітектурі IA 32.

• Основа системи числення $N = 2$.

• Мантиса M має бути подана в *нормалізованому* виді. Нормалізація може відрізнятися для різних типів процесорів. Для ЄС ЕОМ, наприклад, мантиса нормалізованого числа повинна задовольняти умові $1/N \le |M| < 1$. Це означає, що старший біт подання має бути одиничним. Для випадку, коли $N = 2$, це відповідає відношенню $1/2 \le |M| < 1$ або в двійковому виді $0{,}10...00 \le |M| < 0{,}11...11$, тобто перша цифра після коми має бути значущою (одиницею), а порядок p, відповідно, таким, щоб ця умова виконувалася. Для архітектури співпроцесора IA 32 нормалізованим є число дещо іншого виду :

$$A = (-1)^s \cdot N^q \cdot M \qquad (2)$$

Тут:

• s – значення знакового розряду (0 – число більше нуля, 1 – число менше нуля);

• q – порядок числа, його призначення аналогічно призначенню порядку p у формулі(1), але як пояснюється далі, p і q – не одно і теж.

У цій формулі знак мають і порядок дійсного числа, і його мантиса. На рис. 7 видно, що формат зберігання дійсного числа в пам'яті має тільки поле для знаку мантиси. 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 q 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 характеристик 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 32розрядне число визначається директивою DD. При цьому обов'язковою в запису числа є наявність десяткової крапки, 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 «е».

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 «перевернутий» 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.

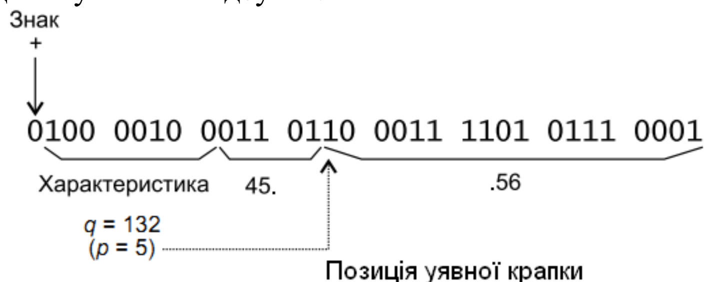


Рис. 9. Двійкове подання в пам'яті дійсного числа в директиві DD

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 1 4 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.

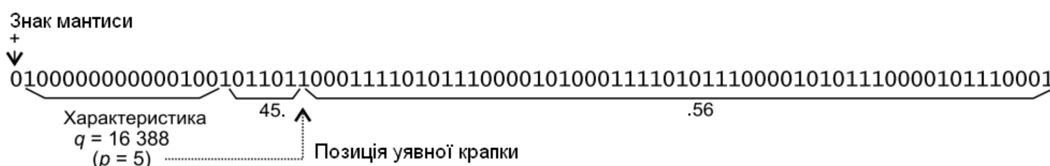


Рис. 10. Двійкове подання в пам'яті дійсного числа в директиві DT

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 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.

Рис. 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. 13), 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).

0	00.. 00	0000 ··· ··· ··· 000
79	78 64	63 0

1	00.. 00	0000 ··· ··· ··· 000
79	78 64	63 0

Рис. 13. Представлення 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 стека співпроцесора, що містить значення нескінченності, приведений на рис. 14.

0	11.. 11	10000 ··· ··· ··· 000
79	78 64	63 0

1	11.. 11	10000 ··· ··· ··· 000
79	78 64	63 0

Рис. 14. Подання значення нескінченності в регістрі стека співпроцесора

З рис. 14 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 стека 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. 15, 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.

а

x	11.. 11	10xxx ··· ··· ··· xxx
79	78 64	63 0

б

x	11.. 11	11xxx ··· ··· ··· xxx
79	78 64	63 0

в

1	11.. 11	11000 ··· ··· ··· 000
79	78 64	63 0

Рис. 15. Подання нечисел в регістрі стека співпроцесора

Спокійне нечисло – бітове значення з одиничним значенням полів характеристики і мантисою, перші два біти якої дорівнюють одиниці (рис. 15, б).

Співпроцесор самостійно не формує сигнальні числа, але як реакції на певні винятки він може формувати спокійні нечисла, наприклад нечисло дійсної невизначеності (рис. 15, 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 стека 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. 16).

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'язково є витягання 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.

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 вираження 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. В Так дерево виразу $a + b \cdot c - d / (a + b)$ зображено на рис. 17, а постфіксний запис матиме вид abc × + dab + / –.

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 стека.

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.

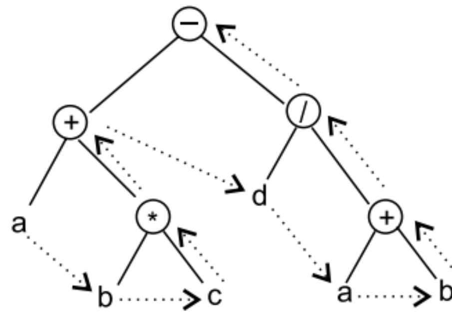


Рис. 17. Подання виразу у вигляді дерева

Команди співпроцесора

- Передачі даних
  - Дійсні дані
    - fld
    - fst(p)
  - Цілочисельні дані
    - fild
    - fist(p)
  - Десяткові дані
    - fbld
    - fbst(p)
  - Завантаження констант
    - fldz
    - fld1
    - fldpi
    - fldl2t
    - fldl2e
    - fldlg2
    - fldln2
  - Обмін
    - fxch
  - Умовне пересилання (Pentium II/III)
    - fcomi
    - fcomip
    - fucomi
- Порівняння даних
  - Дійсні дані
    - fcom/fcomp(p)
    - fucom(p)(pp)
  - Цілочисельні дані
    - ficom(p)
  - Аналіз st(0)
    - fxam
  - З нулем
    - ftst
  - Умовне порівняння (Pentium II/III)
    - fcomi
    - fcomip
    - fucomi
- Арифметичні
  - Дійсні дані
    - Додавання
      - fadd(p)
    - Віднімання
      - fsub(r)/sub(r)p
    - Множення
      - fmul(p)
    - Ділення
      - fdiv(r)/fdiv(r)p
  - Цілочисельні дані
    - Додавання
      - fiadd
    - Віднімання
      - fisub(r)
    - Множення
      - fimul
    - Ділення
      - fidiv(r)
  - Допоміжні арифметичі команди
    - fsqrt — fprem
    - fabs — fprem1
    - fchs — fscale
    - fxtract — frndint
- Трансцендентні
  - Обчислення тригонометричних функцій
    - Синус
      - fsin
      - fsincos
    - Косинус
      - fcos
    - Тангенс
      - fptan
    - Арктангенс
      - fpatan
  - Обчислення логарифмів і степенів
    - fldl2t
    - fldl2e
    - fldlg2
    - fldln2
    - f2xm1
    - fyl2xp1
    - fyl2x
- Керування
  - Ініціалізація співпроцесора
    - f(n)init
  - Робота із середовищем
    - f(n)save
    - f(n)stenv/fldenv
    - f(n)clex
    - fldcw/f(n)stcw
    - f(n)stsw
    - frstor
  - Робота із стеком
    - fincstp/fdecstp
    - ffree
  - Перемикання режимів
    - fsetpm/frstpm
  - fnop
  - fwait

Рис. 18. Функціональна класифікація команд співпроцесора

6.1. Команди передачі даних

Група команд передачі даних призначена для організації обміну між регістрами стека, вершиною стека 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 цілочисельному 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 стека співпроцесора.

• `FST приймач` – збереження дійсного числа з верхівки стека співпроцесора в пам'ять. Як випливає з аналізу мнемокоду команди (відсутній символ Р), 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 цілочисельному 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 стекк 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 стека – 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 логарифма десяти;
- `FLDL2E` – завантаження двійкового логарифма експоненти (числа е);
- `FLDLG2` – завантаження десяткового логарифма двійки;
- `FLDLN2` – завантаження натурального логарифма 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 стека 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 у вершині стека співпроцесора ST(0) з цілим операндом в пам'яті. Після порівняння і установки бітів С3...С0 команда виштовхує значення з ST(0). Довжина цілого операнду – 16 або 32 біта, тобто це ціле слово і коротке ціле(див. таблицю. 17.1).

• `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) > операнду, то 000;
• якщо ST(0) < операнду, то 001;
• якщо ST(0) = операнду, 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 EFLAGS), треба якось записати сформовані біти умови C3, C2, C0 в регістр EFLAGS. У системі команд співпроцесора існує команда `FSTSW`, яка дозволяє запам'ятати слово стану співпроцесора в регістрі AX або комірку 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 EFLAGS/FLAGS здійснюється командою `SAHF`. Ця команда записує вміст AH в молодший байт регістра EFLAGS/FLAGS. 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 стека 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. Цілочисельні арифметичні команди

Цілочисельні 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.

Цілочисельні арифметичні команди забезпечують велику гнучкість задання операндів.

• `FIADD джерело` – команда додає значення ST(0) і цілочисельне джерело, яким виступає 16- або 32-розрядний операнд в пам'яті. Результат додавання запам'ятовується в регістрі стека співпроцесора ST(0).

• `FISUB джерело` – команда віднімає значення цілочисельного джерела з ST(0). Результат віднімання запам'ятовується в регістрі стеку співпроцесора ST(0). Джерелом виступає 16- або 32-розрядний цілочисельний операнд в пам'яті.

• `FIMUL джерело` – команда множить значення цілочисельного джерела на вміст ST(0). Результат множення запам'ятовується в регістрі стеку співпроцесора ST(0). Джерелом виступає 16- або 32-розрядний цілочисельний операнд в пам'яті.

• `FIDIV джерело` – команда ділить вміст ST(0) на значення цілочисельного джерела. Результат ділення запам'ятовується в регістрі стеку співпроцесора ST(0).

Джерелом виступає 16- або 32-розрядний цілочисельний 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) з цілочисельного джерела. Результат віднімання запам'ятовується в регістрі стеку співпроцесора ST(0). Джерелом виступає 16- або 32-розрядний цілочисельний операнд в пам'яті.

• `FIDIVR джерело` – команда ділить значення цілочисельного джерела на вміст ST(0). Результат ділення запам'ятовується в регістрі стеку співпроцесора ST(0). Джерелом виступає 16- або 32-розрядний цілочисельний операнд в пам'яті.

6.4.2. Дійсні арифметичні команди

Схема розміщення операндів дійсних команд традиційна для команд співпроцесора. Один з операндів розміщується у 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 пераховані раніше дійсні формати за винятком розширеного.

На відміну від цілочисельних 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) із значення в ST(0). Результат віднімання запам'ятовується в регістрі стеку співпроцесора ST(0).

• FSUB джерело – команда віднімає значення джерела зі значення в ST(0). Джерело задає адресу елементу 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) із 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) із 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(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 – команда множить значення в ST(0) на вміст довільного регістра стеку ST(i). Результат множення запам'ятовується в регістрі стеку співпроцесора ST(i).

• FMULP st(i), st – 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(0) на значення регістра співпроцесора ST(1). Результат ділення запам'ятовується в регістрі стеку співпроцесора ST(0).

• FDIV 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 – команда робить ділення аналогічно команді FDIV st(i), але результат ділення запам'ятовується в регістрі стеку співпроцесора ST(i).

• FDIVP st(i), st – команда ділить аналогічно команді FDIV st(i), s.

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(i), st – команда ділить вміст регістра ST(i) на вміст верхівки регістра співпроцесора ST(0). Результат ділення запам'ятовується в регістрі стек співпроцесора ST(0).

• FDIVRP st(i), st – 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 показових функцій.

Команди трансцендентних функцій.

• 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.

Команда `WAIT/FWAIT` – це команда очікування. Вона призначена 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 стека на одиницю (поле TOP) в регістрі SWR. Команда не має операндів. Дія команди FINCSTP подібно 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 стека (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 стеке 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.

Висновки.

У програмній моделі співпроцесора можна виділити три групи регістрів: вісім регістрів стеку співпроцесора R0…R7 і три службові регістри – стану SWR, контролю CWR, тегів TWR.

Команди співпроцесора оперують не фізичними номерами регістрів стеку R0…R7, а логічними номерами регістрів ST(0)…ST(7).

Регістр SWR відображає поточний стан співпроцесора після виконання останньої команди. Регістр керування співпроцесором CWR визначає особливості обробки числових даних. Регістр тегів TWR є сукупністю дворозрядних полів. Кожне дворозрядне поле відповідає певному фізичному регістру стека і характеризує його поточний стан.

Співпроцесор підтримує наступні формати даних:

• двійкові цілі числа в трьох форматах – 16, 32 і 64 біти;

• упаковані цілі десяткові (BCD) числа – довжина максимального числа складає 18 упакованих 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 чисельні значення, до яких відносяться :

• денормалізовані дійсні числа – це числа, менші мінімального нормалізованого числа (див. нижче) для кожного речового формату, підтримуваного співпроцесором;

• нуль;

• позитивні і негативні значення нескінченності;

• нечисла;

• невизначеності і непідтримувані формати.

Система команд співпроцесора включає близько 80 машинних команд. Команди поділяються на наступні групи: передачі даних, порівняння даних, арифметичні, трансцендентні, керування.

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.

Література.

1. Столяров А.В. Программирование на языке ассемблера NASM для ОС UNIX. Уч. пособие. – 2-е изд. – М.: Макс-пресс, 2011. – 188 с.

2. Аблязов Р.З. Программирование на ассемблера на платформе x86-64. – М.: ДМК Пресс, 2011. – 304 с.

3. Магда Ю.С. Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006. — 410 с.

4. Кип Р. Ирвин. Язык ассемблера для процесоров Intel. – 4-е изд., Пер. с англ. – М.: Издательский дом ”Вильямс”, 2005. – 912 с.: ил.

5. Харт Джонсон М. Системное программирование в среде Windows / Джонсон М. Харт ; пер. с англ. — М. : Издательский дом «Вильямс», 2005.

6. Юров В.И. Assembler. Учебник для вузов. 2-е изд. – СПб.: Питер, 2003. – 637 с.

7. Голубь Н.Г. Исскуство программирования на ассемблере. Лекции и упражнения. – 2-е изд., испр. и доп., – СПб.: ДиаСофт, 2002. – 656 с.: ил.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Столяров А.В. Программирование на языке асемблера NASM для ОС UNIX. Уч. пособие. – 2-е изд. – М.: Макс-пресс, 2011. – 188 с.
2. Аблязов Р.З. Программирование на ассемблера на платформе x86-64. – М.: ДМК Пресс, 2011. – 304 с.
3. Магда Ю.С. Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006. — 410 с.
4. Харт Джонсон М. Системное программирование в среде Windows / Джонсон М. Харт ; пер. с англ. — М. : Издательский дом «Вильямс», 2005.
5. Кип Р. Ирвин. Язык ассемблера для процесоров Intel. – 4-е изд., Пер. с англ. – М.: Издательский дом ”Вильямс”, 2005. – 912 с.: ил.
6. Юров В.И. Assembler. Учебник для вузов. 2-е изд. – СПб.: Питер, 2003. – 637 с.
7. Голубь Н.Г. Исскуство программирования на ассемблере. Лекции и упражнения. – 2-е изд., испр. и доп., – СПб.: ДиаСофт, 2002. – 656 с.: ил.
8. Електронний ресурс: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.