Тема: Система команд процесора архітектури IA-32.

Команда процесора – інструкція, що визначає операцію яка повинна бути виконана процесором.

Класифікація команд.

Система команд мікропроцесора поділяється на сім груп :

- Команди пересилання даних.
- Арифметичні команди.
- Логічні команди або команди маніпулювання бітами.
- Команди передачі керування.
- Команди обробки рядків (ланцюгові команди).
- Команди керування процесором.
- Команди переривання.

1. Команди пересилання даних.

Види:

- *- Обміну даними*
- *- Роботи з портами вводу виводу*
- *-Роботи зі стеком*

Особливості:

- ***Не можна пересилати дані з однієї комірки пам'яті в іншу.***
- ***Не можна завантажити в регістр сегмента операнд із безпосередньою адресацією.***
- ***Не можна переслати значення одного регістра сегмента в іншій.***
- ***Не можна використовувати регістри CS і IP як приймач у команді MOV.***

MOV dest,src

команда є основою для реалізації **оператора присвоєння** в мовах високого рівня

Команда виконує пересилання даних в регістр з регістру, памяті або безпосереднього операнда.

Приклади:

```
MOV AX,10
MOV EBX,ESI
MOV AL, BYTE PTR MEM
```

XCHG r/m,r	Обмін данними між регістрами або регістром и памяттю.
MOVSXB r,r/m MOVSXW r,r/m	**Пересилання з розширенням** з дублюванням знакового біту: MOVSXB AX,BL; MOVSXB EAX,byte ptr mem. MOVSXW EAX,WORD PTR MEM
MOVZXB r,r/m MOVZXW r,r/m	**Пересилання байта з розширенням** до слова чи подвійного слова з дублюванням нульового біта: MOVSXB AX, BL MOVSXB EAX, byte ptr mem.
XLAT	Завантажити в AL байт з таблиці в сегменті даних, на початок якої вказує EBX (BX), при цьому початкове значення AL грає роль зсуву.
LEA r,m	Завантаження ефективної адреси. (операція взяття адреси) Наприклад, LEA EAX, MEM

Команди роботи з портами вводу виводу.

- **IN** AL (AX, EAX), Port
  IN AL (AX, EAX), DX
  Введення в акумулятор з порту введення-виведення. Порт адресується безпосередньо або через регістр DX.

- **OUT** port, AL (AX, EAX)
  OUT DX, AL (AX, EAX) Вивід в порт вводу-виводу. Порт адресується безпосередньо або через регістр DX.

- [REP] **INSB**
  [REP] INSW
  [REP] INSD Виводить дані з порту, що адресується регістром DX в комірку пам'яті ES: [EDI / DI]. Після введення байта, слова чи подвійного слова проводиться корекція EDI / DI на 1, 2
- [REP] **OUTSB**
  [REP] OUTSW
  [REP] OUTSD Виводить дані з комірки пам'яті, яка визначається регістрами DS: [ESI / SI], у вихідний порт, адреса якого знаходиться в регістрі DX. Після виведення байта, слова, подвійного слова проводиться корекція покажчика ESI / SI на 1, 2, 4 в залежності від розмірності .
- *При наявності префікса REP-процес триває, поки вміст CX не стане рівним 0*

Робота зі стеком

- **PUSH r / m -** Помістити в стек слово або подвійне слово, рекомендується в будь-якому випадку поміщати в стек подвійне слово.
- **PUSHA** Помістити в стек значення регістрів EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.

- **POP reg / mem** Витягти з стека слово або подвійне слово.
- **POPA** Витяг з стека даних в регістри EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.Команда з'явилася, починаючи 386 процесора.
- **PUSHF** Приміщення в стек регістра прапорів.
  POPF Витягти даних у регістр прапорів.

- Приклад:
- PUSH ECX *; зберігаємо в стеку значення регістра лічильника*
- …
- POP ECX *; відновлюємо зі стеку значення регістра лічильника*

2. Арифметичні команди

- **ADD** dest, src - Додавання двох операндів. Перший операнд може бути регістром або коміркою пам'яті, другий - регістром, коміркою пам'яті, константою. Неможливо тільки, коли обидва операнди є коміркою пам'яті.
- dest=dest+src
- **SUB** dest, src Віднімання двох операндів. Останнє аналогічно додаванню
- dest=dest-src
  DEC r / m декремент операнда.
- **CMP** r / m, r / m Віднімання без зміни операндів (порівняння).
  NEG r / m Зміна знаку операнда.
  MUL r / m Множення AL (AX, EAX) на ціле беззнаковое число. Результат, відповідно, буде міститися в AX, DX: AX, EDX: EAX.
- [EDX: EAX] = EAX * r / m
  IMUL r / m Знакове множення (аналогічно MUL). Всі операнди вважаються знаковими. Команда IMUL має також двухоперандний і трехоперандний вигляд.
  DIV r / m (src) беззнакове ділення. Аналогічно беззнакового множення. Здійснює ділення акумулятора і його розширення (AH: AL, DX: AX, EDX: EAX) на дільник src. Частка міститься в акумуляторі, а залишок - в розширенні акумулятора.
- [EDX: EAX] = EAX **/** src; *EAX – частка, EDX залишок*
  IDIV r / m Знакове ділення. Аналогічно беззнакового.

додаткові

- **XADD** dest, src Дана операція виконує на початку обмін операндами, а потім виконує операцію ADD.
- **ADC** dest, src Додавання з урахуванням прапора перенесення - в молодший біт додається біт (прапор) переносу.
- **SBB** dest, src Віднімання з урахуванням біта переносу. З молодшого біта віднімається біт (прапор) переносу.
- **CMPXCHG** r, m, a Порівняння з обміном. Сприймає ці три операнда (регістр - операнд - джерело, комірка пам'яті - операнд - одержувач, акумулятор, тобто AL, AX чи EAX) Якщо значення в операнде-одержувача і акумуляторі рівні, операнд-одержувач замінюється операндом-джерелом, початкове значення операнда - одержувача завантажується в акумулятор.
- **CBW** Розширення байта (AL) в слово з копіюванням знакового біта.
  CWD Розширення слова (AX) в подвійне слово (DX: AX) з копіюванням знакового біта.
  CWDE Розширення слова (AX) в подвійне слово (EAX) з копіюванням знакового біта.
  CDQ Перетворення подвійного слова (EAX) в учетверене слово (EDX: EAX).

- В арифметичних командах встановлюються або скидаються 6 прапорів стану:

- **CF** - установлюється, якщо операція дала беззнаковий результат поза діапазоном (тобто є перенос у знаковий розряд). Позика (7,15,31) викликає вихід з розрядної сітки.

- **OF** - установлюється, якщо в операції вийшов знаковий результат, що знаходиться поза діапазоном (тобто перенос у знаковий розряд) не створює переносу з розрядної сітки або перенос з розрядної сітки відбувається без переносу в знаковий розряд.

- **ZF** - установлюється, якщо результат операції (знаковому або беззнаковий) дорівнює нулеві.

- **SF** - установлюється, якщо старший біт результату операції містить 1, показуючи негативне число.

- **PF** - установлюється, якщо результат операції містить парне число одиничних битов.

- **AF** - установлюється, якщо в десяткових операціях потрібна корекція


Команди передачі управління
Безумовні
Умовні
Управління циклом
Безумовний перехід
Виклик (повернення з) процедури
Виклик (повернееня з) переривань
По регістру прапорів
Арифметичних операцій
Стану регістра ECX
беззнакові
Знакові

Мітки

- При використанні команд переходів використовуються мітки, що являють собою символічне означення адреси певної команди в програмі.
- (змінні символічне означення певної ділянки пам'яті)
- Способи визначення міток
- 1. Через оператор **:** (основний)

  приклад

  M01 : КОП

- 2. Через директиву **label**
- приклад

  M01 LABEL NEAR ; *FAR*

  КОП

- Для всіх команд: мітка обов'язково повинна бути визначена у процедурі у якій виконаний перехід. Перехід за межі процедури **НЕДОПУСТИМИЙ.**

Безумовні переходи

- 1. **JMP label -** безумовний перехід на мітку (аналог команди ***goto***) При роботі в Windows використовується в основному внутрішньосегментний перехід (NEAR) у межах 32-бітного сегмента.
- 2. **CALL NameProc** Виклик процедури**.** *(Часто використовується директива **INVOKE**, котра розгортається в команду **CALL**)*
- 3. **RET** [DWORD]. Повернення з процедури
- 3. **INT** kod_interup Виклик переривання
- 4. **IRET** Повернення з переривання.

- **CALL** target Передає управління процедурою (мітці) із збереженням у стеку адреси, наступної за CALL-командою. У плоскій моделі адресу повернення представляє собою 32-бітне зсув. Міжсегментний виклик передбачає збереження в стеці селектора і зсуву, тобто 48-бітної величини (16 біт - селектор і 32 біта - зміщення).
- **RET** [N] Повернення з процедури. Необов'язковий параметр N припускає, що команда також автоматично чистить стек (звільняє N байт). Команда має різновиди, які вибираються асемблером автоматично, в залежності від того, є процедура ближній або дальній. Можна, однак, і явно вказати тип повернення (RETN або RETF). У випадку плоскої моделі за замовчуванням береться RETN з чотирибайтових адресою повернення

Умовні переходи:

На основі значень прапорів регістра стану:

- JE / JZ - перейти, якщо нуль.
- JNE / JNZ - перейти, якщо менше або дорівнює.

- JC - перейти, якщо перенесення.
- JNC - перейти, якщо немає перенесення.

- JO - перейти, якщо є переповнення.
- JNO - перейти, якщо немає переповнювання.

- JP / JPE - перейти, якщо є паритет.
- JNP / JPO - перейти, якщо немає паритету.

- JNS - перейти, якщо немає знаку.
- JS - перейти, якщо є знак.

По значенню регіста CX (ECX)

- JCXZ - перехід, якщо CX = 0.
- JECXZ - перехід, якщо ECX = 0.

- **Арифметичні без знакові**
- JA / JNBE - перейти, якщо вище.
  JAE / JNB - перейти, якщо вище або дорівнює.
  JB / JNAE - перейти, якщо нижче.
  JBE / JNA - перейти, якщо нижче.
- **Арифметичні знакові**
- JG / JNLE - перейти, якщо більше.
  JGE / JNL - перейти, якщо більше або дорівнює.
  JL / JNGE - перейти якщо менше.
  JLE / JNG - перейти, якщо менше або дорівнює.
- ----------
- JZ/JNZ - == != завжди однакові

Команди управління циклом.

- Команди цієї групи використовують в якості лічильника регістр **ECX**, зменшують вміст його вміст на 1 (ECX--), при кожному виконанні команди.

- **LOOP** - перехід, якщо вміст ECX не дорівнює нулю.
- **LOOPE** (LOOPZ) - перехід, якщо вміст ECX не дорівнює нулю і прапор ZF = 1.
- **LOOPNE** (LOOPNZ) - перехід, якщо вміст ECX не дорівнює нулю і прапор ZF = 0.
-

Команди управління прапорами

Параметрів не мають, сама команда визначає операнди

- **CLC** Скидання прапора перенесення.
  CMC Інверсія прапора перенесення.
  STC Установка прапора перенесення.
  CLD Скидання прапора напрямку.
  STD Установка прапора напрямку.
  CLI Заборона маскуються апаратних переривань.
  STI Дозвіл маскуються апаратних переривань.
  CTS Скидання прапора перемикання завдань.

Логічні операції.

Використовуються при перевірці певних бітів операндів та створення кодових послідовностей. Характеризуються високою швидкодією

- **AND** dest, src Логічна операція «AND». Обнулення біт dest, які дорівнюють нулю у src.
  TEST dest, src Аналогічна «AND», але не змінює dest. Використовується для перевірки ненульових біт.
  OR dest, src Логічна «АБО». У dest встановлюються біти, відмінні від нуля в src.
  XOR dest, src Виключне «АБО».
  NOT dest Перемикання всіх біт (інверсія).
-

Команди зсуву та обертання

Починаючи з мікропроцесора 386, безпосередній операнд src може бути не тільки 1, але довільним числом. У ранніх версіях для кількості зсувів використовувався регістр CL.

RCL / RCR dest, src Циклічний зсув вліво / вправо через біт перенесення CF. Src може бути або CL, або безпосередній операнд.

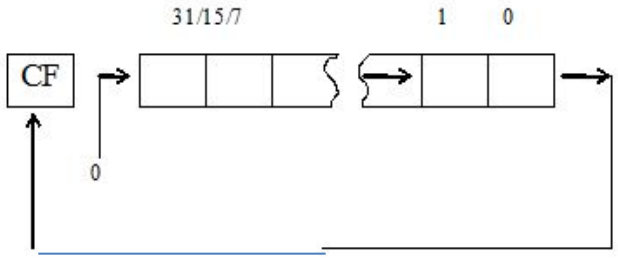
ROL / ROR dest, src Аналогічно командам RCL / RCR, але за іншим, працює з прапором CF. Прапор не бере участь у циклі, але в нього потрапляє біт, який перейшов з початку на кінець чи навпаки.

SAL / SAR dest, src Зрушення вліво / право. Називається ще арифметичним зрушенням. При зсуві вправо дублюється старший біт. При зсуві вліво молодший біт заповнюється нулем. Минулий біт поміщається в CF.

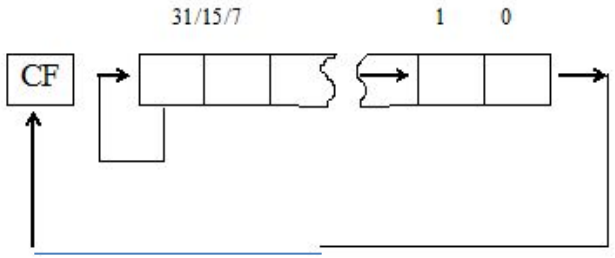
SHL / SHR dest, src Логічний зсув вліво / вправо. Зсув вправо відрізняється від SAR тим, що і старший біт заповнюється нулем.

SHLD / SHRD dest, src, count Трехоперандні команди зсуву вліво / вправо. Першим операндом, як зазвичай, може бути або регістр, або комірка пам'яті, другим операндом повинен бути регістр загального призначення, третім - регістр CL або безпосередній операнд. Суть операції полягає в тому, що dest і src на початку об'єднуються, а потім виробляється зрушення на кількість біт count. Результат знову поміщається в dest.

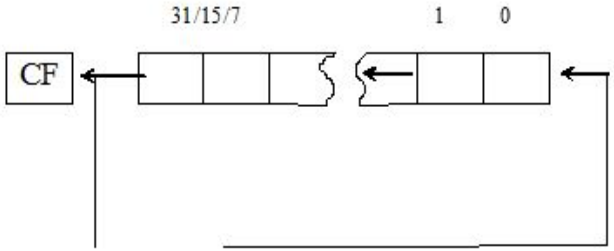
SHR



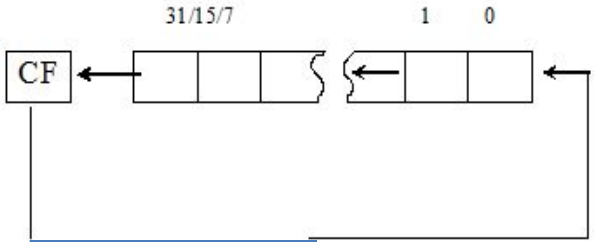
SAR



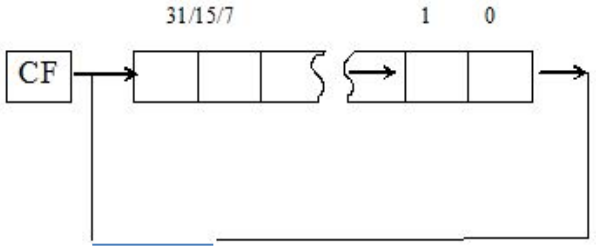
ROL



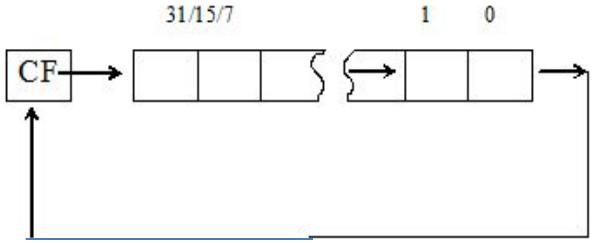
RCL



ROR



RCR



Команди роботи з ланцюгами

- Під **ланцюгами** асемблер розуміє послідовно розміщений в пам’яті набір з байт, слів чи двійних слів (*в С++ - масив*) .
- Команди використовуються три елементи:

- 1. Прапор напрямку - DF. (Direction Flag).
- 2. Регістри адрес джерела та приймача даних EDI та ESI.
- 3. Префікс команди. (не всі команди).
-
- Загальний формат ланцюгової команди.
- **[PREF] КОП [dest, source]**

- **Порядок використання**
- Встановити прапор напряму
- Завантажити ефективні адреси в регістр
- Ініціалізувати лічильник
- Виконати команди

- **Напрям:**
- якшо **DF = 0** (команда CLD) , то команди обробки ланцюгів виконують операції в **напрямку зростання адресів**, тобто від нульового елемента до найстаршого, якщо **DF=1** (команда STD) встановлений в одиницю то операції виконуються в **напрямку зменшення адресів.**

- **Встановлення адрес джерела та приймача даних.**
- EDI приймач (destination) даних.
- ESI джерело (source) даних.
- **Префікс – задає правила виконання команд**

Префікс	Дія	Умова
REP	**dec ECX** **ADD EDI, sizeof (Elem)** **ADD ESI, sizeof (Elem)**	**ECX !=0**
REPE	**dec ECX** **ADD EDI, sizeof (Elem)** **ADD ESI, sizeof (Elem)**	**ECX!=0 AND ZF=0**
REPNE	**dec ECX** **ADD EDI, sizeof (Elem)** **ADD ESI, sizeof (Elem)**	**ECX!=0 AND ZF=1**

Команда ланцюгового пересилання даних

- PREF **MOVS** dest , source

- пересилання даних з однієї області пам’яті в іншу.
- Якщо команда використовується без аргументів, то необхідно вказати підвид команди який характеризує тип даних ланцюга з якими буде працювати команда. Без префіксу команда виконується один раз. (аналогічно MOV).
- MOVSB - 1 байт
- MOVSW - 2 байти
- MOVSD - 4 байти
- **Приклад:** функція **strcpy()** мови **C**

```
.data
MEMW1 DW 10 DUP(1),5 DUP(0)
MEMW2 DW 15 DUP(?)
.code
CLD
LEA ESI, MEMW1
LEA EDI, MEMW2
MOV ECX,15
REP MOVSW ; або REP MOVS MEMW2,MEMW1
```

- CMPS dest,src *Виконується поелементне порівняння, по умові яка задається префіксом, до першого співпадання чи неспівпадання для кількості елементів що знаходяться в ECX елементів двох ланцюгів адреси яких знаходяться в EDI та ESI.*

- *Префікс REPE поки рівні, REPNE поки не рівні.*

- *Приклад:* функція **strcmp()** мови **C**
- CLD

  MOV ECX,N ;довжина

  LEA ESI,STR1

  LEA EDI,STR2

  REPE **CMPSB**
- JNE NO_SAME
- JMP _SAME

- **Команда сканування**
- **SCAS (B W D)** виконує порівняння значення акумулятора і ланцюга символів, який заданий регістром EDI аналогічно до префіксів CMPS
- Приклад 4. Пошук в ланцюгу знаків '$' и замена їх на підкреслення.
- `CLD`
- `MOV AL,'$'`
- `LEA EDI,STR2`
- `MOV ECX,20`
- `LO:`
- `REPNE SCASB  ;здесь поиск`
- `JNZ DONE`
- `MOV BYTE PTR [EDI-1],"_" ;а здесь замена`
- `JMP LO`
- `DONE:`
- **LODS (B W D) [ESI] . Завантаження (без префіксу)** - виконує переміщення ланцюга з області заданої ESI в акумулятор
- **STOS [EDI] Збереження** -пересилка значення акумулятора в область пам'яті яка адресується EDI
- **Команди роботи з портами:**
- **INS [EDI] вивід в порт**
- **OUTS [ESI] ввід з порту**
- для обміну великих блоків пам'яті між областями пам'яті і периферійними пристроями. Використовують префікси. Адреса порту вводу виводу задається регістром DX.

Складні типи даних

- Під **масивом** розуміють набір елементів одного типу послідовно розміщених в пам'яті. Перед застосуванням масиву в Assembler потрібно зарезервувати область пам'яті.
- **Резервування виконується в блоці даних з використання одного з двох методів:**
- 1. Безпосереднім заданням елементів
- 2. З використанням директива **DUP** із визначенням кількості елементів
- **Приклад:**
- .DATA
- MAS1 DD 0,1,2,3,4
- MAS2 DD 5 DUP(0)
-
- Локальний масив можна оголосити тільки наступним чином:
- LOCAL NameArray [const_size]:TYPE

- **Приклад**:
- LOCAL array[20]:BYTE
- початкова ініціалізація не виконується.
- Для доступу до елемента масиву можуть використовуватись різні види адресації.
- В загальному випадку доступ здійснюється за правилом

Доступ до елементів масиву

- Нумерація елементів масиву починається з нуля. **ОБОВ'ЯЗКОВО.** Назва масиву вказує на зміщення в памяті до першого елементу масиву. Інші визначаються як зміщення від початкового.
- Для доступу до елемента масиву можуть використовуватись різні види адресації.
- В загальному випадку доступ здійснюється за правилом
- ***База+(індекс*розмір)***
- Найпростіший вид – непряма базова адресація. Всі дії по формуванню виконуються під час роботи програми.
- **Приклад:**

```
.DATA
MAS1 DD 1,2,0,4,5
.CODE
LEA ESI, MAS1
MOV EAX, [ESI]; нульовий елемент
ADD ESI, 4
MOV EBX, [ESI] ; перший елемент
ADD ESI, 4
ADD EAX, EBX
MOV [ESI], EAX ; сума нульового і першого в другий
```

Структури

- **Структура — це тип даних, що складає з фіксованого числа елементів різного типу.**
- **Для використання структур у програмі необхідний виконати три дії:**
- 1. Задати шаблон структури. За змістом це означає визначення нового типу даних, що згодом можна використовувати для визначення змінних цього типу.
- 2. Визначити екземпляр структури. Цей етап має на увазі ініціалізацію конкретної змінної з заздалегідь визначеної (за допомогою шаблона) структурою.
- 3. Організувати звертання до елементів структури.
- **Загальний синтаксис оголошення.**
- **NameStruc STRUC**
  <поля>
- **NameStruc ENDS**
- **Приклад задання шаблону (оголошення):**
- RECT STRUCT
- LEFT DD ?
  TOP DD ?
  RIGHT DD ?
  BOTTOM DD ?
- RECT ENDS
- **Створення (визначення)**
- **.data**
- **R1 RECT <?>**
- **R2 RECT <1,1,800,600>**
- **Доступ. Виконується за допомогою оператора « . »**
- **MOV R1.LEFT, 20**

Об’єднання

- **Об’єднання** – складний тип даних, призначений для збереження даних різного типу в одній області пам’яті. Розмір об’єднання визначається розміром максимального елемента. Всі елементи вирівнюються по початковій змінній даного типу
- **Загальний синтаксис:**
- **NameUnion UNION**
  < поля >
- **NameUnion ENDS**
-
- **Приклад:**
-
- ```
  Extend UNION
  ```
- ```
        Ascci BYTE ?
  ```
- ```
       Kanj WORD ?
  ```
- ```
  Extend ENDS
  ```
-
- ```
  INITIAL  EXTENDCHAR <J>    ;CREATE A UNION VARIABLE
  ```
- ```
  MOV   AL, INITIAL.ASCII    ;MOVE BYTE-LENGTH DATA
  ```
- ```
  MOV   BX, INITIAL.KANJI    ;MOVE WORD-LENGTH DATA
  ```
  ```

Області видимості

- Під областю видимості змінної (переміщуваного операнду) розуміють ділянку програми в якій змінна доступна до читання чи модифікації

- 1. Глобальна область видимості – проект.
- 2. Файлова область видимості – файл.
- 3. Локальна область видимості – процедура.

- Для використання **глобальних** змінних використовують дві директиви **PUBLIC і EXTRN.**
- **Приклад:**

```
;Оголошення          Використання
;файл T01.asm           T2.asm
PUBLIC A01            EXTRN A01:DWORD
.data
A01 DWORD 2
```

- Змінні з **файловою** та глобальною область видимості розміщуються в сегменті даних, створюються на початку роботи програми і знищуються на її закінченню.

- **Локальні, оголошення:**
- **LOCAL name Var [[count]]:type,…**
- Name – назва змінної
- Count – кількість елементів (необов'язково)
- Type – тип змінних
- **Приклад:**
- LOCAL L:DWORD, R:DWORD, SL:BYTE
- LOCAL array[20]:BYTE

Процедури

- **Процедура (функція)** – це завершений фрагмент коду для реалізації певного алгоритму, який може використовувати свої локальні змінні, та повторно викликатися в програмі.
- Процедура вміщується у виконавчий модуль один раз і в разі її виклику керування передається на її початок. В Assembler процедура повинна обов'язково закінчуватись командою **RET**. За замовчування процедури мають файлову область видимості.
- **Процедура може розміщуватись:**
- У файлі вихідного модуля з якого вона викликається (на початок програми, в середині програми, в кінці програми, але не за межами мітки). У всіх випадках розміщення процедура у файлі виконавчого модуля необхідно унеможливити на санкціонованому передачі управління у процедуру jmp;
- У іншому файлі. В цьому випадку процедура компілюється до об'єктного файлу або до файлу бібліотеки, і під'єднюються до виконавчого модуля. Так як процедури мають за замовчування файлову область видимості , їх необхідно в головній програмі оголосити за допомогою дерективи PROTO.
- Приклад:
- WinMain **PROTO** :DWORD,:DWORD,:DWORD,:DWORD

- **Передача аргументів в процедуру можливо трьома способами:**
  - Через стек
  - Через регістри процесора
  - Через спільну область пам’яті
-
- **Повернення результату з процедури виконується двома способами:**
- Через регістри
- Через спільну область пам’яті
- В Assembler за замовчуванням значення, що повертається з процедури **вміщується в EAX**

- **Правила виклику і передачі аргументу:**

- **stdcall** - правило передачі аргумента, яке використовується по замовчуванню в Win32. згідно даного правила: перший аргумент вміщується в стек останній, відповідно останній – перший. Процедура сама не вирівнює стек. Недолік: у випадку коли кількість аргументів велика, процедура сама не знає як вирівнювати їх.
- **C** - правило передачі по умові c - «перший - останній». Стек вирівнює процедура.
- **PASCAL** - перший – перший; останній – останній. Процедура вирівнює стек
- **Fastcall** - перші три аргумента передаються через регістри процесора , решта аргументів через стек. Процедура вирівнює стек

Спрощений синтаксис оголошення процедур (MASM32, Win32).

- **NameProc PROC** [param01:TYPE, param02:TYPE …]
- [USES reglist]
- [LOCAL varlist]
- ;код процедури
  RET
- **NameProc ENDP**
- Обовязковими елементами є:
- 1. Імя процедури та директива **PROC**.
- 2. Команда повернення з процедури **RET**.
- 3. Імя процедури та директива її завершення **ENDP**
- Додаткові :
- 1. **param01:TYPE** – список аргументів процедури.
- 2. **USES** список регістрів процесора значення яких повинні бути збережені на початку роботи процедури та відновлені по її закінченню
- 3. **LOCAL** -оголошення локальних змінних.

Приклад процедури

```
.code
SayStr PROC sOut:DWORD
    LOCAL L   : DWORD
    LOCAL R   : DWORD
    LOCAL sLen: DWORD
    LOCAL oHandle: DWORD
;   LOCAL     array[20]:BYTE
    INVOKE lstrlenA, sOut
    MOV sLen,EAX
 INVOKE GetStdHandle, STD_OUTPUT_HANDLE
    MOV oHandle,EAX
INVOKE WriteConsole, oHandle, sOut,sLen, addr L,addr R
      MOV EAX,L
      RET
SayStr ENDP
START: ; Наша програма
```

РЕЖИМИ АДРЕСАЦІЇ

- Під режимом адресації розуміють спосіб представлення зсуву адреси операнда.

- Існує два види адресації:
- **Пряма адресація**
- **Непряма адресація.**

Пряма адресація

- ***Абсолютна пряма адресація. (а*дресні**, *переміщувані операнди*). У цьому випадку ефективна адреса є частиною машинної команди яка формується зі значення адресу чи зсуву в команді.
- MOV ax, word ptr [0000] ;записати слово з адреси
- ;ds:0000 у регістр ax
- Як правило використовуються комірки памяті з символічними іменами. Під час трансляції ассемблер обчислює і підставляє значення зсувів цих імен у формовану їм машинну команду в поле зсуву команди.
-
- ***Відносна пряма адресація***. *(Безпосередні операнди*) Використовується для команд умовних переходів, для вказівки відносної адреси переходу.
- **JC ML ;**ПЕРЕХІД НА МІТКУ ML. ЯКЩО ПРАПОР CF = 1
- **MOV AL,2**
- **ML: …**
- Ассемблер обчислює зсув цієї мітки щодо наступної команди (у нашому випадку це mov al ,2) і підставляє його у формовану машинну команду jc.

Непряма адресація.

- в самій команді може знаходитися лише частина ефективної адреси, а інші його компоненти знаходяться в регістрах чи додаткових операндах.
- **Непряма базова (регістрова) адресація**
- ефективна адреса операнда може знаходитися в довільному з регістрів загального призначення, крім sp/esp і bp/ebp. **Синтаксично вирізняється вставкою імені регістра в квадратні дужки [ ].** Приклад, команда
- **mov ax,[ecx]**
- поміщає в регістр **ax** вміст слова за адресою із сегмента даних зі зсувом, що зберігається в регістрі **ecx**.
- Використовується при роботі з масивами.
- **Непряма базова (регістрова) адресація зі зсувом**
- Призначена для доступу до даних з відомим зсувом щодо деякої базової адреси. Використовують для доступу до елементів структур даних, коли зсув елементів відомо заздалегідь, на стадії розробки програми, а базовий адрес структури необхідно обчислюватися динамічно, на стадії виконання програми.
- Приклад:
- mov ax, [edx+3]
- пересилає в регістр **AX** слова з області пам'яті за адресою: значення edx + 3

- **Непряма індексна адресація зі зсувом**
- Для формування ефективної адреси використовується один з регістрів загального призначення з його масштабуванням. Використовується при роботі з масивами. Приклад:
- *MOV AX,MAS[ESI*2]; AX = MAS+(ESI)*2*
- Тільки якщо розмір елементів масиву складає 1, 2, 4 або 8 байт.
- **Непряма базова індексна адресація**
- ефективна адреса формується як сума вмісту двох регістрів загального призначення: базового й індексного.
- Наприклад:
- *MOV EAX,[ESI][EDX]*
- У даному прикладі ефективна адреса другого операнду формується з двох компонентів, (esi)+(edx).
- **Непряма базова індексна адресація зі зсувом**
- Ефективна адреса формується як сума трьох складових: вмісту базового регістра, вмісту індексного регістра і значення полю чи зсуву в команді.
- Приклад:
- *MOV EAX,[ESI+5][EDX] ; EAX=(ESI)+5+(EDX)*
- ADD EAX,ARRAY[ESI][EBX] ; EAX=ARRAY+(ESI)+(EBX)

Математичний сопроцесор

- Всю роботу з дійсними числами покладено на співпроцесор. Він має свої регістри **ST0-ST7** (стек чисел з плаваючою комою), свій порядок роботи з ними, свої команди. Є три режими точності для дійсних чисел:
- 1. одинарна точність (32 біти - 4 байти);
- 2. подвійна точність (64 біти - 8 байтів);
- 3. підвищена точність (80 бітів - 10 байтів).
- Найпоширенішою є **подвійна точність (64 біти - 8 байтів)**. Для оголошення таких змінних використовують такий формат:
- X DQ 0
- Y DQ ? ; тут dq = define quarterword (означити почетвірне слово).

Представлення дійсних чисел

```
Одинарна точність
1 бит 8 бит     23 бита
---T-------T--------------------¬
¦Зн¦Порядок¦      Мантиса       ¦
L--+-------+---------------------
Двійна точність
1 бит 11 бит          52 бита
---T---------T---------------------------------¬
¦Зн¦ Порядок ¦             Мантиса              ¦
L--+---------+---------------------------------
 Розширена точність
1 бит 15 бит                    64 бита
---T-------------T-----------------------------------¬
¦Зн¦   Порядок   ¦                Мантиса              ¦
L--+-------------+-----------------------------------
```

- **Класифікація команд:**
- Усі команди співпроцесора можна розділити на кілька груп:
  - команди пересилання даних;
  - арифметичні команди;
  - команди порівнянь чисел;
  - трансцендентні команди;
  - керуючі команди.

Конвертація дійсного числа в рядок і навпаки

функція	Дія
StrToFloat, addr szBuffer, addr X	перетворює рядок szBuffer в число X
FloatToStr, X, addr szBuffer	перетворює число X в рядок szBuffer (небагато знаків після коми)
FloatToStr2, X, addr szBuffer	перетворює число X в рядок szBuffer (розширена точність)

Основні команди

команда	пояснення
FLD X FSTP Y ...	завантажити в “стек дійсних чисел” змінну X (ST0 := X, а всіх інших – “посунути” вниз) вибрати зі “стеку дійсних чисел” число і помістити його в змінну Y (Y := ST0) ...
FADD FSUB FMUL FDIV ...	(ST0 := ST1 ST0) + (ST0 := ST1 - ST0) - (ST0 := ST1 * ST0) * (ST0 := ST1 / ST0) / ...
FSIN FCOS FPATAN FSQRT FABS FCHS FYL2X ...	ST0 := sin(ST0) ST0 := cos(ST0) ST0 := tg(ST0) ST0 := VST0 ST0 := |ST0| ST0 := —ST0 ST0 := y ■ $\log_2$ x (ST0 • $\log_2$ ST1) ...
FLDPI FLD1 FLDZ FLDL2E FLDLN2 ...	завантажити на вершину стеку число π завантажити на вершину стеку число 1 завантажити на вершину стеку число 0 завантажити на вершину стеку число $\log_2$ e завантажити на вершину стеку число ln2 ...

Команди порівняння

FCOM X	Порівняння ST0 > X
FICOM X	Цілочисельне порівняння ST0 > X
FCOMP X	Порівняння ST0 > X і витягування зі стеку
FICOMP	Цілочисельне порівняння ST0 > X і витягування зі стеку
FCOMPP	Порівняння ST0 > X і витягування зі стеку
FTST	Порівняння операнду з 0
FXAM	Анализ операнда