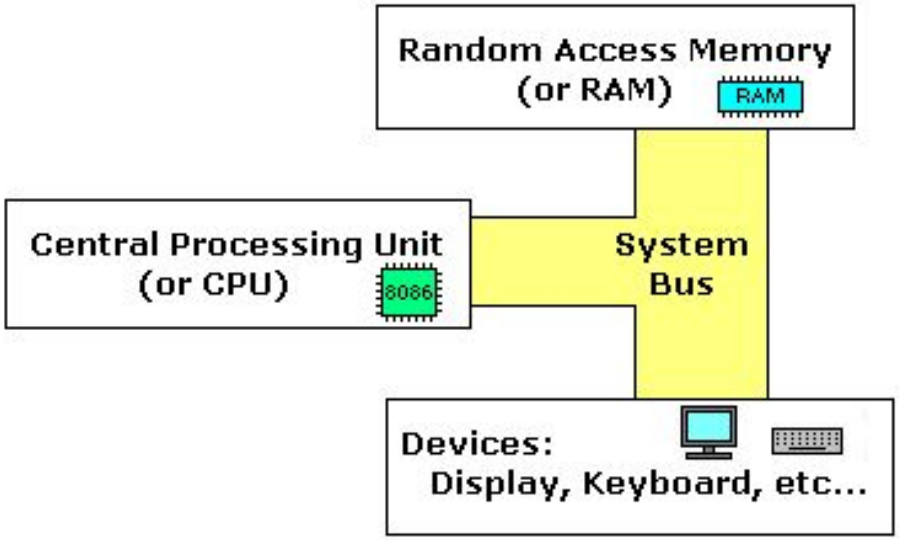
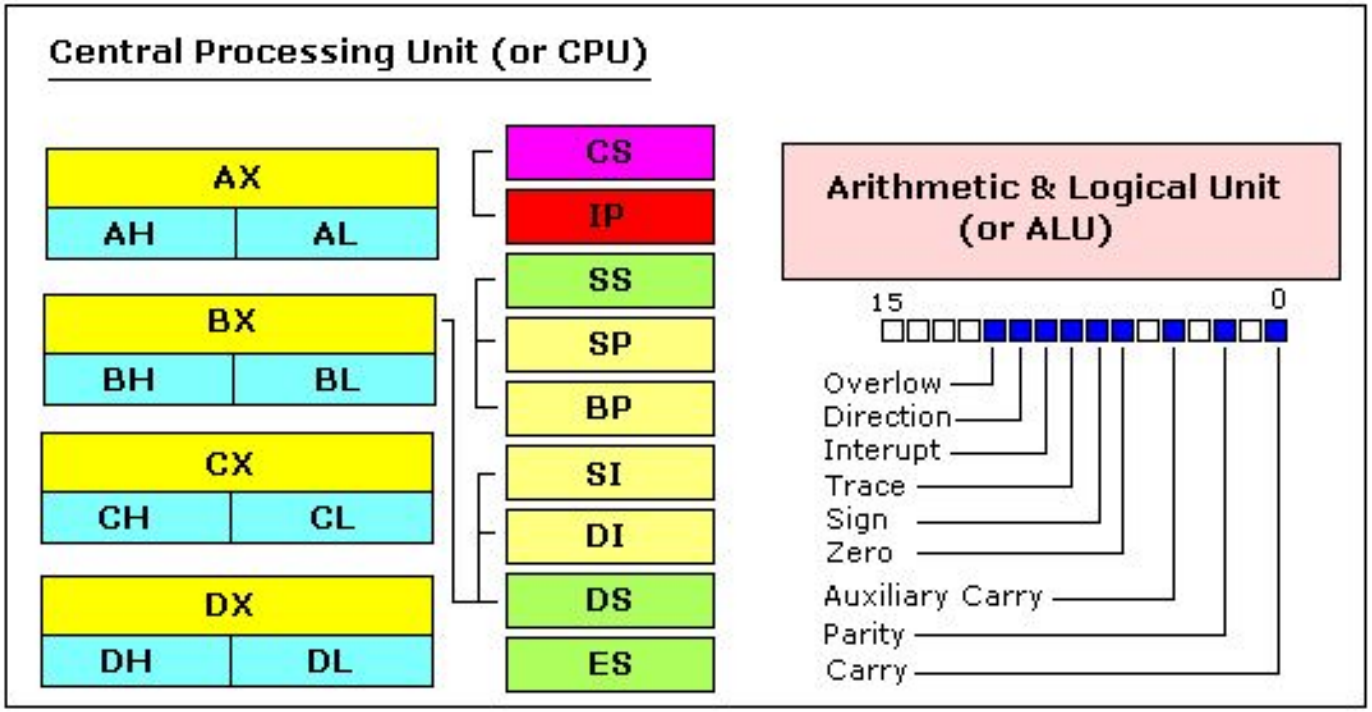
Що таке мова Асемблера?

Мова Асемблера - це мова програмування низького рівня. Для початку ми повинні ознайомитися з загальною структурою комп'ютера, щоб надалі розуміти, про що йде мова. Спрощена модель комп'ютера:



System bus - системная шина (окрашена желтым цветом) соединяет различные компоненты компьютера.
CPU - центральный процессор - это сердце компьютера. Большинство вычислений происходит в **CPU**.
RAM - оперативная память (ОЗУ). В оперативную память загружаются программы для выполнения.

Будова процесора



РЕГІСТРИ ЗАГАЛЬНОГО ПРИЗНАЧЕННЯ

Процесор 8086 має 8 регістрів загального призначення, кожен регістр має ім'я:

- **AX** - регістр-акумулятор (розділений на два регістра: **AH** і **AL**).
- **BX** - регістр базового адреса (розділений на **BH / BL**).
- **CX** - регістр-лічильник (розділяється на **CH / CL**).
- **DX** - регістр данных (розділяється на **DH / DL**).

- **SI** - регістр - індекс джерела.
- **DI** - регістр - індекс призначення.
- **BP** – вказівник бази.
- **SP** - вказівник стека.

Незважаючи на ім'я регістра, програміст сам визначає, для яких цілей використовувати регістри загального призначення. Основне призначення регістра - зберігання числа (змінної). Розрядність вищеописаних регістрів 16 біт, тобто, наприклад, 0011000000111001b (в двійковій системі) або 12345 в десятковій системі.

4 регістра загального призначення (AX, BX, CX, DX) розділені на дві частини. До кожної частини можна звертатися як до окремого регістру. Наприклад, якщо AX = **0011000000111001b**, то AH = **00110000b**, а AL = **00111001b**. Старший байт позначається буквою "H", а молодший байт - буквою "L".

Оскільки регістри розташовані всередині процесора, то працюють вони значно швидше, ніж пам'ять. Звернення до пам'яті вимагає використання системної шини, а на це йде більше часу. Звернення до регістрів взагалі не забирає час. Тому ви повинні намагатися зберігати змінні в регістрах. Кількість регістрів дуже невелике і багато регістри мають спеціальне призначення, яке не дозволяє використовувати їх для зберігання змінних, але все ж вони є найкращим місцем для запису тимчасових даних і обчислень.

СЕГМЕНТНІ РЕГІСТРИ

- CS - вказує на сегмент, що містить початкову адресу поточної програми.
- DS - зазвичай вказує на початковий адресу сегмента даних (змінних).
- ES - додатковий сегментний регістр.
- SS - містить початкову адресу сегмента стека.

Хоча в сегментних регістрах можна зберігати будь-які дані, робити це нерозумно. Сегментні регістри мають строго певне призначення - забезпечення доступу до блоків пам'яті.

Сегментні регістри працюють спільно з регістрами загального призначення для доступу до пам'яті. Наприклад, якщо ми хочемо отримати доступ до пам'яті з фізичною адресою 12345h (в шістнадцятковій системі числення), ми повинні встановити DS = 1230h і SI = 0045h. І це правильно, тому що таким чином ми можемо отримати доступ до пам'яті, фізичну адресу якої більше, ніж значення, яке може помістится в одиночному регістрі.

Процесор обчислює фізичну адресу, множачи значення сегментного регістра на 10h і додаючи до отриманого результату значення регістра загального призначення (1230h * 10h + 45h = 12345h):

```
 12300
+  0045
-------
 12345
```

Адреса, сформована за допомогою двох регістрів, називається реальною адресою.
За замовчуванням регістри BX, SI і DI працюють з сегментним регістром DS; регістри BP і SP працюють з SS.
Інші регістри загального призначення не можуть формувати реальну адресу!
Також, хоча BX може формувати реальну адресу, BH і BL не можуть!

РЕГІСТРИ СПЕЦІАЛЬНОГО ПРИЗНАЧЕНИЯ

- **IP** - командний покажчик.
- **Флаговий регистр** - визначає поточний стан процесора.

Регістр **IP** завжди працює спільно з сегментним регістром CS і вказує на виконувану в даний момент команду.
Флаговий регістр автоматично змінюється процесором після математичних операцій. Він дозволяє визначати тип результату і передавати управління іншим ділянкам програми.
Взагалі ви не можете безпосередньо звертатися до цих регістрів.

Доступ до пам’яті

Для доступу до пам'яті можна використовувати наступні чотири регістри: BX, SI, DI, BP.
Комбінуючи ці регістри всередині квадратних дужок [ ], ми можемо отримати доступ до різних розташувань в пам'яті. Можливі наступні комбінації (режими адресації):

[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (тільки змінна-зміщення) [BX]	[BX + SI] + d8 [BX + DI] + d8 [BP + SI] + d8 [BP + DI] + d8
[SI] + d8 [DI] + d8 [BP] + d8 [BX] + d8	[BX + SI] + d16 [BX + DI] + d16 [BP + SI] + d16 [BP + DI] + d16	[SI] + d16 [DI] + d16 [BP] + d16 [BX] + d16

d8 - позначення для 8-ми бітової підстановки.
d16 - позначення для 16-ми бітової підстановки.

Підстановка може бути безпосереднім значенням або зміщенням змінної, або навіть і тим і іншим. Вона перетворюється компілятором в одиночне безпосереднє значення.
Підстановка може бути як всередині так і поза квадратних дужок ([]), компілятор генерує однаковий машинний код в обох випадках.
Підстановка - це величина зі знаком, тому вона може бути як позитивною, так і негативною.

Зазвичай компілятор розрізняє d8 і d16 і генерує необхідний машинний код.
Наприклад, уявімо що **DS = 100**, **BX = 30**, **SI = 70**.
Наступний спосіб адресації: **[BX + SI] + 25**
обчислюється процесором для цієї фізичної адреси: **100 * 16 + 30 + 70 + 25 = 1725**.

За замовчуванням сегментний регістр DS використовується для всіх способів адресації, крім способу, який використовується з регістром BP. В останньому випадку використовують сегментний регістр SS.

Легко запам'ятати всі можливі комбінації за допомогою таблиці:

BX	SI	+ disp
BP	DI	

Вы можете формировать все имеющие силу комбинации, взяв по одному пункту из каждого столбца, либо пропустить какой-либо столбец и ничего из него не взять. Как вы можете видеть, **BX** и **BP** никогда не идут вместе. **SI** и **DI** также не могут быть вместе. Здесь приведен пример имеющего силу способа адресации: **[BX+5]**.

Ви можете формувати всі комбінації, взявши по одному пункту з кожного стовпчика, або пропустити будь-який стовпець і нічого з нього не взяти. Як видно, BX і BP ніколи не йдуть разом. SI і DI також не можуть бути разом. Приклад способу адресації: [BX + 5].

Значення в сегментному регістрі (CS, DS, SS, ES) називається "segment (сегмент)", а значення в регістрі загального призначення (BX, SI, DI, BP) називається "offset (зміщення)". Якщо DS містить значення 1234h, а SI містить значення 7890h, то це можна записати як **1234:7890**. Фізична адреса буде такою: 1234h * 10h + 7890h = 19BD0h.

Щоб вказати компілятору тип даних, повинні використовуватися відповідні префікси:

BYTE PTR - для байта.
WORD PTR - для слова (два байта).
Наприклад:

```
BYTE PTR [BX]     ; доступ до байту.
   або
WORD PTR [BX]     ; доступ до слова.
```

Emu8086 підтримує короткі префікси:
b. - для **BYTE PTR**
w. - для **WORD PTR**
іноді компілятор може обчислити тип даних автоматично, але ви не можете і не повинні покладатися на це, якщо один з операндів є безпосереднім значенням.

Команда MOV

- Копіює другий операнд (джерело) в перший операнд (приймач).
- Операнд-джерело може бути безпосереднім значенням, регістром загального призначення або місцем розташування пам'яті.
- Регістр-приймач може бути регістром загального призначення або місцем розташування пам'яті.
- Обидва операнди повинні мати однаковий розмір байта або слова.

Ці типи операндів підтримуються:

MOV регістр, пам’ять
MOV пам’ять, регістр
MOV регістр, регістр
MOV пам’ять, безпосереднє значення (число)
MOV регістр, безпосереднє значення
регістр: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

пам’ять: [BX], [BX+SI+7], змінна, і т.д...

безпосереднє значення: 5, -24, 3Fh, 10001101b, і т.д...

Для сегментних регістрів підтримуються тільки ці типи **MOV**:

MOV сегментний регістр, пам’ять
MOV пам’ять, сегментний регістр
MOV регістр, сегментний регістр
MOV сегментний регістр, регістр
сегментний регістр: DS, ES, SS, і тільки як другий операнд: CS.

регістр: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

пам’ять: [BX], [BX+SI+7], змінна і т.д...

Команда **MOV** не меже використовуватися для установки значень регістрів **CS** і **IP**.

Коротка програма, яка демонструє використання команди **MOV**:

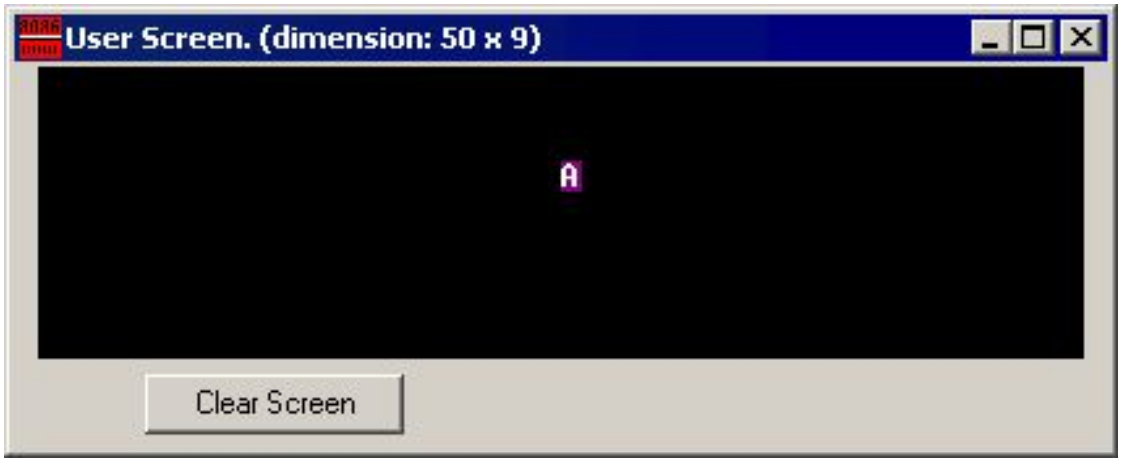
```
#MAKE_COM#          ; команда компілятора для створення COM-файла.
ORG 100h            ; директива, необхідна для COM-програми.
MOV AX, 0B800h      ; встановити AX в шістнадцядкове значення B800h.
MOV DS, AX          ; копіювати значення із AX в DS.
MOV CL, 'A'         ; установить в CL ASCII-код символу 'A', тобто 41h.
MOV CH, 01011111b   ; встановити CH в двійкове значення.
MOV BX, 15Eh        ; встановити BX в 15Eh.
MOV [BX], CX        ; копіювати вміст із CX в пам’ять з адресою B800:015E
RET                 ; повернутись в операційну систему.
```

Ви можете скопіювати і вставити вищеописану програму в редактор коду Emu8086, і натиснути кнопку [Compile and Emulate] (або натиснути клавішу F5 на клавіатурі).

Вікно емулятора має відкритися із завантаженою програмою. Натисніть кнопку [Single Step] (покроковий режим) і спостерігайте за вмістом регістрів.

Крапка з комою (";") використовується для коментарів. Всі символи, які слідують за ";", ігноруються компілятором.

Ви повинні побачити щось подібне, коли програма закінчить свою роботу:



Фактично, вищеописана програма записує дані безпосередньо в відеопам'ять, так що MOV - це дуже потужна інструкція.

Змінні

Змінні зберігаються в пам'яті за певними адресами. Програмісту простіше мати справу з іменами змінних, ніж з адресами в пам'яті. Наприклад, змінна з ім'ям "**var1**" буде зрозуміліша в коді програми, ніж адреса 5A73: 235B, особливо коли кількість змінних велике.

Наш компілятор підтримує два типи змінних: BYTE і WORD.

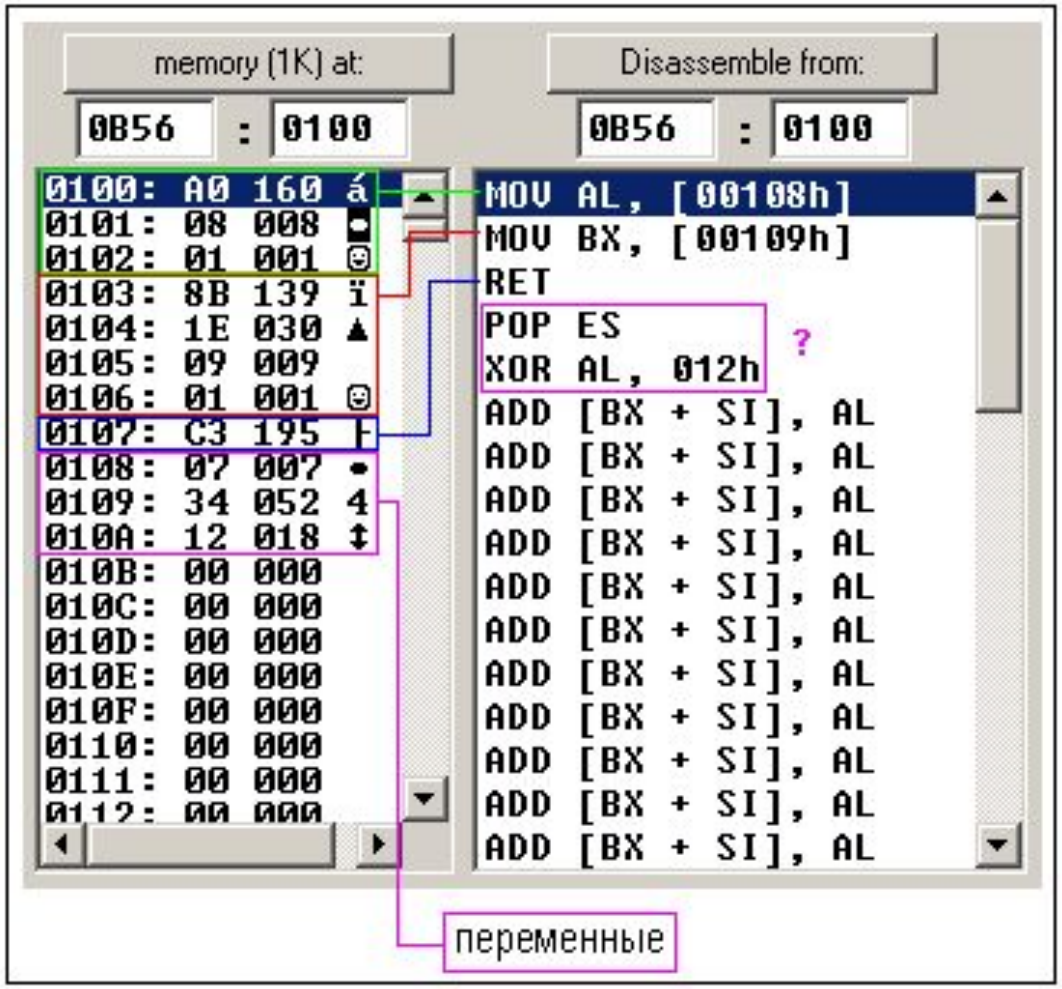
Синтаксис для оголошення змінних:
ім'я DB значення
ім'я DW значення
DB - Define Byte - визначає байт.
DW - Define Word - визначає слово.
ім'я - може бути будь-якою комбінацією букв або цифр, не повинно починатися з літери. Можна оголошувати безіменні змінні, які мають адресу, але не мають імені.
значення - може бути будь-який числовий величиною, яка представлена в будь-якої з підтримуваних систем числення (шістнадцятковій, двійковій або десятковій). Значення може також бути символом "?" для НЕ ініціалізованих змінних.

Команда **MOV** також використовується для копіювання значення з джерела в приймач.

Давайте розглянемо ще один приклад з командою MOV:

```
#MAKE_COM#
ORG 100h
MOV AL, var1
MOV BX, var2
RET    ; завершення
програми.
VAR1 DB 7
var2 DW 1234h
```

Якщо вставити код в редактор коду Emu8086 і натиснути клавішу F5, щоб відкомпілювати і завантажити цей код в емулятор. Побачимо приблизно таку картину:

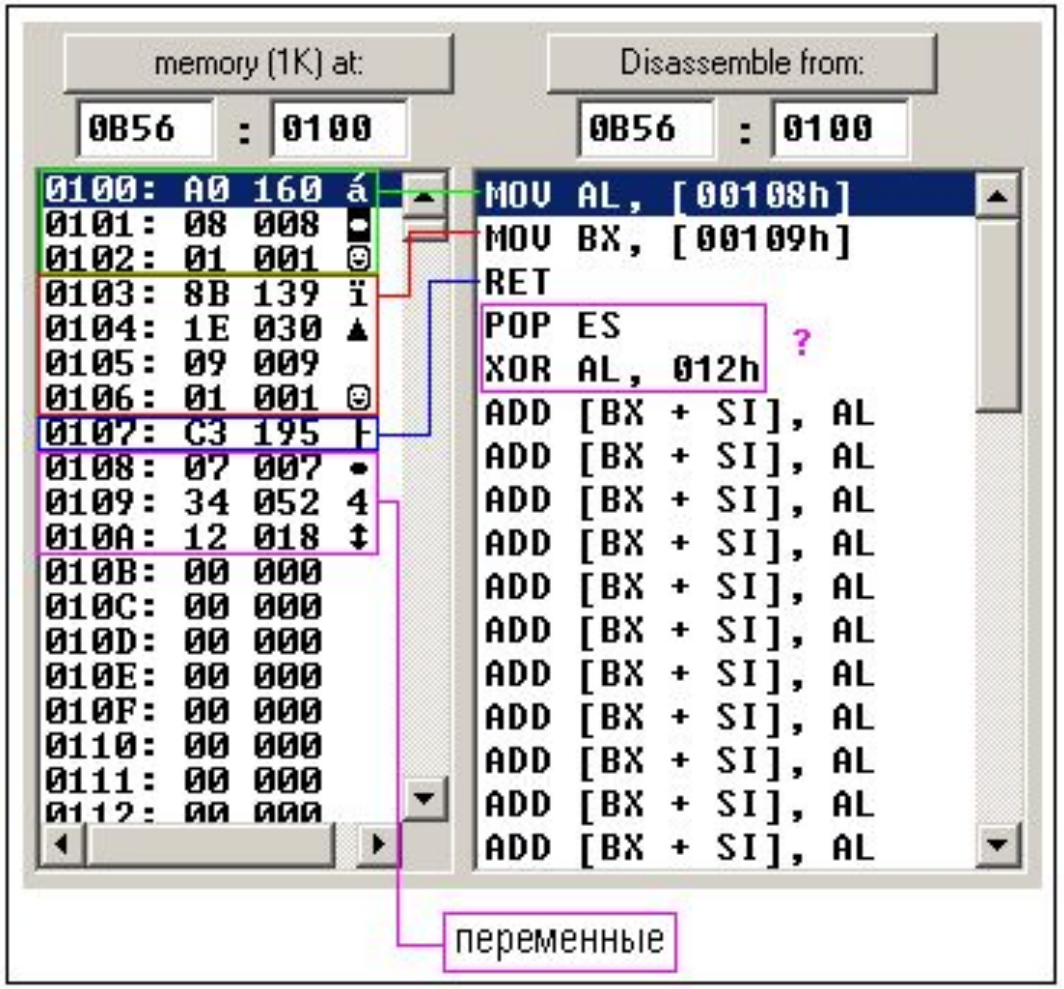


На малюнку ви можете помітити команди, схожі на ті, що використовуються в нашому прикладі. Тільки змінні замінені фактичними місцями розташування в пам'яті. Коли компілятор створює машинний код, він автоматично замінює імена всіх змінних їх зміщеннями. За замовчуванням сегмент завантажений в регістр DS (в COM-файлах значення регістра DS встановлюється таким же, що і значення в регістрі CS - сегменті коду).

У таблиці пам'яті (memory) перший стовпець - це зміщення, другий стовпець - це шістнадцяткове значення, третій стовпець - десяткове значення, а останній рядок - це символ ASCII, який відповідає цьому числу.

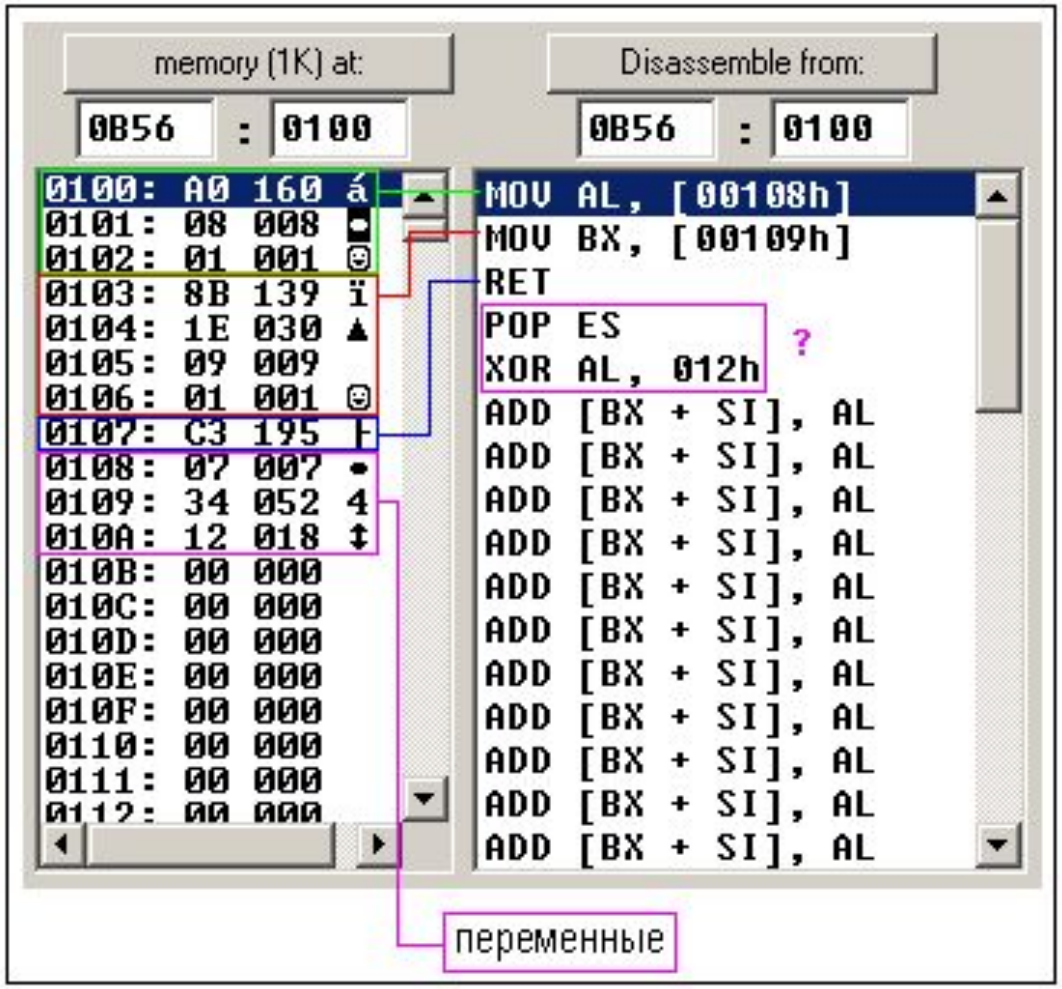
Компілятор не чутливий до регістру, тому "VAR1" і "var1" - це одне й те саме.

Зсув змінної VAR1 - це 0108h, а повна адреса - 0B56: 0108.



Зсув змінної var2 - це 0109h, а повна адреса - 0B56: 0109. Ця змінна має тип WORD, тому займає 2 байта. Прийнято молодший байт записувати за меншою адресою, тому 34h розміщується перед 12h.

Ви можете побачити деякі інші інструкції після команди RET. Це трапляється тому, що дизассемблер не знає, де починаються дані. Він тільки обробляє значення в пам'яті і розуміє їх як такі, які є в інструкції процесора 8086.



Можна навіть написати програму, використовуючи тільки директиву DB:

```
#MAKE_COM#
ORG 100h
DB 0A0h
DB 08h
DB 01h
DB 8Bh
DB 1Eh
DB 09h
DB 01h
DB 0C3h
DB 7
DB 34h
DB 12h
```

Якщо скопіювати вищенаведений код в редактор коду Emu8086 і відкомпілювати і завантажити цей код в емулятор. Отримаємо той же самий код і той же самий результат роботи програми!

Компілятор тільки перетворює вихідний код програми в набір байтів. Цей набір байтів називається **машинним кодом**. Процесор обробляє машинний код і виконує його.

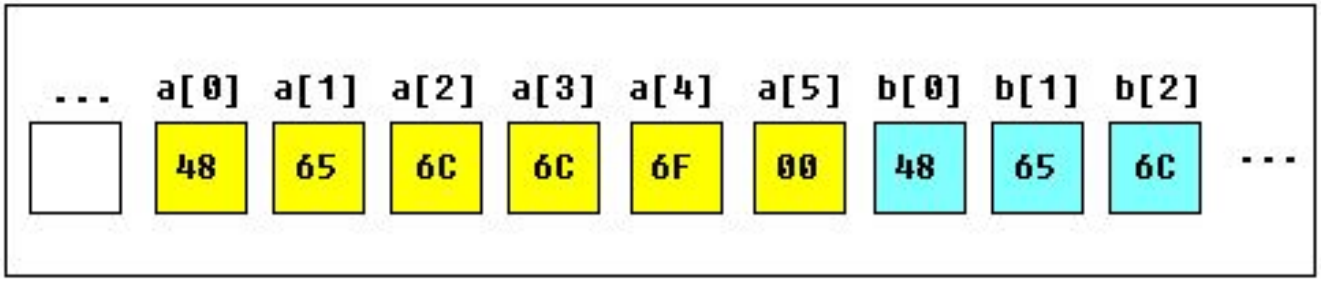
ORG 100h - це директива компілятора (вона вказує компілятору як обробляти вихідний код). Ця директива дуже важлива при роботі зі змінними. Вона вказує компілятору, який виконуваний файл буде завантажуватися в **зміщення (offset)** 100h (256 байтів), так що компілятор повинен обчислити правильну адресу для всіх змінних, коли він розміщує імена змінних з їх **зміщеннями**. Директиви ніколи не перетворюються в який-небудь реальний машинний **код**. Чому виконуваний файл завантажується по **зсуву 100h**? Операційна система зберігає деякі дані про програму в перших 256 байтах, починаючи від **CS** (сегмента коду), такі як параметри командного рядка і т.д. Все це справедливо тільки для **COM**-файлів, файли **EXE** завантажуються зі зміщення **0000** і зазвичай використовують спеціальний сегмент для змінних.

Масиви

Масив можна розглядати як ланцюжок змінних. Текстовий рядок - це приклад масиву байтів, в якому кожен символ представлений значенням ASCII-коду (0..255).

Ось деякі приклади визначення масивів:
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0

b - це точна копія масиву **a** - коли компілятор бачить рядок, взятий в лапки, він автоматично перетворює її в набір байтів. Ця таблиця показує ділянку пам'яті, де ці масиви оголошені:



Можна отримати значення будь-якого елементу масиву, використовуючи квадратні дужки, наприклад:

MOV AL, a[3]

Можна також використовувати будь-якої з регістрів **BX, SI, DI, BP**, наприклад:

```
MOV SI, 3
MOV AL, a[SI]
```

Якщо необхідно оголосити великий масив, можна використовувати оператор **DUP**.
Синтаксис для **DUP**:
кількість DUP (значення)
кількість - кількість дублікатів (будь-яка константа).
значення - вираз, який буде дублюватись оператором DUP.
Наприклад:

```
c DB 5 DUP(9)
```

а це альтернативний спосіб:

```
c DB 9, 9, 9, 9, 9
```

ще один приклад:

```
d DB 5 DUP(1, 2)
```

а це альтернативний спосіб присвоєння:

```
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
```

Звичайно, ви можете використовувати **DW** замість **DB**, якщо потрібно зберігати числа понад 255, або менше -128. **DW** не може бути використаний для оголошення рядків!

Оператор **DUP** не може містити більше 1020 знаків в якості операнда! (В останньому прикладі 13 знаків). Якщо вам необхідно оголосити дуже великий масив, розділіть його на два рядки (ви отримаєте один великий масив в пам'яті).

Отримання адреси змінної

Є така команда **LEA** (Load Effective Address) і альтернативний оператор **OFFSET**. Як **OFFSET** так і **LEA** можуть бути використані для одержання зсуву адреси змінної.

LEA потужніша, тому що вона також дозволяє вам отримати адресу індексованих змінних. Отримання адреси змінної може бути дуже корисно в різних ситуаціях, наприклад, якщо вам необхідно помістити параметр в процедуру.

Приклад:

```
ORG 100h
MOV    AL, VAR1                ; перевірити значення VAR1, помістити його в AL.
LEA    BX, VAR1                ; записати адрес змінної VAR1 в BX.
MOV    BYTE PTR [BX], 44h      ; змінити вміст змінної VAR1.
MOV    AL, VAR1                ; перевірити значення VAR1, помістити її в AL.
RET
VAR1   DB  22h
END
```

Другий приклад, який використовує **OFFSET** замість **LEA**:

```
ORG 100h
MOV   AL, VAR1              ; перевірити значення VAR1, помістити його в AL.
MOV   BX, OFFSET VAR1       ; записати адрес змінної VAR1 в BX.
MOV   BYTE PTR [BX], 44h    ; змінити вміст змінної VAR1.
MOV   AL, VAR1              ; перевірити значення VAR1, помістити її в AL.
RET
VAR1  DB  22h
END
```

Обидва приклади функціонально ідентичні.

Ці рядки:
LEA BX, VAR1
MOV BX, OFFSET VAR1
навіть компілюються в однаковий машинний код: MOV BX, **num**
num - це 16-бітове значення зміщення змінної.

Врахуйте, що тільки ці регістри можуть використовуватися всередині квадратних дужок (як покажчики пам'яті):
BX, SI, DI, BP!

Константи

Константи подібні змінним, але вони існують до того, як ваша програма відкомпільована. Після визначення константи її значення не може бути змінено. Для визначення константи використовується директива **EQU**:
ім’я EQU < будь-яки вираз >

Наприклад:

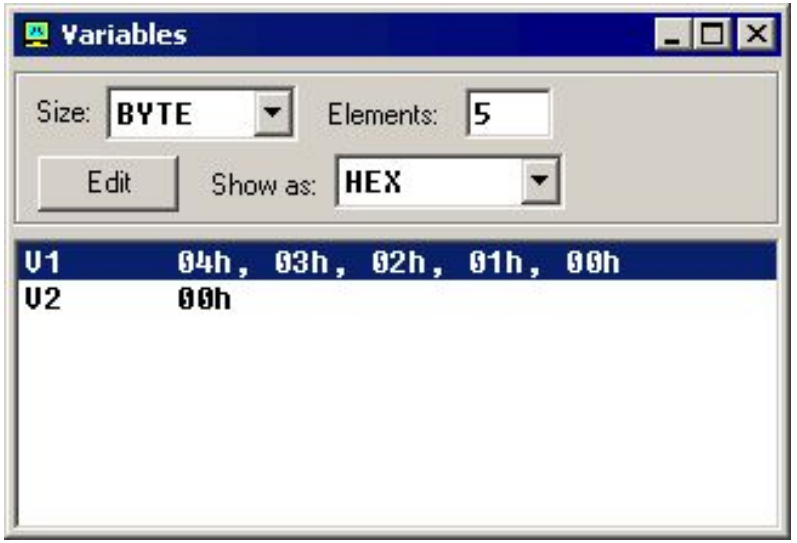
```
k EQU 5

MOV AX, k
```

Цей приклад функціонально ідентичний коду:

```
MOV AX, 5
```

Можна спостерігати змінні під час виконання програми, якщо вибрати пункт "Variables" в меню "View" емулятора.



Щоб спостерігати масиви, потрібна клацнути по змінній і встановити властивість Elements - розмір масиву. У Ассемблері немає строгих типів даних, тому будь-які змінні можуть бути представлені як масив.

Змінна може бути переглянута в будь-якій числовій системі:

- **HEX** - шістнадцяткова (основа 16).
- **BIN** – двійкова (основа 2).
- **OCT** - вісімкова (основа 8).
- **SIGNED** – десяткова із знаком (основа 10).
- **UNSIGNED** - десяткова без знаку (основа 10).
- **CHAR** - коди ASCII-символів (всього 256 символів, деякі символи невидимі).

Можна редагувати змінні, коли програма виконується, просто клацнувши двічі по змінної або вибрати її і натиснути кнопку **Edit**.

Можна вводити числа в будь-якій системі, шістнадцяткові цифри повинні мати суфікс "**h**", двійкові - суфікс "**b**", восмерічние - суфікс "**o**", десяткові цифри не вимагають суфікса. Рядок може бути введена в такий спосіб:

'hello world', 0

(Цей рядок закінчується нулем).

Масив може бути введений в такий спосіб:

1, 2, 3, 4, 5

(Масив може бути масивом байтів і слів, це залежить від того, чи вибраний **BYTE** або **WORD** для введеної змінної).

Вирази перетворюються автоматично, наприклад:
якщо введено цей вираз:
5 + 2
воно буде перетворено в 7 і т.д ...

Переривання

Переривання можна розглядати як номер функції. Ці функції роблять програмування більш легким - замість написання коду шляхом друкування символів ви можете просто викликати переривання і воно все зробить за вас. Існують також функції переривань, які працюють з дисками і іншим "залізом". Ми називаємо такі функції **програмними перериваннями**.

Переривання можуть бути також викликані різними пристроями. Такі переривання називаються **апаратними перериваннями**. Але зараз нас цікавлять тільки **програмні переривання**.

Щоб виконати **програмне переривання**, використовують команду **INT**, яка має дуже простий синтаксис:

INT значення

Де **значення** може бути числом в діапазоні від 0 до 255 (або від 0 до 0FFh), зазвичай ми будемо використовувати шістнадцяткові числа. Ви можете подумати, що є тільки 256 функцій, але це не так. Кожне переривання може мати підфункції. Щоб визначити підфункцію, в регістр **AH** потрібно записати її номер перед викликом переривання. Кожне переривання може мати до 256 підфункцій (таким чином ми отримуємо 256 * 256 = 65536 функцій). В основному використовується регістр **AH**, але іноді можуть використовуватися і інші регістри. Зазвичай інші регістри використовуються для запису параметрів та даних підфункції.

Наступний приклад використовує переривання INT 10h і підфункцію 0Eh, щоб надрукувати повідомлення "Hello!". Ця функція виводить символ на екран, переміщаючи курсор і прокручуючи екран в разі потреби.

```
#MAKE_COM#        ; інструкція компілятора для створення COM-файлу.
ORG    100h
; Підфункція, яку ми використовуємо
; не змінює регістр AH після завершення,
; так що ми можемо використовувати його тільки один раз
MOV    AH, 0Eh    ; вибір підфункції.
; Підфункція INT 10h / 0Eh приймає
; в якості прикладу ASCII-код символу,
; який потрібно записувати в регістр AL.
MOV    AL, 'H'    ; ASCII-код: 72
INT    10h          ; надрукувати його!
MOV    AL, 'e'    ; ASCII-код: 101
INT    10h          ; надрукувати його!
MOV    AL, 'l'    ; ASCII-код: 108
INT    10h          ; надрукувати його!
MOV    AL, 'l'    ; ASCII-код: 108
INT    10h          ; надрукувати його!
MOV    AL, 'o'    ; ASCII-код: 111
INT    10h          ; надрукувати його!
MOV    AL, '!'    ; ASCII-код: 33
INT    10h          ; надрукувати його!
RET                ; вернуться в операційну систему.
```

Бібліотека загальних функцій - emu8086.inc

Щоб полегшити програмування, є кілька загальних функцій, які можна включати в програму. Щоб використовувати у програмі функції, визначені в іншому файлі, потрібно застосувати директиву **INCLUDE**, за якою слідує ім'я файлу. Компілятор автоматично знайде файл в тій же папці, де розміщений файл з вихідним кодом програми, а якщо там цього файлу не виявиться, то пошук буде продовжено у папці **Inc**.

Щоб використовувати будь-які функції в emu8086.inc, потрібно вписати наступний рядок у вихідному файлі:**include emu8086.inc**

emu8086.inc визначає наступні **макроси**:

- **PUTC char** - макрос з одним параметром, друкує ASCII-символ в поточній позиції курсору.
- **GOTOXY col, row** - макрос з двома параметрами, встановлює позицію курсора.
- **PRINT string** - макрос з одним параметром, друкує рядок.
- **PRINTN string** - макрос з одним параметром, друкує рядок. Це те ж саме, що і PRINT, але автоматично додається "переведення каретки" в кінці рядка (аналогічно процедурі Writeln в Паскалі).
- **CURSOROFF** - приховує текстовий курсор.
- **CURSORON** - показує текстовий курсор.

Щоб використовувати будь-якої з вищеописаних макросів, просто надрукуйте його ім'я в потрібному місці вашого коду і, якщо необхідно, параметри. наприклад:

```
include emu8086.inc
ORG    100h
PRINT 'Hello World!'
GOTOXY 10, 5
PUTC 65          ; 65 - це ASCII-код для букви 'A'
PUTC 'B‘
RET              ; повернутися в операційну систему.
END              ; директива для закінчення компіляції.
```

Коли компілятор обробляє ваш вихідний код, він шукає файл emu8086.inc для оголошених макросів і замінює ім'я макросу реальним кодом. Взагалі макроси - це відносно невеликі ділянки коду, часте використання макросів може зробити вашу програму (здійсненний файл) занадто великий (для оптимізації розміру краще використовувати процедури).

emu8086.inc також визначає наступні процедури:

- **PRINT_STRING** - процедура для друку рядка з нульовим закінченням з поточної позиції курсора. Отримує адреса рядка в регістрі **DS**: **SI**. Щоб використовувати цю процедуру, слід оголосити: **DEFINE_PRINT_STRING** перед директивою **END**.
- **PTHIS** - процедура для друку рядки з нульовим закінченням з поточної позиції курсора (як і PRINT_STRING), але отримує адресу з стека. Рядок з нульовим закінченням повинна бути визначена тільки після команди CALL. наприклад:

  CALL PTHIS
  db 'Hello World!', 0

  Щоб використовувати цю процедуру, слід оголосити: **DEFINE PTHIS** перед директивою **END**.
- **GET_STRING** - процедура для отримання рядка з нульовим закінченням від користувача. Прийнята рядок записана в буфер, адресу якого вказано в **DS**: **DI**, розмір буфера повинен бути в **DX**. Процедура завершує введення, якщо натиснута кнопка 'Enter'. Щоб використовувати цю процедуру, слід оголосити: **DEFINE_GET_STRING** перед директивою **END**.
- **CLEAR SCREEN** - процедура для очищення екрана (виконує повне прокручування екрана і встановлює курсор в його верхній частині) Щоб використовувати цю процедуру, слід оголосити: **DEFINE_CLEAR_SCREEN** перед директивою **END**.

- **SCAN_NUM** - процедура, яка отримує багатозначне число ІЗ ЗНАКОМ з клавіатури, і записує результат в регістр **CX**. Щоб використовувати цю процедуру, слід оголосити: **DEFINE_SCAN_NUM** перед директивою **END**.
- **PRINT_NUM** - процедура, яка друкує число зі знаком, яке знаходиться в регістрі **AX**. Щоб використовувати цю процедуру, слід оголосити: **DEFINE_PRINT_NUM** перед директивою **END**.
- **PRINT_NUM_UNS** - процедура, яка друкує число без знака з регістра **AX**. Щоб використовувати цю процедуру, слід оголосити: **DEFINE_PRINT_NUM_UNS** перед директивою **END**.

Щоб використовувати будь-яку з вищеописаних процедур, ви повинні спочатку оголосити функцію в нижній частині вашого файлу (але перед END !!), а потім використовувати команду CALL, за якою слідує ім'я процедури.

```
include 'emu8086.inc'
ORG    100h
LEA    SI, msg1                    ; попросить ввести число
CALL   print_string   ;
CALL   scan_num                    ; отримати число в CX.
MOV    AX, CX                      ; копіювати число в AX.
; надрукувати наступні рядки:
CALL   pthis
DB  13, 10, 'You have entered: ', 0
CALL   print_num                   ; надрукувати число з AX.
RET                                ; вернуться в операційну систему.
msg1   DB  'Enter the number: ', 0
DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS   ; потрібно для print_num.
DEFINE_PTHIS
END                                ; директива закінчення компіляції.
```

Спочатку компілятор обробляє оголошення (це просто звичайні макроси, які розглядаються до процедур). Коли компілятор отримує команду CALL, він замінює ім'я процедури адресою коду, де оголошена процедура. Коли виконається команда CALL, то управління буде передано процедурі. Це дуже зручно: навіть якщо ви викличте в вашому коді одну і ту ж процедуру 100 раз, то ви все-одно будете мати відносно невеликий розмір виконуваного файлу.