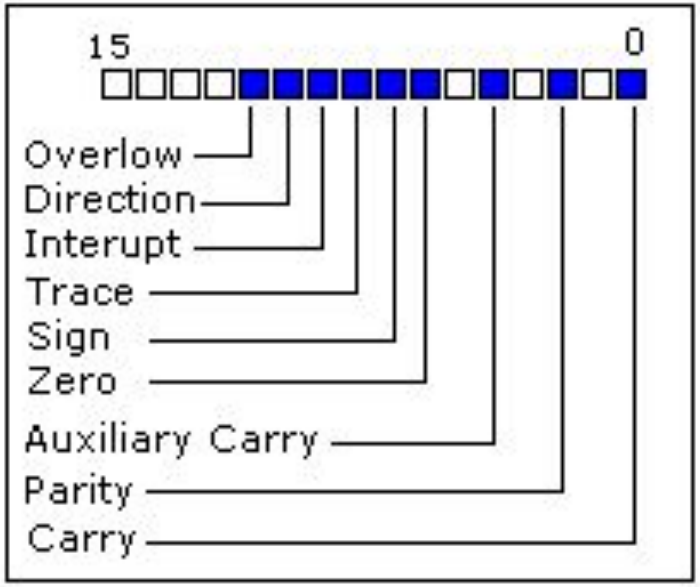
Арифметичні і логічні команди

Більшість арифметичних і логічних команд впливають на регістр стану процесора (або Прапори)



Як ви можете бачити, в цьому регістрі 16 біт. Кожен біт називається прапором і може приймати значення 1 або 0.

Carry Flag (CF) - перенос - цей прапор встановлюється в **1**, коли трапляється беззнаковое переповнення. Наприклад, якщо ви збільшили байт **255 + 1** (результат не поміщається в діапазоні 0 ... 255). Якщо переповнення не відбувається, цей прапор встановлений в **0**.

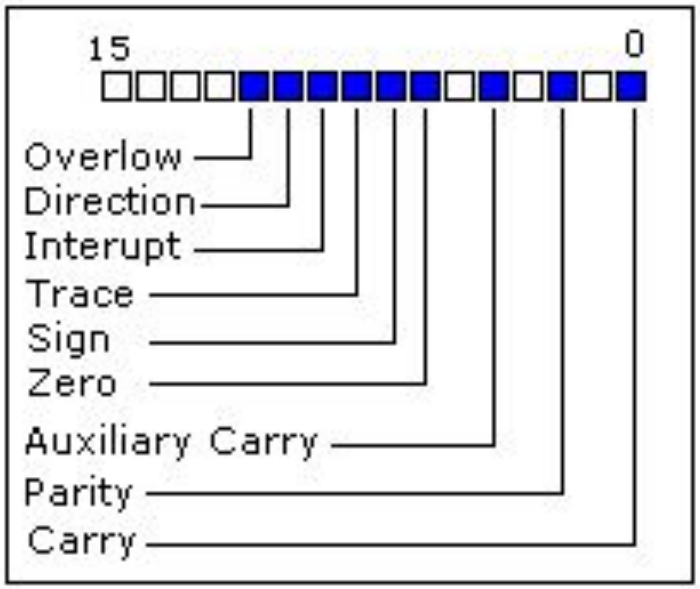
Zero Flag (ZF) - нуль - встановлюється в **1с**, якщо результат дорівнює нулю. Якщо результат не нульовий, то цей прапор встановлюється в **0**.

Sign Flag (SF) - знак - встановлений в 1с, якщо результат - **негативне число**. Якщо результат позитивний, то цей прапор встановлюється в **0**. Зазвичай цей прапор приймає значення старшого значущого біта.

Overflow Flag (OF) - переповнення - встановлюється в **1**, якщо трапляється переповнення при арифметичних операціях зі знаком. Наприклад, якщо ви збільшили байт **100 + 50** (результат не поміщається в діапазоні -128 ... 127).

Parity Flag (PF) - контроль парності - цей прапор встановлюється в **1**, якщо в молодших 8-бітових даних парне число. Якщо число непарне, то цей біт встановлений в **0**. Навіть якщо результат - це слово, то аналізуються тільки 8 молодших біт!

Auxiliary Flag (AF) - зовнішній перенесення - встановлений в **1**, якщо трапилося **переповнення без знака** молодших 4-х бітів (тобто перенесення з 3-го біта).



Interrupt enable Flag (IF) - переривання - якщо цей прапор встановлений в **1**, то процесор реагує на переривання від зовнішніх пристроїв.

Direction Flag (DF) - напрям - цей прапор використовується деякими командами для обробки ланцюжка даних. Якщо прапор встановлений в **0** - обробка відбувається в прямому напрямку, якщо **1** - в зворотному.

Є три групи команд.

Перша група: **ADD**, **SUB**,**CMP**, **AND**, **TEST**, **OR**, **XOR**

Ці типи операндів підтримуються:

REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate

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

memory: [BX], [BX+SI+7], змінна, і т.д...
immediate: 5, -24, 3Fh, 10001101b, і т.д...

Після операції між операндами результат завжди записується в перший операнд. Команди **CMP** і **TEST** впливають тільки на прапори і не записують результат (ця команда використовується для прийняття рішення під час виконання програми).

Ці команди впливають тільки на прапори:
CF, **ZF**, **SF**, **OF**, **PF**, **AF**.

- **ADD** - Додати другий операнд до першого.
- **SUB** - Відняти другий операнд з першого.
- **CMP** - Відняти другий операнд з першого тільки для прапорів.
- **AND** - Логічне І між усіма бітами двох операндів. При цьому дотримуються правила:

1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0

Як бачите, ми отримуємо 1 тільки в тому випадку, якщо обидва біти рівні 1.

- **TEST** - Те ж саме, що **AND**, але **тільки для прапорів**.
- **OR** - логічне АБО між усіма бітами двохоперандів. При цьому дотримуються правила:

1 OR 1 = 1
1 OR 0 = 1
0 OR 1 = 1
0 OR 0 = 0

Як бачите, ми отримуємо **1** кожен раз, коли хоча б один біт дорівнює **1**.

- **XOR** - логічне XOR (АБО з виключенням) між усіма бітами двох операндів. При цьому дотримуються правила:

1 XOR 1 = 0
1 XOR 0 = 1
0 XOR 1 = 1
0 XOR 0 = 0

Як бачите, ми отримуємо **1** кожен раз, коли біти мають різне значення.

Друга група: **MUL**, **IMUL**, **DIV**, **IDIV**

Ці типи операндів підтримуються:

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

memory: [BX], [BX+SI+7], змінна, і т.д...

Команди **MUL** и **IMUL** впливають тільки на ці прапори:
CF, **OF**

Якщо результат перевищує розмір операнда, то ці прапори встановлені в **1**, якщо результат уміщається в розмір операнда, то ці прапори встановлені в **0**.

Для команд **DIV** та **IDIV** прапори не визначені.

- **MUL** - беззнаковое множення:
  якщо операнд - це **байт**:
  AX = AL * операнд.
  якщо операнд - це **слово**:
  (DX AX) = AX * операнд.

- **IMUL** - множення зі знаком:
  якщо операнд - це **байт**:
  AX = AL * операнд.
  якщо операнд - це **слово**:
  (DX AX) = AX * операнд.

- **DIV** - беззнаковое розподіл:
  якщо операнд - це **байт**:
  AL = AX / операнд
  AH = залишок (модуль).
  якщо операнд - це **слово**:
  AX = (DX AX) / операнд
  DX = залишок (модуль).

- **IDIV** - розподіл зі знаком:
  якщо операнд - це **байт**:
  AL = AX / операнд
  AH = залишок (модуль).
  якщо операнд - це **слово**:
  AX = (DX AX) / операнд
  DX = залишок (модуль).

Третя група: **INC**, **DEC**, **NOT**, **NEG**

Ці типи операндів підтримуються:

REG
memory
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
memory: [BX], [BX+SI+7], змінна, і т.д…
Команди **INC** и **DEC** впливають тільки на ці прапори:
ZF, **SF**, **OF**, **PF**, **AF**.

Команда **NOT** не впливає ні на які прапори!

Команда **NEG** впливає тільки на ці прапори:
CF, **ZF**, **SF**, **OF**, **PF**, **AF**.

- **NOT** - інвертування кожного байта операнда.
- **NEG** - Змінює знак операнда (доповнення до двох). Зазвичай вона інвертує кожен біт операнда, а потім додає до нього одиницю. Наприклад, 5 перетвориться в -5, а -2 перетворюється в 2.

Управління ходом програми

Управління ходом програми - дуже важлива річ. Це те, що змушує програму приймати рішення, в залежності від деяких умов.

Безумовні переходи

Основна команда, яка передає управління в іншу точку програми - це **JMP**.
Основний синтаксис команди **JMP**:
JMP мітка

Щоб оголосити **мітку** в вашій програмі, просто роздрукуйте її ім'я і в кінці додайте двокрапку ":". Мітка може бути будь-якою комбінацією символів, але не повинна починатися з цифри. Наприклад, нижче представлені три правильних оголошення міток:

label1:
label2:
a:

Мітка може бути оголошена на окремому рядку або перед будь-якою іншою командою, наприклад:

```
x1:
MOV AX, 1
x2: MOV AX, 2
```

Приклад команди **JMP**:

```
ORG    100h
MOV    AX, 5          ; записати в AX число 5.
MOV    BX, 2          ; записати в BX число 5.
JMP    calc           ; перейти до 'calc'.
back:  JMP stop       ; перейти до 'stop'.
calc:
ADD    AX, BX         ; добавити BX до AX.
JMP    back           ; перейти до 'back'.
stop:
RET                   ; повернутись в операційну
систему.
END                   ; директива для припинення
компіляції.
```

Звичайно, є більш простий шлях для обчислення результату з двома числами, але це хороший приклад застосування команди JMP.

Як ви можете бачити з цього прикладу, команда JMP може передавати управління іншій ділянці програми, який може знаходитися як після цієї команди, так і перед нею. Цей перехід може бути здійснений в межах поточного сегмента коду (65,535 байтів).

Короткі умовні переходи

Подібно команді JMP, яка виконує безумовний перехід, існують команди, які здійснюють умовний перехід (перехід, який здійснюється тільки в тому випадку, якщо виконується певна умова). Ці команди поділяються на три групи. Перша група тільки перевіряє окремий прапор, друга - порівнює числа зі знаком, третя - порівнює числа без знака.

Команди переходу, які перевіряють одиночний прапор

Команда	Опис	Умова	Зворотня команда
JZ , JE	Перехід, якщо "рівно" ("нуль"). Тобто якщо порівнювані значення рівні, то ZF = 1 і перехід виконується	ZF = 1	JNZ, JNE
JC , JB, JNAE	Перехід, якщо є перенос ("нижче", "не вище або дорівнює").	CF = 1	JNC, JNB, JAE
JS	Перехід по знаку.	SF = 1	JNS
JO	Перехід по переповнення.	OF = 1	JNO
JPE, JP	Перехід, якщо є паритет або паритет парний.	PF = 1	JPO
JNZ , JNE	Перехід по "не дорівнює" або по «не нуль"	ZF = 0	JZ, JE
JNC , JNB, JAE	Перехід, якщо немає переносу ("вище або дорівнює" або "не нижче").	CF = 0	JC, JB, JNAE
JNS	Перехід, якщо немає знака.	SF = 0	JS
JNO	Перехід, якщо немає переповнення.	OF = 0	JO
JPO, JNP	Перехід, якщо немає паритету або паритет непарний.	PF = 0	JPE, JP

Як бачимо, існують команди, які виконують однакові дії. Це нормально. Вони навіть ассемблюются в однаковий машинний код, тому буде непогано, якщо запам'ятаємо, що при компіляції команди **JE**, після дизассемблювання отримаємо її як: **JZ**.

Різні імена використовуються для того, щоб робити програми більш легкою для розуміння і кодування.

Команда	Опис	Умова	Зворотня команда
JE , JZ	Перехід, якщо "рівно" (=). перехід, якщо "ноль".	ZF = 1	JNE, JNZ
JNE , JNZ	Перехід, якщо "не рівно " (<>). Перехід, якщо "не нуль".	ZF = 0	JE, JZ
JG , JNLE	Перехід, якщо "більше" (>). Перехід, якщо "не менше або рівно" (not <=).	ZF = 0 and SF = OF	JNG, JLE
JL , JNGE	Перехід, якщо "менше" (<). Перехід, якщо "не більше або рівно" (not >=).	SF <> OF	JNL, JGE
JGE , JNL	Перехід, якщо "більше або рівно" (>=). Перехід, якщо "не менше" (not <).	SF = OF	JNGE, JL
JLE , JNG	Перехід, якщо "менше або рівно" (<=). Перехід, якщо "не більше" (not >).	ZF = 1 or SF <> OF	JNLE, JG

<> - цей знак означає "не дорівнює"

Команди переходу для чисел без знаків

Команда	Опис	Умова	Зворотня команда
JE , JZ	Перехід, якщо "рівно" (=). Перехід, якщо "нуль".	ZF = 1	JNE, JNZ
JNE , JNZ	Перехід, якщо "не рівно" (<>). Перехід, якщо "не нуль".	ZF = 0	JE, JZ
JA , JNBE	Перехід, якщо "вище" (>). Перехід, якщо "не нижче або рівно" (not <=).	CF = 0 and ZF = 0	JNA, JBE
JB , JNAE, JC	Перехід, якщо "нижче" (<). Перехід, якщо "не вище або рівно" (not >=). Перехід по переносу.	CF = 1	JNB, JAE, JNC
JAE , JNB, JNC	Перехід, якщо "вище або рівно" (>=). Перехід, якщо "не нижче" (not <). Перехід, якщо "нема переносу".	CF = 0	JNAE, JB
JBE , JNA	Перехід, якщо "нижче або рівно" (<=). Перехід, якщо "не вище" (not >).	CF = 1 or ZF = 1	JNBE, JA

Зазвичай, якщо потрібно порівняти два числових значення, то використовують команду **CMP** (вона робить те ж саме, що і команда SUB (віднімання), але не зберігає результат, а впливає тільки на прапори.

Логіка дуже проста, наприклад:

потрібно порівняти числа 5 і 2,

5 - 2 = 3

результат - НЕ НУЛЬ (Прапор Нуля - Zero Flag (ZF) встановлено в 0).

Другой приклад:

потрібно порівняти 7 і 7,

7 - 7 = 0

результат - НУЛЬ! (Прапор Нуля - Zero Flag (ZF) встановлено в 1 і команди **JZ** або **JE** виконають перехід).

Приклад команди CMP і умовного переходу:

```
include emu8086.inc
ORG    100h
MOV    AL, 25     ; записати в AL число 25.
MOV    BL, 10     ; записати в BL число 10.
CMP    AL, BL     ; порівняти AL з BL.
JE     equal      ; якщо AL = BL (ZF = 1), то перейти до мітки
equal.
PUTC   'N'        ; інакше, якщо AL <> BL, то продовжити
виконання
JMP    stop       ; програми - надрукувати 'N' и перейти до мітки
stop.
equal:            ; якщо програма на цій мітці,
PUTC   'Y'        ; то AL = BL, тому виведемо на екран 'Y'.
stop:
RET               ; сюди приходимо в любому випадку
END
```

Всі умовні переходи мають одне серйозне обмеження - на відміну від команди **JMP**, вони можуть виконувати перехід тільки на **127** байтів вперед або на **128** байтів назад (врахуйте, що великі команди ассемблюються в 3 і більше байтів).

Ми можемо легко подолати це обмеження, використовуючи наступний метод:

- Взяти зворотню команду з наведеної вище таблиці і виконати перехід до мітки **label_x**.
- Використовувати команду **JMP** для переходу до потрібної ділянки програми.
- Визначити мітку **label x**: тільки після команди **JMP**.

label_x: - може бути будь-яким ім'ям.

Приклад:

```
include emu8086.inc
ORG    100h
MOV    AL, 25     ; записати в AL число 25.
MOV    BL, 10     ; записати в BL число 10.
CMP    AL, BL     ; порівняти AL із BL.
JNE    not_equal  ; перехід, якщо AL <> BL (ZF = 0).
JMP    equal
not_equal:
; уявімо, що тепер в нас розміщується код, який асемеблюється більше ніж в 127 байтів...
PUTC   'N'        ; якщо ми опинились тут, то AL <> BL,
JMP    stop       ; тоді друкуємо 'N', і переходимо до мітки stop.
equal:            ; якщо ми опинились тут,
PUTC   'Y'        ; тоді AL = BL, тоді друкуємо 'Y'.
stop:
RET               ; сюди переходимо в любому випадку.
END
```

Інший, рідше використовується метод, являє собою застосування безпосереднього значення (адреси) замість мітки. Якщо безпосереднє значення починається з символу '$', то виконується відносний перехід - перехід щодо поточногї адреси. Компілятор обчислює команду, яка знаходиться по заданому зміщенню, і виконує перехід безпосередньо до неї. Наприклад:

```
ORG    100h
; безумовний перехід вперед:
; пропускаємо наступні два байти,
JMP $2
a DB 3    ; 1 байт.
b DB 4    ; 1 байт.
; перехід назад на 7 байтів, якщо BL <> 0:
; (Команда JMP займає 2 байти)
MOV BL,9
DEC BL      ; 2 байти.
CMP BL, 0   ; 3 байти.
JNE $-7
RET
END
```

Процедури

Процедура - це частина коду, яка може бути викликана з вашої програми для виконання будь-якої певної задачі. Процедури роблять програму більш структурної і доступною для розуміння. У загальному випадку процедура повертає програму до тієї ж самої точки, звідки вона була викликана.

Синтаксис для оголошення процедури:
ім’я PROC

```
    ; тут знаходиться
    ; код процедури ...
```

RET
ім’я ENDP

ім’я - це ім'я процедури. Одне і теж ім'я повинно бути у верхній і нижній частині, це використовується для перевірки правильності закриття процедур.

Ви вже знаєте, що команда **RET** використовується для повернення в операційну систему. Ця ж команда використовується для повернення з процедури (фактично операційна система сприймає вашу програму, як спеціальну процедуру).

PROC і **ENDP** - це директиви компілятора, тому вони не асимілюються в який-небудь реальний машинний код. Компілятор тільки запам'ятовує адресу процедури.

Команда **CALL** використовується для виклику процедури.

Приклад:

```
ORG    100h
CALL   m1
MOV    AX, 2
RET                ; повернутися в операційну систему.
m1     PROC
MOV    BX, 5
RET                ; вернуться в програму, із якої була
викликана.
m1     ENDP
END
```

Вищеописаний приклад викликає процедуру **m1**, яка виконує команду **MOV BX**, **5**. Після закінчення процедури, програма виконує команду, наступну після команди **CALL**, тобто команду**: MOV AX, 2**.

Є кілька способів передачі параметрів процедурі. Найпростіший з них - використання регістрів. Тут представлений приклад процедури, яка приймає два параметри в регістрах **AL** і **BL**, примножує їх і повертає результат в регістр **AX**:

```
ORG   100h
MOV   AL, 1
MOV   BL, 2
CALL  m2
CALL  m2
CALL  m2
CALL  m2
RET               ; повернутись в операційну систему.
m2    PROC
MUL   BL            ; AX = AL * BL.
RET               ; повернутись до викликаної програми.
m2    ENDP
END
```

У цьому прикладі значення регістра **AL** змінюється кожен раз при виклику процедури, а стан регістра **BL** не змінюється. Таким чином ми отримали алгоритм обчислення числа 2 в 4-му ступені.
В результаті в регістрі **AX** буде число **16** (або 10h).

Тут дано ще один приклад, в якому використовується процедура для виведення на екран повідомлення **Hello World!**:

```
ORG    100h
LEA    SI, msg        ; загрузити адресу msg в регістр SI.
CALL   print_me
RET                   ; вернуться в ОС.
; ================================================================
; ця процедура друкує рядок, рядок повинен закінчуватися нулем (мати нуль в кінці),
; адреса рядка повинен бути в регістрі SI:
print_me     PROC
next_char:
   CMP  b.[SI], 0     ; перевірити регістр SI, якщо він
   JE   stop          ; дорівнює 0, перейти до мітки stop
   MOV  AL, [SI]      ; інакше отримати ASCII-символ.
   MOV  AH, 0Eh       ; номер функції для друку символу.
   INT  10h           ; використовуємо переривання для друку
                      ; символу із AL.
   ADD  SI, 1         ; збільшити індекс строкового масиву
   JMP  next_char     ; повернутися і надрукувати інший символ
stop:
RET                   ; вернуться в програму.
print_me     ENDP
; ================================================================
msg    DB  'Hello World!', 0   ; рядок з нульовим закінченням
END
```

"**b.**" - префікс перед [SI] означає, що нам необхідно порівнювати байти, а не слова. Якщо ви хочете порівняти слова, додайте префікс "**w.**" замість "**b.**". Якщо один з порівнюваних операторів - регістр, то вставляти префікси не потрібно, так як компілятор знає розмір кожного регістру.

Стек

Стек - це область пам'яті для зберігання тимчасових даних. Стек використовується командою **CALL** для зберігання адреси, щоб програма могла повернутися до того місця, звідки була викликана процедура. Команда **RET** отримує цю адресу з стека і повертає керування з цього зміщення. те ж саме відбувається, коли команда **INT** викликає переривання, вона записує в стек регістр прапорів, сегмент і зсув коду. Команда **IRET** використовується для повернення після виклику переривання.

Ви можете використовувати стек для зберігання будь-яких даних. Для роботи зі стеком є дві команди:

PUSH - записує 16-ти бітні значення в стек.

POP - отримує 16-ти бітні значення із стека.

Синтаксис для команди **PUSH**:

PUSH REG
PUSH SREG
PUSH memory
PUSH immediate
REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, CS.

memory: [BX], [BX+SI+7], 16-ти бітна змінна і т.д...

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

Синтаксис для команди **POP**:

POP REG
POP SREG
POP memory
REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, (кроме CS).

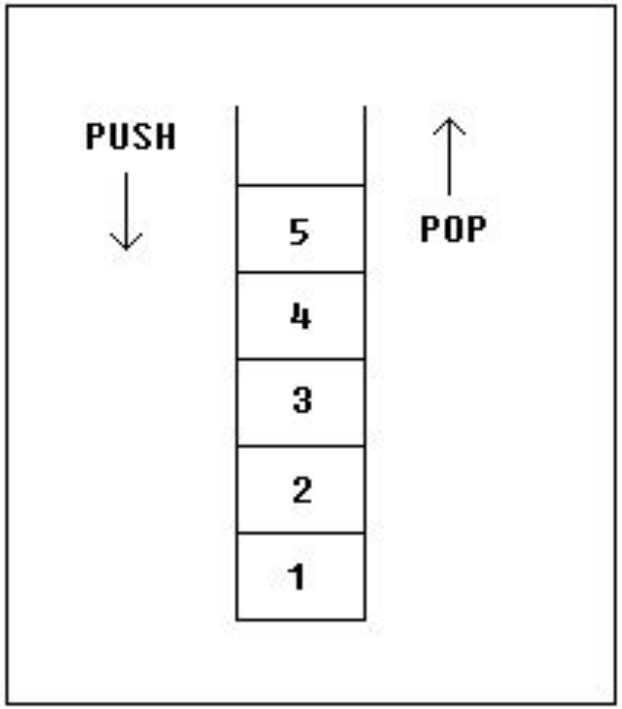
memory: [BX], [BX+SI+7], 16-ти бітна змінна і т.д...

- Команди **PUSH** і **POP** працюють тільки з 16-ти бітними значеннями!
- Примітка: **PUSH immediate** працює тільки на процесорах 80186 і вище!

Стек використовує алгоритм **LIFO** (Last In First Out - Останнім прийшов - першим пішов), це означає, що якщо ми помістимо ці значення одне за іншим в стек:

1, 2, 3, 4, 5

то першим значенням, яке ми можемо отримати з стека, буде **5**, потім **4**, 3, **2** і тільки потім **1**.



Дуже важливо застосовувати рівну кількість команд **PUSH** і **POP**, інакше стек може бути порушений і неможливо буде повернутися в операційну систему. Як ви вже знаєте, ми використовуємо команду **RET** для повернення в операційну систему. Коли програма запускається, її адреса записується в стек (зазвичай це **0000h**).

Команди **PUSH** і **POP** дуже корисні, тому що для зберігання даних зазвичай недостатньо тільки регістрів. Ось вихід із ситуації:

- Записати значення регістра в стек (використовуючи **PUSH**).
- Використовувати цей регістр в своїх цілях.
- Відновити попереднє значення регістра із стека (використовуючи **POP**).

Приклад:

```
ORG    100h
MOV    AX, 1234h
PUSH   AX          ; записати значення із AX в стек.
MOV    AX, 5678h   ; змінити значення регістра AX.
POP    AX          ; відновити попереднє значення AX.
RET
END
```

Стек можна також використовувати для того, щоб поміняти місцями значення в регістрах:

```
ORG    100h
MOV    AX, 1212h   ; записати в AX число 1212h.
MOV    BX, 3434h   ; записати в BX число 3434h.
PUSH   AX          ; записати значення AX в стек.
PUSH   BX          ; записати значення BX в стек.
POP    AX          ; встановити в AX значення BX.
POP    BX          ; встановити в BX значення AX.
RET
END
```

Обмін даними відбувається тому, що стек використовує алгоритм **LIFO** (Останнім прийшов - першим вийшов), тому коли ми поміщаємо в стек число **1212h**, а потім - **3434h**, то при зверненні до стека ми спочатку отримаємо число **3434h**, і тільки потім - **1212h**.

Область пам'яті стека встановлюється за допомогою регістрів **SS** (Stack Segment - сегмент стека) і **SP** (Stack Pointer - покажчик стека). Зазвичай операційна система встановлює значення цих регістрів на початок програми.

Команда «**PUSH джерел**о» робить наступне:

- Віднімає 2 з регістра SP.
- Записує значення джерела за адресою SS: SP.

Команда «**POP приймач**» робить наступне:

- Записує дані, розміщені за адресою SS: SP в приймач.
- Збільшує на 2 значення регістра SP.

Поточна адреса покажчика в **SS**: **SP** називається **вершиною стека**.

Для **COM**-файлів сегмент стека - це зазвичай і сегмент коду, а покажчик стека встановлений в значення **0FFFEh**. За адресою **SS**: **0FFFEh** записується адреса повернення для команди **RET**, яка виконується в кінці програми.

Ви можете спостерігати за роботою стека, натиснувши кнопку [**Stack**] у вікні емулятора. Вершина стека відзначена знаком "**<**".

Макроси

Макроси - це ті ж процедури, тільки віртуальні. Макроси подібні процедурам, але вони виконуються тільки під час компіляції. Після компіляції всі макроси замінюються реальними командами. Якщо ви оголосите макрос і ніколи не будете використовувати його в вашому коді, компілятор просто проігнорує його. emu8086.inc містить хороші приклади використання макросів. Цей файл містить декілька макросів, які роблять створення коду більш легким.

```
Визначення макроса:
ім’я    MACRO  [параметри,...]
          <команди>
ENDM
```

На відміну від процедур, макрос повинен бути визначений перед ділянкою коду, де він буде використовуватися, наприклад:

```
MyMacro    MACRO  p1, p2, p3
    MOV AX, p1
    MOV BX, p2
    MOV CX, p3
ENDM
ORG 100h
MyMacro 1, 2, 3
MyMacro 4, 5, DX
RET
```

Вищеописаний код еквівалентний наступному набору команд:

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX
```

Деякі важливі зауваження про макроси і процедурах:
• Якщо ви хочете використовувати процедуру, ви повинні застосувати команду CALL, наприклад:
CALL MyProc
• Якщо ви хочете використовувати макрос, ви можете просто надрукувати його ім'я. приклад:
MyMacro
• Процедура розташовується за певною адресою в пам'яті. Якщо ви скористаєтеся процедурою 100 раз, процесор буде передавати управління в цю частину пам'яті. Повернення з процедури виконується командою RET. Для зберігання адреси повернення використовується стек. Команда CALL займає 3 байти, так що розмір виконуваного файлу збільшується дуже незначно, незалежно від того, скільки разів викликається процедура.
• Макрос виконується безпосередньо в коді програми. Тому якщо ви використовуєте один і той же макрос 100 раз, то компілятор буде розпаковувати цей макрос також 100 раз, роблячи здійсненний файл все більше і більше, кожен раз вставляючи в програму команди макросу.
• Ви повинні використовувати стек або регістри загального призначення для передачі параметрів процедурі.
• Для передачі параметрів макросу, вам досить просто надрукувати їх після імені макросу. наприклад:
MyMacro 1, 2, 3
• Щоб позначити кінець макросу, досить директиви ENDM.
• Щоб позначити кінець процедури, ви повинні надрукувати ім'я процедури перед директивою ENDP.

Макрос розпаковується безпосередньо в коді, тому, якщо є мітки всередині макровизначення, ви можете отримати помилку "Duplicate declaration" (Подвійне оголошення), якщо макрос використовується два або більше разів. Щоб уникнути такої проблеми, використовуйте директиву **LOCAL**, яка супроводжує імена змінних, міток або імен процедур. наприклад:

```
MyMacro2    MACRO
        LOCAL label1, label2
        CMP  AX, 2
        JE label1
        CMP  AX, 3
        JE label2
        label1:
                INC  AX
        label2:
                ADD  AX, 2
ENDM
ORG 100h
MyMacro2
MyMacro2
RET
```

Якщо ви плануєте використовувати ваші макроси в декількох програмах, то краще розмістити всі макроси в окремому файлі. Помістіть цей файл в каталог **Inc** і застосуєте директиву **INCLUDE ім'я** файлу для використання макросів.

Як створити операційну систему

Зазвичай, коли комп'ютер стартує, він намагається завантажитися з першого 512-байтового сектора (це Циліндр 0, Головка 0, Сектор 1) дискети в дисководі A: в пам'ять за адресою 0000h: 7C00h і передати їй управління. Якщо це не вдається, то BIOS намагається використовувати MBR першого жорсткого диска.

Це для дискети. Ті ж принципи використовуються при завантаженні з жорсткого диска. Але використання дискети має кілька переваг:

- Ви можете зберегти існуючу операційну систему неушкодженою (Windows, DOS ...).
- Завантажувальний запис дискети легко модифікувати.

Приклад простий завантажувальної програми для дискети:

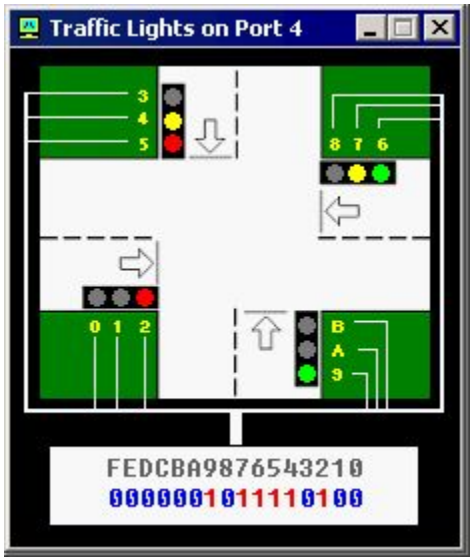
```
; директива для створення BOOT-файлу:
#MAKE_BOOT#
; Завантажувальний запис розміщується в 0000:7C00.  Цю інформацію потрібно повідомити
компілятору:
ORG 7C00h  ; завантажуємо адресу повідомлення в регістр SI:
LEA SI, msg  ; функція телетайпа:
MOV AH, 0Eh
print:   MOV AL, [SI]
         CMP AL, 0
         JZ done
         INT 10h   ; друк в режимі телетайпа.
         INC SI
         JMP print
; очікування натискання будь-якої клавіші:
done:      MOV AH, 0
           INT 16h
; записати магічне значення в 0040h:0072h:
;   0000h - холодна загрузка.
;   1234h - гаряча загрузка.
MOV     AX, 0040h
MOV     DS, AX
MOV     w.[0072h], 0000h   ; холодна загрузка.
JMP  0FFFFh:0000h    ; перезавантаження!
new_line EQU 13, 10
msg DB  'Hello This is My First Boot Program!'
    DB  new_line, 'Press any key to reboot', 0
```

Скопіюйте описаний вище приклад в редактор вихідного коду **Emu8086** і натисніть кнопку [**Compile and Emulate**]. Емулятор автоматично завантажить "**.boot**" файл за адресою **0000h**: **7C00h**.

Ви можете керувати ним як звичайною програмою або використовувати меню **Virtual Drive -> Write 512 bytes at 7C00h to -> Boot Sector** віртуального дисковода (файл FLOPPY_0 в каталозі, де встановлений емулятор). Після запису вашої програми в Віртуальний Дисковод, ви можете вибрати **Boot from Floppy** з меню **Virtual Drive**.

"**.boot**"-файлы имеют ограничение 512 байтов (размер сектора). Если ваша операционная система имеет размер, превышающий это ограничение, то вам придется использовать программу для загрузки из других секторов.

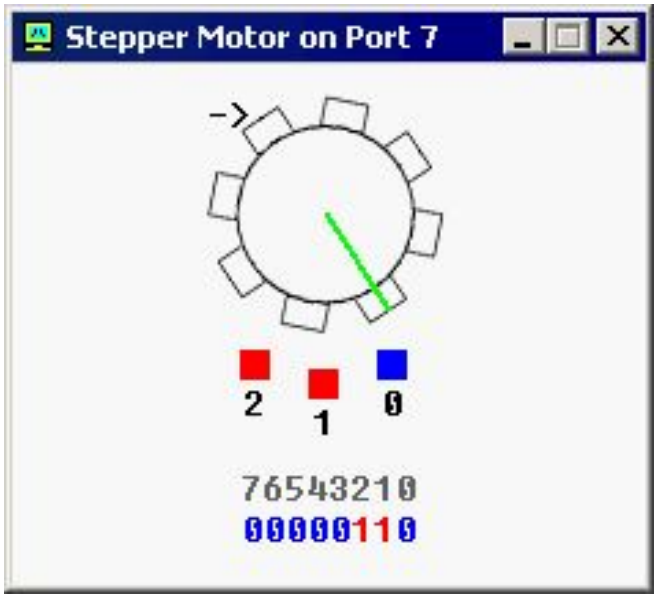
Светофор



Зазвичай для управління світлофором використовується масив (таблиця) значень. У певний період часу значення читається з масиву і відправляється в порт. наприклад:

```
; директива для створення BIN-файлу:
#MAKE_BIN#
#CS=500#
#DS=500#
#SS=500#
#SP=FFFF#
#IP=0#
; пропустити таблицю даних:
JMP start
table DW 1000011100001b
      DW 110011110011b
      DW 0011100001100b
      DW 011110011110b
start:
MOV SI, 0
; установить лічильники на кількість
; елементів в таблиці:
MOV CX, 4
next_value:
; получить значення із таблиці:
MOV AX, table[SI]
; встановити значення в порт вводу-виводу
світлофора:
OUT 4, AX
; наступне слово:
ADD SI, 2
CALL PAUSE
LOOP next_value
; почати із другого першого значення
JMP start
; ===========================
PAUSE PROC
; зберегти регістри:
PUSH CX
PUSH DX
PUSH AX
; встановити інтервал (1 мільйон мікросекунд - 1
секунда):
MOV     CX, 0Fh
MOV     DX, 4240h
MOV     AH, 86h
INT     15h
; відновити регістри:
POP AX
POP DX
POP CX
RET
PAUSE ENDP
; ===========================
```

Крокoвий двигун



Двигун може виконати "напівкрок" за допомогою пари магнітів, які повертають ротор двигуна на певний кут. Потім включається наступна пара магнітів і повертає ротор двигуна на наступний «крок» і т.д.

Двигун може виконати повний крок, якщо його повернути парою магнітів, потім іншою парою магнітів і в кінці - одиночним магнітом і т.п. Кращий спосіб здійснити повний крок - це виконати його як два півкроку.

Напівкрок - це 11.25 градусів.
Повний крок - це 22.5 градуса.

Двигун може обертатися як за годинниковою стрілкою, так і проти годинникової стрілки.

Робот

Для управління роботом повинен бути використаний набір алгоритмів для досягнення максимальної ефективності. Найпростішим, але дуже неефективним є алгоритм випадкових переміщень (див. Robot.asm в каталозі Samples).

Можна також використовувати таблицю даних (як і в світлофорі). Це може бути добре, якщо робот завжди працює в одній і тій же середовищі.