ЭВМ и Периферийные устройства

Вопрос с флагами

Пусть мы работает с 8-битныи операндом. Например, мы работает с AL

OF (Overflow Flag, флаг переполнения) срабатывает если результат слишком велик для помещения в 8-битный операнд. То есть:

- Сумма двух положительных **знаковых** превышает 127;
- Разность двух **знаковых** отрицательных операндов меньше -128.

CF (Carry Flag, флаг переноса) срабатывает если сумма двух **беззнаковых** операндов превышает 255;

SF (Sign Flag, флаг переноса) срабатывает, если результат становится меньше нуля

Естественно, это просто пример для 8 битных операндов. Для 16 битных и далее логика будет той же, только допустимые размеры изменятся.

Программа на Ассемблере под Win32

```
.386
 .model flat,stdcall ;  плоская модель памяти, соглашение о вызове процедур
option casemap:none ; регистр команд неважен

;набор подключаемых библиотек
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib

.data
    dig dd 123456890d
    MsgBoxCaption  db "Программа",0 ; заголовок она сообщения
    MsgBoxText       db  “HELLO WORLD!",0
    ifrmt db "%d", 0
    buf db ?


.code
    start: ; стартовая метка. Она должна присутствовать

        invoke MessageBox, NULL, ADDR MsgBoxText, ADDR MsgBoxCaption, MB_OK; Выводим HELLO WORLD

        invoke wsprintf, ADDR buf, ADDR ifrmt, dig ; перевод числа dig в строку и помещение её в buf
        invoke MessageBox, NULL, ADDR buf, ADDR MsgBoxCaption, MB_OK; выводим число 1234567890

        invoke ExitProcess, NULL; Завершить процесс

    end start ; конец программы
```

ExitProcess

Завершает работы программы с кодом результата.

```
invoke ExitProcess, код_результата_работы_in
```

MessageBox

Показывает информационной сообщение.

```
 invoke MessageBox,
дескриптор_родительского_окна_in,
адрес_текста_in,
адрест_текста_заголовка_in,
константа_обозначающая_набор_кнопок_окна_in
```

wsprintf

Преобразует число в строку.

```
 invoke wsprint,
адрес_результирующей_строки_out,
адрес_формата_строки_in,
Число1_in,число 2_in, …
```

Некоторые варианты формата:

- %d или %i – знаковый целый;
- %u – без знаковый целый.

Программа на Ассемблере под Win32 (консоль)

```
.386
…
BSIZE equ 14 ; заводим псевдооператор . BSIZE заменится 14

.data
    helloworld db "HELLO WORLD!",13,10
    ifrmt db "%d", 0
    dig dd 123456890d
    stdout dd ? ; поместим туда дескриптор консоли
    cWritten dd ? ; Будет хранить количество выведенных на экран символов
    buf db BSIZE dup (?)


.code
    start:
        invoke GetStdHandle, STD_OUTPUT_HANDLE ; Получить дескриптор для вывода данных и поместить его в eax
        mov stdout,eax ; stdout= значение eax

        invoke WriteConsole, stdout, ADDR helloworld, SIZEOF helloworld, ADDR cWritten, NULL

        invoke wsprintf, ADDR buf, ADDR ifrmt, dig ; перевод числа dig в строку
        invoke WriteConsole, stdout, ADDR buf, SIZEOF buf, ADDR cWritten, NULL ; Теперь выводим число

        invoke Sleep, 3000d ; Пауза на 3 сек
        invoke ExitProcess, NULL; Завершить процесс
    end start
```

WriteConsole

Пишет строку в консоль.

invoke **WriteConsole**,
декскриптор_устройства_вывода_in, адрес_сообщения_in,
размер_сообщения_in,
адрес_колличества_выведеных_символов_out,
зарезервировано

GetStdHandle

Получает дескриптор стандартного устройства и помещает его в EAX

```
invoke GetStdHandle, константа_номера_стандартного_устройства_in
```

Вместо константы номера стандартного устройства можно использовать обычное число. Для консоли это -11. Или STD_OUTPUT_HANDLE

«equ» или «=»

Эти псевдооператоры предназначен для присвоения некоторому выражению символического имени или идентификатора.

Впоследствии, когда в ходе трансляции этот идентификатор встретится в теле программы, ассемблер подставит вместо него соответствующее выражение.

Иными словами это просто что-то вроде автозамены.

имя_идентификатора equ **строка или числовое_выражение**

имя_идентификатора = строка или числовое_выражение

«equ» или «=»

Псевдооператор “**=**” удобно использовать для определения простых абсолютных (то есть не зависящих от места загрузки программы в память) математических выражений.

Главное условие то, чтобы транслятор мог вычислить эти выражения во время трансляции.

```
.data
adr1 db 5 dup (0)
adr2 dw 0
len = 43
len = len+1 ;можно и так, через предыдущее определение len = adr2-adr1
```

Ещё раз подчёркиваю – это автозамена ещё на этапе трансляции.

Обмен данными XCHG

Команда XCHG меняет значения регистров или регистров и памяти:

```
mov eax , 237h
mov ecx, 978h
xchg eax, ecx
```

В результате
eax = 978h
ecx = 237h

Переходы (прыжки)

Условный переход это такая команда процессору, при которой в зависимости от состояния регистра флагов производится передача управления по некоторому адресу иначе говоря прыжок.

Этот адрес может быть ближним или дальним. Прыжок считается ближним, если адрес, на который делается прыжок, находится не дальше чем 128 байт назад и 127 байт вперёд от следующей команды.

Дальний прыжок это прыжок дальше, чем на [-128,127] байт.

Безусловный переход jmp

jmp – команда прыжка к указанной метке. Не трогает стек в отличии от call. Может осуществлять и коротки и длинные прыжки.

```
jmp метка
```

```
start:
      jmp metka1

      invoke ExitProcess, NULL; Сюда не попадёт

      metka1: ; попадёт сюда
      mov ax,77d
      invoke ExitProcess, NULL;

   end start
```

Проще говоря, это аналог GOTO.

Условные переходы. cmp.

cmp – команда сравнения двух операндов.

cmp операнд1, операнд2

Выставляет флаги в зависимости от результата. Фактически вычитает **операнд1** из **операнда2** . В этот момент, естественно, выставляются флаги.

Замечу, что один из операндов запросто можно сделать 0 для ряда целей.

Условные переходы. cmp + je...

Код команды	Реальное условие	Условие для CMP
JA	CF=0 и ZF=0	если выше
JAE **JNC**	CF=0	если выше или равно если нет переноса
JB **JC**	CF=1	если ниже если перенос
JBE	CF=1 или ZF=1	если ниже или равно
JE **JZ**	ZF=1	если равно если ноль
JO	OF=1	если есть переполнение
JS	SF=1	если есть знак
JP	PF=1	если есть четность

Код команды	Реальное условие	Условие для CMP
JG	ZF=0 и SF=OF	если больше
JGE	SF=OF	если больше или равно
JL	SF<>OF	если меньше
JLE	ZF=1 или SF<>OF	если меньше или равно
JNE **JNZ**	ZF=0	если не равно если не ноль
JNO	OF=0	если нет переполнения
JNS	SF=0	если нет знака
JNP	PF=0	если нет четности

Пример. Прыжок на metka произойдет если eax=777h:

```
cmp eax, 777h
jz metka
```

Условные переходы могут делать только ближний прыжок.

Макрокоманды условного оператора.

В MASM существуют макроскрипты, упрощающие написание условий

```
.IF eax==1
;eax равен 1
.ELSEIF eax==3
; eax равен 3
.ELSE
; eax не равен 1 и 3
.ENDIF
```

Эта конструкция очень полезна. Вам не нужно вставлять сравнения и переходы, а только вставьте директиву .IF (не забудьте точку перед **.IF** и **.ELSE** и т.д.). Директива **.ENDIF** нужна для определения ещё одного сравнения, если предыдущие сравнения были ложными.

Инструкции после директивы **.ELSE** выполняются только в том случае, если все сравнения были ложными.

Макрокоманды условного оператора.

Также доступна вложенность:

```
.IF eax==1
.IF ecx!=2
; eax= 1 и ecx не равно 2
.ENDIF
.ENDIF
```

Доступны также привычные вам логические операторы. Вышенаписанное, например, можно записать как:

```
.IF (eax==1 && ecx!=2)
; eax = 1 и ecx не равно 2
.ENDIF
```

Подчеркиваю, что это не команды ассемблера, а макрокоманды, которые впоследствии раскладываются на команды ассемблера. Важно помнить об этом.

Макрокоманды условного оператора.

==	равно
!=	не равно
>	больше
<	меньше
>=	больше или равно
<=	меньше или равно
&	проверка бита
!	инверсия (NOT)
&&	логическое 'И' (AND)
||	логическое 'ИЛИ' (OR)
CARRY?	флаг переноса (cf) установлен
OVERFLOW?	флаг переполнения (of) установлен
PARITY	флаг паритета (pf) установлен
SIGN?	флаг знака (sf) установлен
ZERO?	флаг нуля (zf) установлен

Организация циклов. loop

loop проверяет, равен ли регистр ECX нулю, если
он не равен нулю, то значение регистра ECX уменьшается на 1 и
совершается ближний прыжок на указанное смещение (или метку)

```
mov ecx, 023h
repeat:
…; обязательно ближнее расстояние
Loop repeat
```

Тело цикла выполнится 23h раза.

Команда **loope** делает то же самое, но перед прыжком проверяет, установлен ли флаг ZF, если он установлен, то прыжок совершается. Точно тоже самое делает команда **loopz**. Команды **loopne** и **loopnz** делают то же самое что и loope, но прыгают, если флаг ZF сброшен.

Макрокоманды циклов.

.REPEAT - Эта конструкция выполняет блок, пока условие не истинно:

```
.REPEAT
; код здесь
.UNTIL eax==1
```

Эта конструкция повторяет код между .REPEAT и .UNTIL, пока eax не станет равным 1.
Вы можете использовать директиву .BREAK, чтобы прервать цикл и выйти.

```
.WHILE edx==1
inc eax ; увеличивает eax на 1
.IF eax==7
.BREAK
.ENDIF
.ENDW
```

Если eax=7, цикл while будет прерван.

Логические битовые операции

Логические операции с битами - OR, XOR, AND, NOT. Эти команды работают с приемником (операнд1) и источником(операнд2), исключение команда NOT (там только один операнд)

Каждый бит в приемнике сравнивается с тем же самым битом в источнике, и в зависимости от команды, 0 или 1 помещается в бит приемника:

Логические битовые операции. AND, OR

AND (логическое И) устанавливает бит результата в 1, если оба бита, бит источника и бит приемника установлены в 1.

приемник	1010
источник	1100
результат	1000

OR (логическое ИЛИ) устанавливает бит результата в 1, если один из битов, бит источника или бит приемника установлен в 1.

приемник	1010
источник	1100
результат	1110

Логические битовые операции. XOR, OR

XOR (НЕ ИЛИ) устанавливает бит результата в 1, если бит источника отличается от бита приемника.

приемник	1010
источник	1100
результат	0110

XOR позволяет быстро обнулять что-либо

```
xor eax, eax ; eax=0
```

NOT инвертирует бит источника.

приемник	1010
результат	0101

Логические битовые операции. Пример.

```
mov ax, 3406d
mov dx, 13EAh
xor ax, dx
```

ax = 3406 (десятичное), в двоичном - 0000110101001110.
dx = 13EA (шестнадцатиричное), в двоичном - 0001001111101010.

Выполнение операции XOR на этими битами:

Приемник = 0000110101001110 (ax)
Источник = 0001001111101010 (dx)
Результат = 0001111010100100 (новое значение в ax)

Стек. push, pop

Процессор имеет аппаратную поддержку стека. При этом стек, хранится в оперативной памяти

push – поместить значение (2 или 4 байта) в стек
pop – достать значение из стека (2 или 4 байта)

Стек растёт в сторону уменьшения адресов памяти.

Значение, помещенное в стек последним, извлекается первым.

Регистр ESP хранит адрес вершины стека.

Стек используется для хранения параметров процедур
И их локальных переменных.

Естественно, вы можете использовать стек в своих целях, **но вы отвечаете за его содержимое**

Стек. push

Пусть стек находится в следующем состоянии.
(стек здесь заполнен нулями, но в действительности это не так, как здесь). ESP стоит в том месте, на которое он указывает)

Смещение	1203	1204	1205	1206	1207	1208	1209	120A	120B
Значение	00	00	00	00	00	00	00	00	00
					ESP				

```
mov ax, 4560h
push ax
```

Смещение	1203	1204	1205	1206	1207	1208	1209	120A	120B
Значение	00	00	**60**	**45**	00	00	00	00	00
			ESP						

```
mov ax, 0FFFFh
push ax
```

Смещение	1203	1204	1205	1206	1207	1208	1209	120A	120B
Значение	**FF**	**FF**	**60**	**45**	00	00	00	00	00
	ESP								

Стек. pop

С момента предыдущего слайда стек находится в таком состоянии

Смещение	1203	1204	1205	1206	1207	1208	1209	120A	120B
Значение	**FF**	**FF**	**60**	**45**	00	00	00	00	00
	ESP								

pop edx

Смещение	1203	1204	1205	1206	1207	1208	1209	120A	120B
Значение	FF	FF	60	45	00	00	00	00	00
					ESP				

edx теперь равно 4560FFFF

Обратите внимание, что команда **pop** не чистит стек

Подпрограммы. Call.

Процедуры задаются директивами **proc** и endp. **Proc** обозначает начало процедуры, а **endp** конец процедуры. Вот пример объявления
процедуры:

```
SomeProc proc
...ещё код...
Ret ; обязательно
SomeProc endp
```

Вызов процедуры

```
Call SomeProc
```

Параметры процедуры должны быть выложены в стек перед её вызовом в обратном порядке.

Подпрограммы. Invoke.

Существует улучшенный способ вызова и задания процедур.

Invoke <функция>, <параметр1>, <параметр2>, <параметр3>

Для этого нужно сначала объявить прототип:

PROTO STDCALL testproc :DWORD, :DWORD, :DWORD

STDCALL указывать необязательно. DWORD тип. Возможны типы, например WORD, BYTE и т. д..
Вызов процедуры:

Invoke testproc, 1, 2, 3,

При этом ещё на этапе компиляции будут проверено количество параметров и их тип.

Подпрограммы. 2 способа.

```
MyProc PROTO :DWORD,:DWORD
.code
        MySimpleProc proc
            mov eax, dword ptr [esp+4]
            mov ecx, dword ptr [esp+8]
            ret
        MySimpleProc endp


        MyProc proc myparam1:DWORD, myparam2:DWORD
            LOCAL var1:DWORD ; локальная переменная 1
            LOCAL var2:BYTE,var3:WORD ; локальные переменные 2 и 3
            mov eax,myparam1
            mov ecx,myparam2
            ret
        MyProc endp

    start:
        xor eax,eax
        xor ecx,ecx
        push 2h
        push 1h
        call MySimpleProc
        invoke MyProc ,1h,2h
        invoke ExitProcess, NULL; Завершить процесс
    end start
```

Сдвиги. Логический сдвиг.

Логический свдиг – новые биты заполняются нулями, ушедшие исчезают.
SHL операнд, количество_сдвигов
SHR операнд, количество_сдвигов
SHL и **SHR** сдвигают биты операнда (регистр/память) влево или вправо соответственно на один разряд:

```
; al =              01011011 (двоичное)
shr al, 3 ; al= 00001011
```

Это означает: сдвиг всех битов регистра al на 3 разряда вправо. Так что al станет 00001011.

Биты слева заполняются нулями, а биты справа выдвигаются (исчезают). Последний выдвинутый бит, становится значением флага переноса CF.