Машинно-зависимые языки программирования

Лекция 3

Команды, которые мы не успели изучить (немного забегаем вперед, рассматривая команды не только процессора 8086):

`CMOVcc <приемник> <источник>` – команда условной пересылки данных. Будет пересылать значения из источника в приемник только при выполнении условия (аналогично команде Jcc).

`XCHG <операнд1>, <операнд2>` – обмен операндов между собой. Выполняется над двумя регистрами, либо над регистром и переменной.

> **?** *Почему она не может поменять местами две переменные?*
> Потому что шина адреса у нас всего одна и с оперативной памятью мы можем работать только в одну сторону (либо мы выставляем адрес и с него что-то читаем, либо выставляем и пишем). Выставить сразу два адреса нельзя, поэтому поменять сразу два значения памяти нельзя.

`XLAT [адрес] /XLAB` – трансляция в соответствии с таблицей.

Эта команда помещает в AL байт из таблицы[1] по адресу DS:BX со смещением относительно начала таблицы, равным AL.
Особенности:

1. Адрес, указанный в исходном коде, не обрабатывается компилятором и служит в качестве комментария.
2. Если в адресе явно указан сегментный регистр, он будет использован вместо DS (то есть смещение всегда берется из BX, а сегментная часть из указанного регистра)

Такая команда может использоваться, например, для транслитерации или для перевода 16-ричных чисел в символы (где мы значения от 0 до 9 будем конвертировать в цифры, а значения от 10 до 15 будем конвертировать в буквы от A до F).

Фактически, операция нужна для конвертации по словарю (причем “не линейно, а по массиву”). Мы объявляем целевой массив в памяти, настраиваем на него адреса DS:BX и в AL кладем номер, который хотим из этого массива получить. После этого этот номер подменяется значением из этого массива.

Да ну бред какой-то
Если вы хотите у меня спросить, как это работает, то лучше не спрашивайте

---

[1] Про таблицу – про себя надо читать это как массив, который не превышает 256 ключей памяти

`LEA <приемник>, <источник>` – вычисление эффективного адреса.
Вычисляет эффективный адрес источника и помещает его в приемник. Под адресом здесь тоже нужно понимать смещение.

Используется для:

1. Вычисления адресов на лету (позволяет вычислять адреса, описанные сложными методами адресации)
2. Быстрых вычислений

   ```
   lea bx, [bx + bx * 4]
   lea bx, [ax + 12]
   ```

   Такие вычисления занимают меньше памяти, чем соответствующие MOV и ADD, не изменяют флаги.
   (поддерживается сложение с константой и сложение/умножение какого-то регистра)

`ADD; ADC; SUB; SBB <п>, <и> (двоичная арифметика)`

С командами ADD и SUB мы уже знакомы. Команды ADC и SBB нужны для сложения каких-то больших чисел, которые в наши регистры не помещаются (работа с 32-разрядными числами).

—
П Есть у нас два слова. Старшая часть первого записана в dx, младшая в ax. Второе слово аналогично делит регистры bx и cx. Как мы можем их сложить? Складываем младшие половинки, в результате они могут сложится как без переполнения, так и с переполнением (в этом случае у нас выставляется флаг переполнения), после нужно сложить старшие половины, но так, чтобы учесть вопрос переполнения. Чтобы не делать это руками, используем ADC (сложение с учетом флага CF).

```
add ax, cx
adc dx, bx
```

Фактически, мы сложим приемник, источник и флаг CF. В результате в паре регистров dx и ax будет искомая сумма.
—

С вычитанием все точно также

```
sub ax, cx
sbb dx, bx
```

Обе команды выставляют сра9зу все 6 возможных арифметических флагов (CF, OF, SF, ZF, AF, PF).

`IMUL, MUL  (умножение чисел со знаком)`

```
IMUL <источник>[2]
IMUL <приемник>, <источник>
IMUL <приемник>, <источник1>, <источник2>
```

// Команды с 2 и 3 операндами уже не из нотации процессора 8086, если мы захотим использовать эти возможности, то в начале программы нужно будет писать соответствующие директивы (н, .586)

`IDIV, DIV (целочисленное деление со знаком)`

```
IDIV <источник>
```

Результат округляется в сторону нуля, знак остатка совпадает со знаком делимого.

Он был немногословен

`INC <п>, DEC <п> (инкремент, декремент)`

- Увеличивает/уменьшает приемник на 1.
- В отличие от ADD, не изменяет CF[3].
- OF, SF, ZF, AF, PF устанавливаются в соответствии с результатом.

`NEG <п>` – команда изменения знака (переводит число в доп код и прибавляет к нему единичку).

> **?** Когда требуется поменять знак?
> Например нам нужно выводить числа на экран. Для этого мы сначала должны определить знак, вывести его, а потом вывести само число.

---

[2] Как это работает, если верить интернету: когда размерность операнда составляет 8 бит, то команда `IMUL` производит умножение содержимого регистра AL на значение операнда и помещает результат в регистр AX.
Если операнд - 16-битное слово, команда IMUL производит умножение содержимого регистра AX на значение операнда и помещает результат в пару регистров DX:AX.

[3] `CF` – флаг переноса. Возникает в случае перехода через разрядную сетку при работе с беззнаковыми числами.

`DAA, DAS, AAA, AAS, AAM, AAD (десятичная арифметика)`

> **?** Что такое двоично-десятичные числа?
>
> Задействуются не все возможные двоичные коды. Если число не упакованное, то из 256 возможных значений для байта будет использоваться только 10 (*что-то такое он там говорит*), то есть число – это байт от 00h до 09h. Если упакованное, то будет байт от 00h до 99h (цифры A..F не задействуются).

Если такие числа пытаться складывать или как-то там изменять, то возникают проблемы с коррекцией:

`19h + 1 = 1Ah ⇒ 20h`

(типа должны были получить 20, а получили 1A ~~**ааааааа чего, все же правильно**~~, в общем перфоманс ситуации в том, что у нас числа упакованные, поэтому про буквы мы ничего не знаем, поэтому так и прыгаем)

Вердикт: команды есть, но особо никому не сдались.

`DAA`, `DAS` – команды для коррекции упакованных дд чисел после сложения/вычитания
`AAA`, `AAS` – команды для коррекции неупакованных дд чисел после сложения/вычитания
`AAM`, `AAD` – коррекция для неупакованных дд чисел после умножения/деления.

```
inc al
daa
```

`AND, OR, XOR, NOT, TEST` – логические команды (см. лекцию 1)

`SAR, SAL, SHR, SHL, ROR, ROL, RCR, RCL` – арифметический, логический, циклический сдвиг.

- SAR – shift arithmetic right, SAL – shift arithmetic left (арифметический, значит число сдвигается с учетом знака и знак (старший бит) в результате сохраняется)
- SAL тождественна SHL
- SHR зануляет старший бит, SAR – сохраняет (знак)
- ROR, ROL – циклический сдвиг вправо/влево
- RCR, RCL – циклический сдвиг через CF (задействуется не 16 бит, а 17, младший бит попадет в CF, а CF попадет в старший бит при сдвиге вправо)

`BT, BTR, BTS, BTC, BSF, BSR, SETcc` – операции над битами и байтами

> **?** Что такое байт?
> Минимальная адресуемая единица памяти

Работать с битами напрямую нельзя, но у нас есть чудо команды (которые нам правда тоже вряд ли понадобятся):

- BT <база>, <смещение> – считать в CF значение бита из битовой строки

  > **?** Что такое битовая строка?
  > Последовательность битов, которая интерпретируется как независимые биты, а не как какие-то данные (рассуждения о масках). Пример – регистр флагов.

- BTS <база>, <смещение> – установить бит в 1
- BTR <база>, <смещение> – сбросить бит в 0
- BTC <база>, <смещение> – инвертировать бит
- BSF <приемник>, <источник> – прямой поиск бита (от младшего разряда)
- BSR <приемник>, <источник> – обратный поиск бита (от старшего разряда)
- SETcc <приемник> – выставляет приемник (1 байт) в 1 или 0 в зависимости от условия, аналогично Jcc.

Организация циклов

- `LOOP <метка>` – уменьшает CX и выполняет “короткий” переход на метку, если CX не равен 0 (128 байт в обе стороны)
- `LOOPE / LOOPZ <метка>` – цикл “пока равно” / ”пока ноль” (флаг ZF)
- `LOOPNE / LOOPNZ <метка>` – цикл “пока не равно” / “пока не ноль” (флаг ZF)

Декрементируется CX и выполняется переход, если CX не ноль и если не выполняется условие (см флаг ZF)

// нужно переслушать разбор отладчика здесь

Строковые операции: копирование, сравнение, сканирование, чтение, запись

Строка источник - DS:SI, строка-приемник - ES:DI
За один раз обрабатывается один байт (слово).

`MOVS / MOVSB / MOVSW <п> <и>` – копирование
`CMPS / CMPSB / CMPSW <п> <и>` – сравнение
`SCAS / SCASB / SCASW <и>` – сканирование (сравнение с AL / AX)
`LODS / LODSB / LODSW <и>` – чтение (в AL / AX)
`STOS / STOSB / STOSW <п>` – запись (из AL / AX)

Исторически есть два способа хранения строк:

1. Сишный – нулевой байт в конце
2. Паскалевский – в первых двух байтах (или одном) хранится длина, а дальше идет непосредственно набор символов.

Разница в том, что во втором случае длину мы знаем заранее и нам не нужно пробегать всю строку, чтобы ее узнать, но при этом длина ограничена (а в си типа нет?). Еще один плюс паскалевского представления – нулевой байт может быть частью строки. *(Какой в ассемблере метод он не сказал..)*

? Как работают команды выше

Это довольно составные команды. Они читают с адреса источника байт (или два байта или может быть четыре байта), увеличивают индексы SI или DI на единицу (или на два, или на 4). Далее, если команды связаны с записью, то они будут записывать прочитанные байты в приемник.

! несмотря на указание <п> и <и> в командах, на самом деле в коде они прописываться не будут

Все команды работают в несколько тактов.

Если мы хотим работать со строками, а не с одним символом, то используются префиксы (пишутся в одну строчку через пробел с командой):

`REP / REPE / REPZ / REPNE / REPNZ`

При добавлении подобного префикса получится “цикл”, который будет выполняться столько раз, сколько выставлен CX (как в loop).
Команды с Z кроме CX будут еще контролировать состояние флага ZF.

// ну мы поняли, самим разбираться придется

Управление флагами

`STC / CLC / CMC` – установить / сбросить / инвертировать CF
`STD / CLD` – установить / сбросить DF[4]
`LAHF` – загрузка флагов состояния в AH
`SAHF` – установка флагов состояния из AH
`CLI / STI` – запрет[5] / разрешение прерываний (IF)

---

[4] Если DF == 0, то обработка строк будет идти слева направо, то есть по возрастанию адресов памяти. Если же выставить его в 1, то команды обработки строк будут идти в обратную сторону (то есть, например, они будут копировать данные из источника в приемник по уменьшению адресов от старшего к младшему, то есть справа налево)

[5] Программные прерывания блокировать глупо (мы их сами вызываем), а вот аппаратные прерывания блокировать (нажатие клавиатуры, прерывание таймера) имеет смысл (на время критичных системных операций)

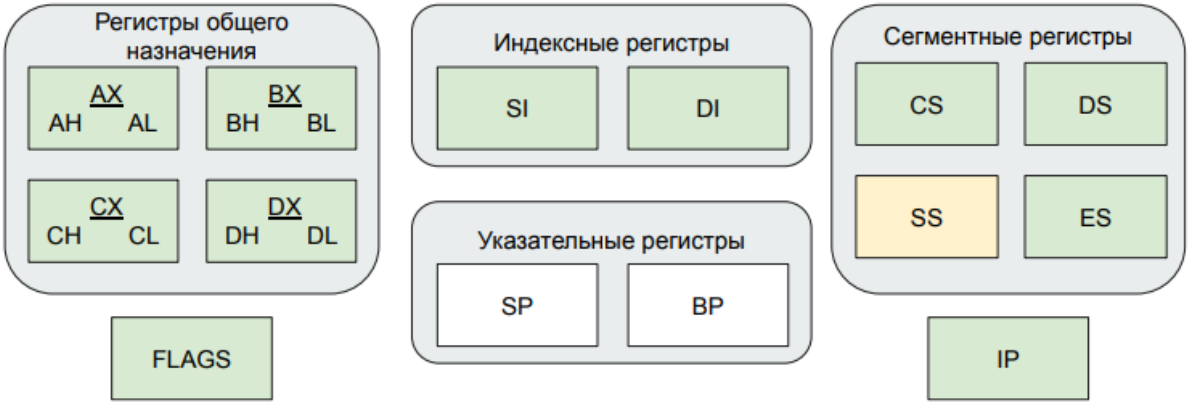
Загрузка сегментных регистров

`LDS <п>, <и>` – загрузить адрес, используя DS
`LES <п>, <и>` – загрузить адрес, используя ES
`LFS <п>, <и>` – загрузить адрес, используя FS
`LGS <п>, <и>` – загрузить адрес, используя GS
`LSS <п>, <и>` – загрузить адрес, используя SS

(все сегментные регистры, кроме CS, так как регистр с кодом напрямую менять низя)

<п> – регистр, <и> – переменная

Стек



SP – Stack Pointer (меняется автоматически процессором, но руками его тоже можно поменять)
BP – Base Pointer (вспомогательный регистр, используется программистом или компилятором)

Стек – структура данных, которая работает по принципу первым пришел, последним ушел (LIFO / FILO).
Сегмент стека – область памяти программы, используемая ее подпрограммами, а также (вынужденно) обработчиками прерываний.

SP – указатель на вершину стека
В x86 стек растет вниз, в сторону уменьшения адресов. При запуске программы SP указывает на конец сегмента.

? Почему так сделали
Чтобы проще было отследить переполнение стека (если адрес дошел до 0, значит мы заполнили все) => экономия регистра (не используем доп регистр для хранения размера стека).

Команды непосредственной работы со стеком

`PUSH <и>` – поместить данные в стек. Уменьшает SP на размер источника и записывает значение по адресу SS:SP
`POP <п>` – считать данные из стека. Считывает значения адреса SS:SP и увеличивает SP на величину приемника.
`PUSHA` – поместить в стек регистры AX, CX, DX, BX, SP, BP, SI, DI.
`POPA` – загрузить регистры из стека (SP игнорируется)
`PUSHF` – поместить в стек содержимое регистра флагов
`POPF` – загрузить регистр флагов из стека.

`CALL <операнд>` – вызов процедуры

- Сохраняет адрес следующей команды в стеке (уменьшает SP [в случае ближнего перехода на 2 байта, в случае дальнего на 4] и записывает по его адресу IP либо CS:IP, в зависимости от размера аргумента)
- Передает управление на значение аргумента

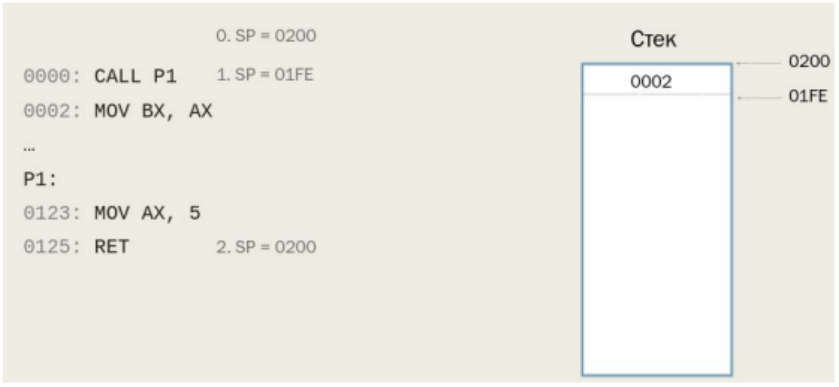
`RET / RETN / RETF <число>` – возврат из процедуры

- Загружает из стека адрес возврата, увеличивает SP
- retn – ближний возврат (2 байта), retf – дальний возврат (4 байта)
- Если указан операнд, его значение будет дополнительно прибавлено к SP для очистки стека от параметров

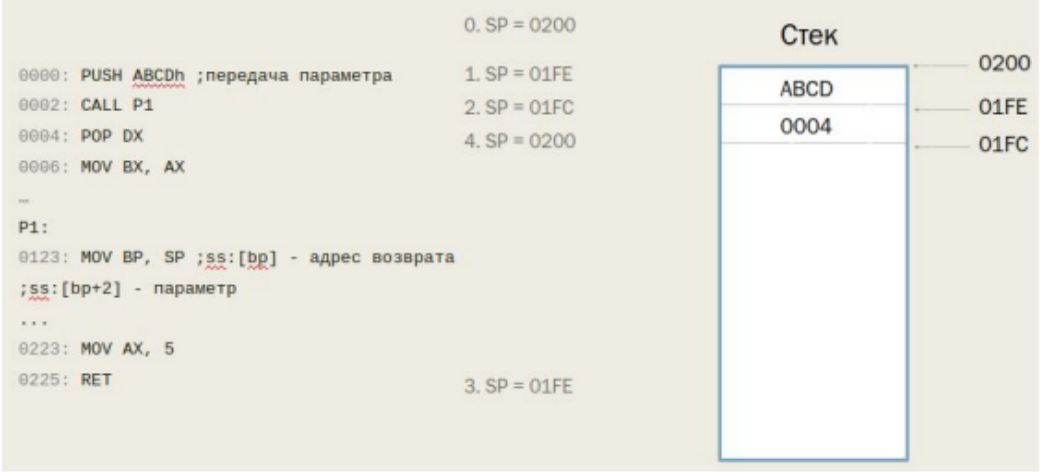
BP – base pointer

- Используется в подпрограмме для сохранения “начального” значения SP (типа чтобы в подпрограмме можно было писать что-то в стек, а по возвращении можно было вернуть состояние стека)
- Адресация параметров (то, что положила в стек вызывающая программа)
- Адресация локальных переменных (глобальные переменные обычно валяются в сегменте данных, локальные же там хранить нельзя, так как в этом случае при каждом вызове они будут перезатираться)

Пример вызова подпрограммы №1



Пример вызова подпрограммы №2



Пример вызова подпрограммы №3

Соглашения о вызовах – как передавать параметры (стек или регистры), как возвращать результат (по си возврат идет через ax), способы освобождения памяти от параметров.

Стековый кадр – механизм передачи аргументов и выделения временной памяти с использованием аппаратного стека. Содержит информацию о состоянии подпрограммы.

Включает в себя:

- параметры
- адрес возврата (обязательно)
- локальные переменные