LAB_2

Сам код:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <conio.h>

int main(void)

{

int aa;

clrscr();

int A[10]={2,1,2,3,4,5,6,7,8,9};

int B[10]={1,1,0,0,0,0,0,0,0,0};

A[0] = 2;

printf ("\n § зґЁп Ё¤ґЄб аґJЁбва®ў is, di = %x %x ", _SI, _DI);

//TASK 0 for

int i;

printf("\n Array: \n");

for(i = 0; i < 10; i++)

{

printf("\nA[i] = %d", A[i]);

}

printf("\n");

/*TASK 0 ASM SOURCE

mov eax, 0

jmp 0075

*uslovie*

inc eax

cmp eax, n

jl *metka*

*/

//TASK 1

/*

asm {

add ax, bx

add ax, ss:[bx]
```

```
and ax bx

}

*/

//TASK 2 A[]->B[] tiny

asm{
lea bx, A[0]
lea si, B[0]
mov cx, 0
}
label:
asm {
mov ax, ss:[bx]
mov ss:[si], ax
add bx, 2
add si, 2
inc cx
cmp cx, 10
jl label
}

//TASK 3 A[]->B[] loop

/*

asm{
lea bx, A[0]
lea si, B[0]
mov cx, 10
}

label:
asm{
mov ax, ss:[bx]
mov ss:[si], ax
add bx, 2
add si, 2
loop label
}

*/

//TASK 4 - REP MOVS - push ds

/*

asm {
REP MOVSb
cld
mov cx, 10
lea si, A[0]
lea di, B[0]
push ds
mov ax, es
mov ds, ax
rep movsw
pop ds
}

*/

//TASK 5 A[]->B[] - large lds

/*

int far* pA = (int*)A;

int far* pB = (int*)B;
```

```
asm {
push es
push ds
cld
lds si, pA
les di, pB
mov cx, 10
rep movsw
pop ds
pop es
}

*/

/*
//TASK 6 b8000 ->b8500

int far* pA = (int*) 0xB8000000;

int far* pB = (int*) 0xB8500000;

clrscr();

printf("%s", "qwertyuiop");

asm {
push ds
push es
lds si, pA
les di, pB
mov cx, 128
rep movsw
pop es
pop ds
}
*/

for(i = 0; i < 10; i++) {
printf("\nA[i] = %d B[i] = %d", A[i], B[i]);
}


getch();

return 0;

}
```

Объяснение каждого шага

01. Random commands

Этот блок кода содержит ассемблерные команды, которые выполняют арифметические операции.

```
add ax, bx
add ax, [bx]
and ax, bx
```

- `add ax, bx` — складывает значения регистров AX и BX и сохраняет результат в AX.
- `add ax, [bx]` — добавляет значение, находящееся по адресу, указанному в BX, к AX.
- `and ax, bx` — выполняет побитовую операцию AND между значениями в AX и BX.

---

02. A[ ] -> B[ ] tiny

```
asm {
        lea bx, A[0]  ; Загружает адрес первого элемента массива A в регистр BX
        lea si, B[0]  ; Загружает адрес первого элемента массива B в регистр SI
        mov cx, 0     ; Инициализирует счетчик CX в 0 (для отслеживания количества скопированных элементов)
}
label:
asm {
        mov ax, [bx]    ; Загружает текущее значение из массива A (по адресу в BX) в регистр AX
        mov [si], ax    ; Сохраняет значение из AX в массив B (по адресу в SI)
        add bx, 2       ; Увеличивает адрес в BX на 2 (переход к следующему 16-битному элементу массива A)
        add si, 2       ; Увеличивает адрес в SI на 2 (переход к следующему 16-битному элементу массива B)
        inc cx          ; Увеличивает счетчик CX на 1
        cmp cx, 10      ; Сравнивает CX с 10
        jl label        ; Если CX меньше 10, переходит обратно к метке label для следующей итерации

}
```

Объяснение:

Этот блок копирует элементы из массива A в массив B с использованием цикла.

- `lea bx, A[0]` — загружает адрес первого элемента массива A в регистр BX.
- `lea si, B[0]` — загружает адрес первого элемента массива B в регистр SI.
- `mov cx, 0` — инициализирует счетчик CX, который будет использоваться для отслеживания количества скопированных элементов.
  В метке `label`:
- `mov ax, [bx]` — загружает значение из массива A (по адресу, хранящемуся в BX) в AX.
- `mov [si], ax` — сохраняет значение из AX в массив B (по адресу, хранящемуся в SI).
- `add bx, 2` и `add si, 2` — увеличивают адреса для перехода к следующему элементу массива (предполагая, что `int` размером 2 байта).
- `inc cx` — увеличивает счетчик.
- `cmp cx, 10` — сравнивает счетчик с 10.
- `jl label` — если CX меньше 10, переход к метке `label` (продолжение цикла).

---

03. A[ ] -> B[ ] loop

```
asm {
        lea bx, A[0]  ; Загружает адрес первого элемента массива A в регистр BX
        lea si, B[0]  ; Загружает адрес первого элемента массива B в регистр SI
        mov cx, 10    ; Устанавливает счетчик CX в 10 (количество элементов для копирования)
}
label1:
asm {
        mov ax, [bx]   ; Загружает текущее значение из массива A в регистр AX
        mov [si], ax   ; Сохраняет значение из AX в массив B
        add bx, 2      ; Переходит к следующему элементу массива A (размер слова - 2 байта)
        add si, 2      ; Переходит к следующему элементу массива B
        loop label1    ; Уменьшает CX на 1 и переходит к метке label1, если CX не равен 0
}
```

Объяснение:

Этот блок также копирует элементы из массива A в массив B, но использует инструкцию `loop`, которая автоматически уменьшает CX на 1 и переходит к метке, если CX не равен нулю.

- `mov cx, 10` — устанавливает счетчик на 10, чтобы скопировать 10 элементов.
- `loop label1` — выполняет те же действия, что и в предыдущем блоке, но с использованием инструкции `loop` , что упрощает код.

---

04. A[ ] -> B[ ] - REP MOVS - push ds

```
asm {
    REP MOVSB
    cld                 ; Очищает флаг направления, устанавливая его в прямое направление (копирование от меньших адресов к
большим).
    mov cx, 10          ; Устанавливает CX в 10 — количество байтов для копирования.
    lea si, A[0]       ; Загружает адрес первого элемента массива A в регистр SI (Source Index).
    lea di, B[0]       ; Загружает адрес первого элемента массива B в регистр DI (Destination Index).
    push ds             ; Сохраняет текущее значение сегмента данных (DS) на стеке.
    mov ax, es          ; Загружает значение сегмента ES в регистр AX.
    mov ds, ax          ; Устанавливает сегмент данных (DS) в значение, хранящееся в AX.
    rep movsw           ; Копирует 10 (как указано в CX) 16-битных слов из DS:SI в ES:DI.
    pop ds              ; Восстанавливает значение сегмента данных (DS) из стека.
}
```

Объяснение:

Этот блок использует команду `REP MOVSB` для копирования массива.

- `REP MOVSB` — повторяет операцию перемещения байтов (MOVSB) CX раз.
- `cld` — очищает флаг направления для перемещения вперед.
- `mov cx, 10` — устанавливает количество байтов для копирования.
- `lea si, A[0]` и `lea di, B[0]` — загружают адреса массивов A и B.
- `push ds` — сохраняет сегмент данных на стеке.
- `mov ax, es` и `mov ds, ax` — переключает сегмент данных на ES для перемещения данных.
- `pop ds` — восстанавливает сегмент данных из стека.

---

05. A[ ] -> B[ ] - large lds

```
int far* pA = (int*)A;
int far* pB = (int*)B;

asm {
    push es          ; Сохраняем сегмент ES на стеке
    push ds          ; Сохраняем сегмент DS на стеке
    cld              ; Очищаем флаг направления (для копирования вперед)
    lds si, pA      ; Загружаем сегмент и адрес pA в SI
    les di, pB      ; Загружаем сегмент и адрес pB в DI
    mov cx, 10      ; Устанавливаем количество 16-битных слов для копирования
    rep movsw       ; Копируем CX слов (16-битных) из DS:SI в ES:DI
    pop ds          ; Восстанавливаем сегмент DS
    pop es          ; Восстанавливаем сегмент ES
}
```

Объяснение:

Этот блок копирует массивы с использованием сегментных регистров и команды `rep movsw`.

- **`far*`** указывает на работу с данными, которые могут находиться в другом сегменте памяти (важно для 16-битных систем, таких как DOS). `far` позволяет работать с памятью, которая может находиться за пределами текущего сегмента.
- `push es` и `push ds` — сохраняют текущие значения сегментных регистров на стеке.
- `cld` — указывает, что копирование будет выполнено в прямом направлении.
- `lds si, pA` — загружает сегмент и смещение для адреса `pA` в регистр `si`.
- `les di, pB` — загружает сегмент и смещение для адреса `pB` в регистр `di`.
- `rep movsw` — копирует 16-битные слова из массива A в массив B.

06. b8000 ->b8500

```
int far* pA = (int*) 0xB8000000;
```

```
int far* pB = (int*) 0xB8500000;

clrscr();

printf("%s", "qwertyuiop");

asm {
    push ds         ; Сохраняем сегмент данных
    push es         ; Сохраняем сегмент дополнительной памяти
    lds si, pA      ; Загружаем сегмент и смещение для адреса pA (B8000) в регистр DS:SI
    les di, pB      ; Загружаем сегмент и смещение для адреса pB (B8500) в регистр ES:DI
    mov cx, 128     ; Устанавливаем количество 16-битных слов (128 слов = 256 байтов)
    rep movsw       ; Копируем 128 слов (256 байтов) из DS:SI в ES:DI
    pop es          ; Восстанавливаем сегмент ES
    pop ds          ; Восстанавливаем сегмент DS
}
```

Объяснение:

- **`clrscr()`** — это стандартная функция, которая очищает экран, заполняя текстовую видеопамять пробелами и обновляя экран
- Код выводит строку `"qwertyuiop"` на экран с использованием видеопамяти в текстовом режиме (VGA).
- Затем он копирует содержимое видеопамяти с адреса **`0xB8000`** (где выводится текст) на адрес **`0xB8500`** (следующая область видеопамяти).
- Эта операция копирует первые 256 байт (128 символов) из одной области экрана в другую.

Пример вывода:

1. **Строка `"qwertyuiop"`** появляется на экране (например, в верхней строке).
2. После копирования содержимого, то же самое отображение текста будет продублировано в другой части экрана, начиная с позиции, соответствующей области памяти **`0xB8500`** (обычно это примерно середина экрана).

Как это выглядит на экране:

```
qwertyuiop                <-- оригинальная строка в начале экрана
[остальные символы или пробелы]
...
qwertyuiop                <-- дублированная строка (копия из другой области памяти)
```

Таким образом, текст `"qwertyuiop"` отображается дважды: один раз в стандартной области экрана и ещё раз в области, куда были скопированы данные видеопамяти.