

Машинно-зависимые языки программирования

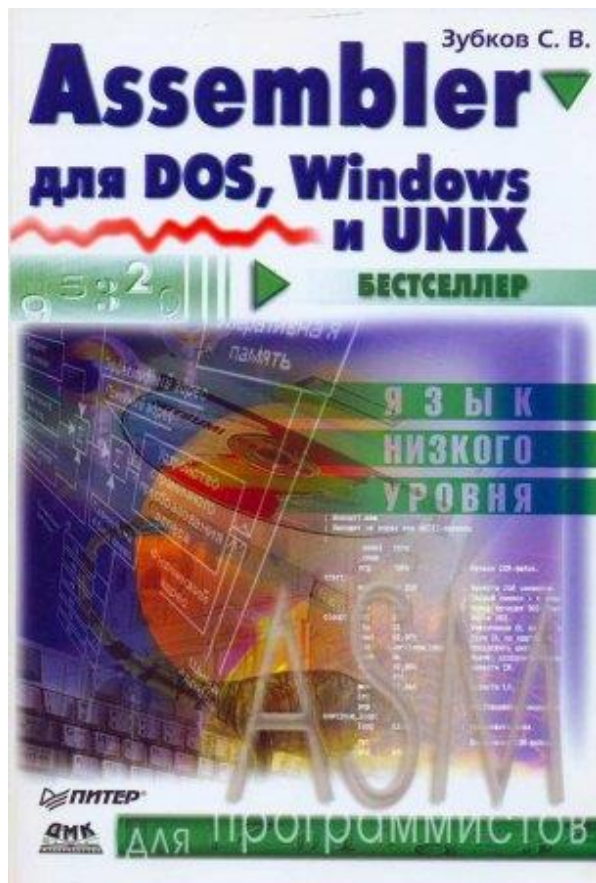
Савельев Игорь Леонидович

1. Литература
2. С чего начать?
3. Программная архитектура процессора
4. Сегменты, данные, инструкции
5. Арифметические операции
6. Логические операции
7. Условные операции
8. Организация циклов
9. Работа с массивами
10. Функции
11. Ввод/вывод
12. Сопроцессор

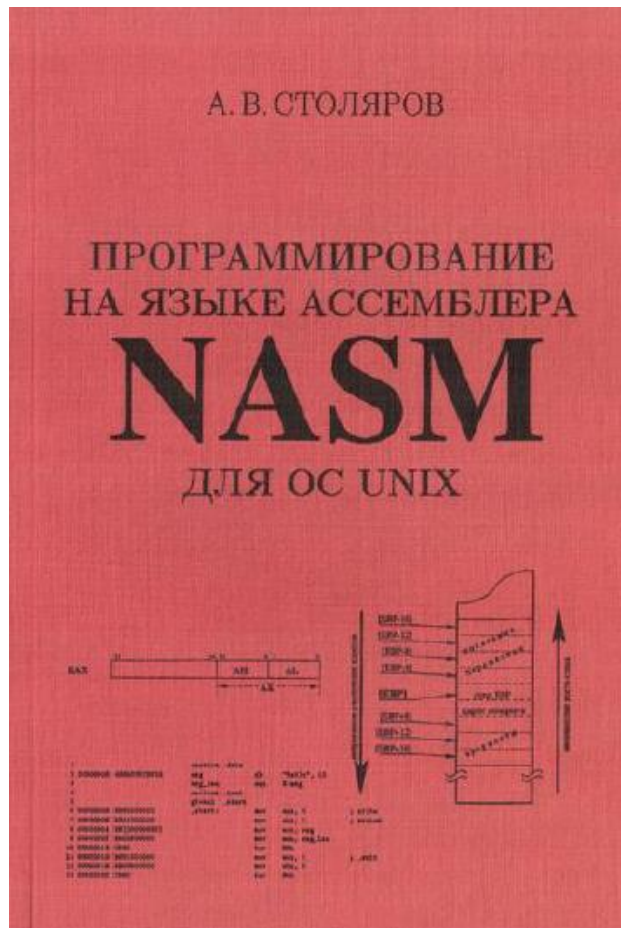
1. Литература



1. Литература



1. Литература



<https://www.felixcloutier.com/x86/index.html>

<https://filippo.io/linux-syscall-table/>

<https://www.opennet.ru/>

<https://stackoverflow.com/>

<https://dzen.ru/a/X2uFSG0z7E66hLAG>



Old Programmer
5,9К подписчиков

Подписаться



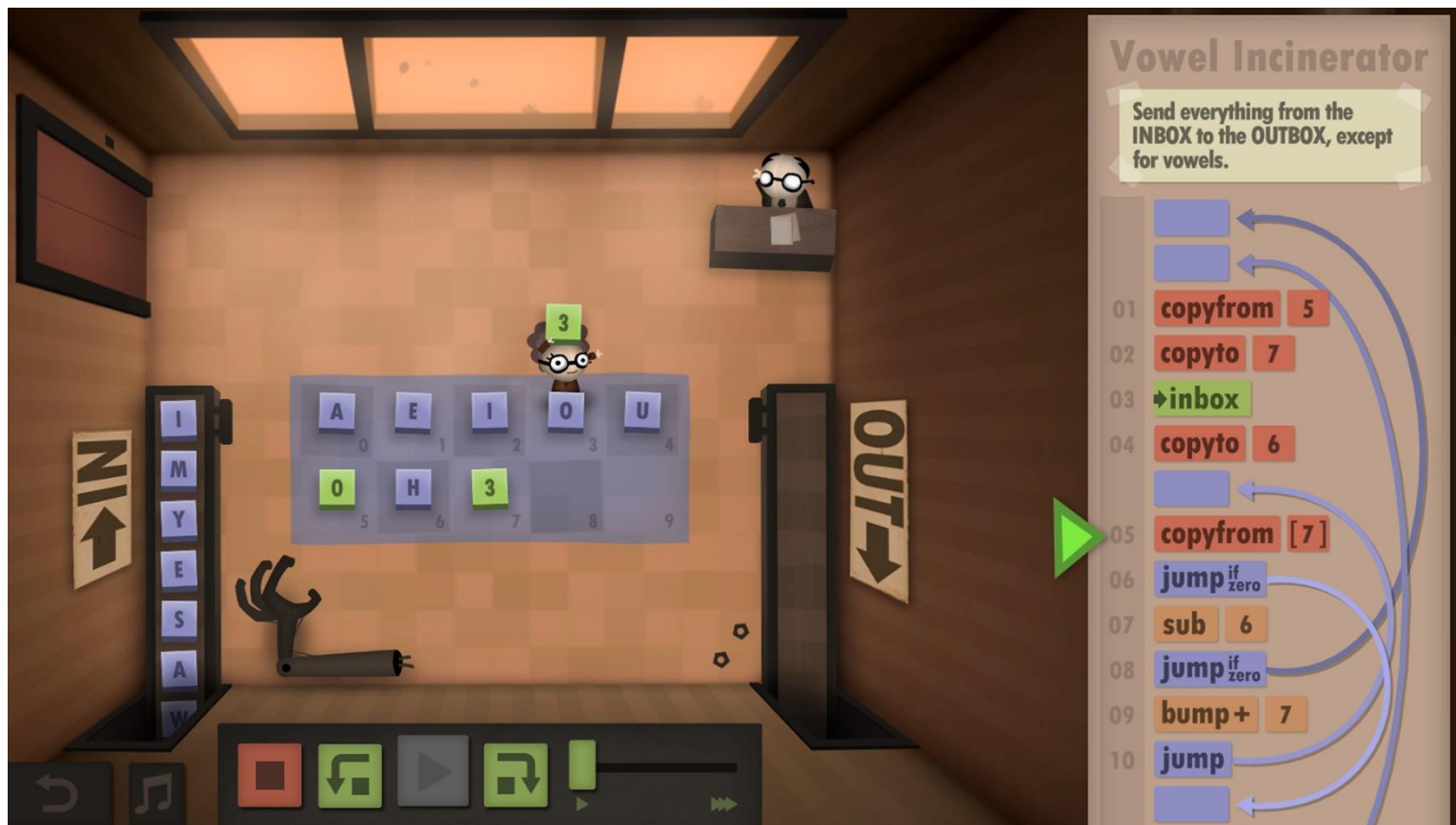
Язык ассемблера (x86-64, Linux). Путеводитель по ресурсам канала Old Programmer

23 сентября 2020 · 1К прочитали

Весь мой канал [Old Programmer](#) о программировании и программистах
представлен по темам [здесь](#).

О моих ресурсах по языку ассемблер

Human Resource Machine



Причины использования языка ассемблер

Программирование на ассемблере сейчас не так распространено как раньше. Однако есть причины для изучения этого языка. Вот основные из них:

- Образовательная причина. Очень важно знать, как микропроцессоры и компиляторы работают на уровне машинных инструкций.
- Отладка и проверка. Для просмотра ассемблерного кода, сгенерированного компилятором, или просмотра листинга, выданного дизассемблером, с целью поиска ошибок и проверки качества оптимизации критических участков программы.

2. С чего начать?

```
autofs4 e1000 floppy sg microcode keybdev mousedev hid input usb-uhci usbcore e
xt3 jbd halley_sd_mod scsi_mod
CPU: 1
EIP: 0047:[<544f54414c>] Not tainted
EFLAGS: 464f4e54
```

```
EIP is at rebalance_sensibility_matrix [kernel] 0x547 (2.16.1991.5.47.Lancs/fy
1.1ys)
```

```
eax: 00000000 ebx: c39d9018 ecx: c03a8250 edx: c3a17a84
```

```
esi: c39d8ffc edi: 4e415445 ebp: 464f4e54 tfg: 4745454b
```

```
Process kswapd (pid: 11, stackpage=c82e1000)
```

```
Stack: 00000000 00000001 00000000 53424144 00000000 00000000 00000047 c03a77000
00544647 00000000 00000001 00000000 0047c3p0 0000r2d2 544d5242 000000000
00415245 00005745 004e4f54 4d454e3f 00005745 00415245 4445564f 210000000
```

```
Call Trace: [<c0156c24>] do_pay_attention_007_pages_kswapd [kernel] 0x204 (0x
c82e1fac)
```

```
[<c0156d38>] kswapd [kernel] 0x47 (0x47555255)
```

```
[<c0156d38>] kswapd [kernel] 0x0 (0x4d445454)
```

```
[<c01095ad>] kernel_thread_helper [kernel] 0x5 (0x48454c50)
```

```
Code: 0f 0b 1b 03 40 05 47 2b b4 e92a f7 ff ff b8 04 00 00 e9 e9
```

```
Kernel panic: Fatal exception
```

Причины использования языка ассемблер

- Создание компиляторов. Понимание технологии ассемблирования непосредственно для создания компиляторов, отладчиков и других средств разработки.
- Ограниченность в ресурсах. Разработка для микроконтроллеров, недостаточное количество памяти или низкое быстродействие.
- За это платят.

Задача – найти номер наиболее старшего бита в числе unsigned int

```
static __inline__ int top_bit(unsigned int bits)
{
    int res = 31;
    int i = 0;

    for ( ; i < 32; i ++ )
    {
        if (bits & 0x80000000)
        {
            break;
        }
        bits <<= 1;
    }

    return (res - i);
}
```

```
static __inline__ int top_bit(unsigned int bits)
{
    int res = 31;
    int i = 0;

    @begin:
    if ( i < 32 )
    {
        if (bits & 0x80000000)
        {
            goto @end;
        }
        bits << 1;
        i++;
        goto @begin;
    }
    @end:
    return (res - i);
}
```

Пример (opal\plugins\audio\G.722.1\libg722_1\src\basop32.c)

```
/* Find the bit position of the highest set bit in a word
\param bits The word to be searched
\return The bit number of the highest set bit, or -1 if the word is zero. */
static __inline__ int top_bit(unsigned int bits)
{
    int res;

    if (bits == 0)
        return -1;
    res = 0;

    if (bits & 0xFFFF0000)
    {
        bits &= 0xFFFF0000;
        res += 16;
    }
    if (bits & 0xFF00FF00)
    {
        bits &= 0xFF00FF00;
        res += 8;
    }
}
```

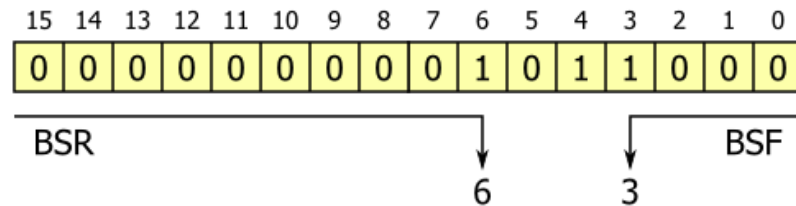
```
if (bits & 0xF0F0F0F0)
{
    bits &= 0xF0F0F0F0;
    res += 4;
}
if (bits & 0xCCCCCCCC)
{
    bits &= 0xCCCCCCCC;
    res += 2;
}
if (bits & 0xAAAAAAAA)
{
    bits &= 0xAAAAAAAA;
    res += 1;
}
return res;
}
```

Пример

Задача – найти номер наиболее старшего бита в числе unsigned int

```
/*! \brief Find the bit position of the highest set bit in a word
\param bits The word to be searched
\return The bit number of the highest set bit, or -1 if the word is zero. */
static __inline__ int top_bit(unsigned int bits)
{
    int res;

#ifdef __i386__ || defined(__x86_64__)
    __asm__ (" xori %[res],%[res];\n"
             " decl %[res];\n"
             " bsr  %[bits],%[res]\n"
             : [res] "=&r" (res)
             : [bits] "rm" (bits));
    return res;
#elif defined(__ppc__) || defined(__powerpc__)
    __asm__ ("cntlzw %[res],%[bits];\n"
             : [res] "=&r" (res)
             : [bits] "r" (bits));
    return 31 - res;
#endif
}
```



2. С чего начать? Простой пример

```
1                                section .data
2 00000000 B900                  a dw 185

3                                section .text
4                                global _start
5                                _start:
6 00000000 668B0425[00000000]      mov ax,[a]
7 00000008 66F7D0                 not ax
8 0000000B 6683C001              add ax,1
9
10 0000000F B801000000          mov eax,1
11 00000014 BB00000000          mov ebx,0
12 00000019 CD80               int 0x80
```

- Машинные коды
- Ассемблер

2. С чего начать? Простой пример

Создаем файл mytest.asm

```
section .data
    a dw 185
section .text
global _start
_start:
    mov ax,[a]
    not ax
    add ax,1

    mov eax,1
    mov ebx,0
    int 0x80
```

```
unsigned short a = 185;
```

```
void main()
{
    a = !a;
    a ++;
}
```

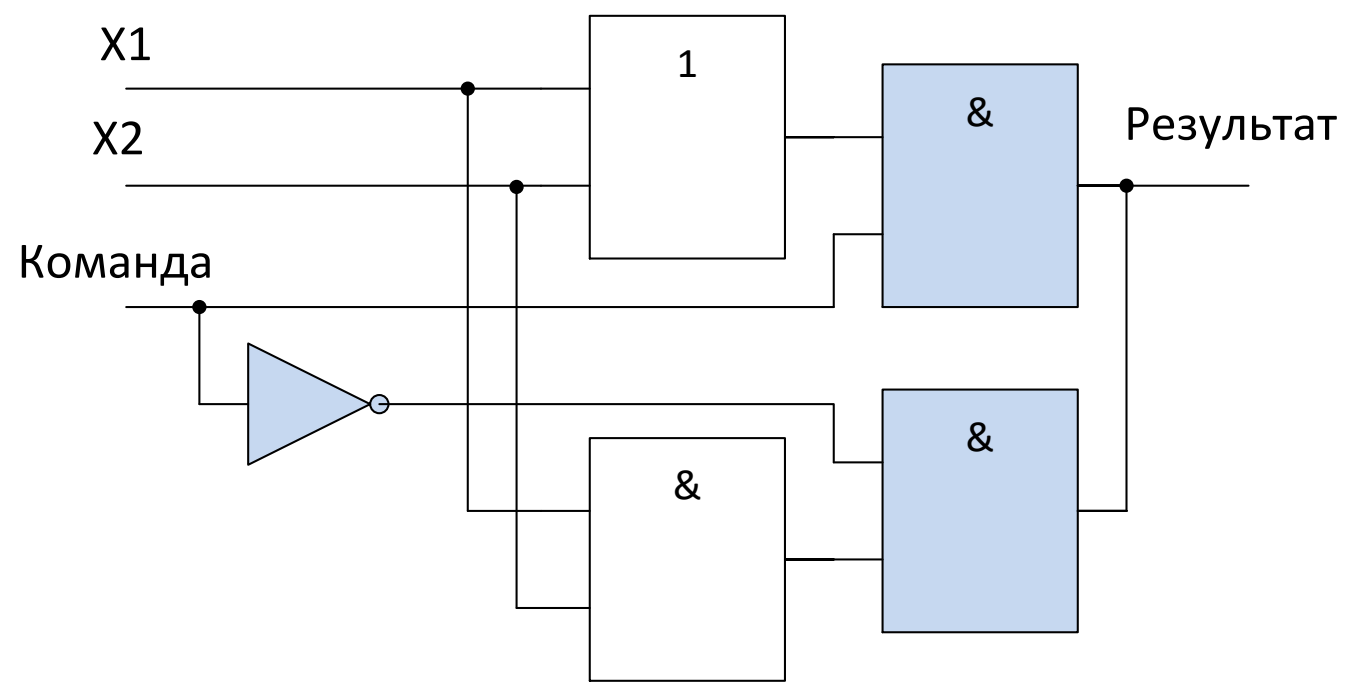
Сборка

```
nasm -g -f elf64 mytest.asm -F dwarf
```

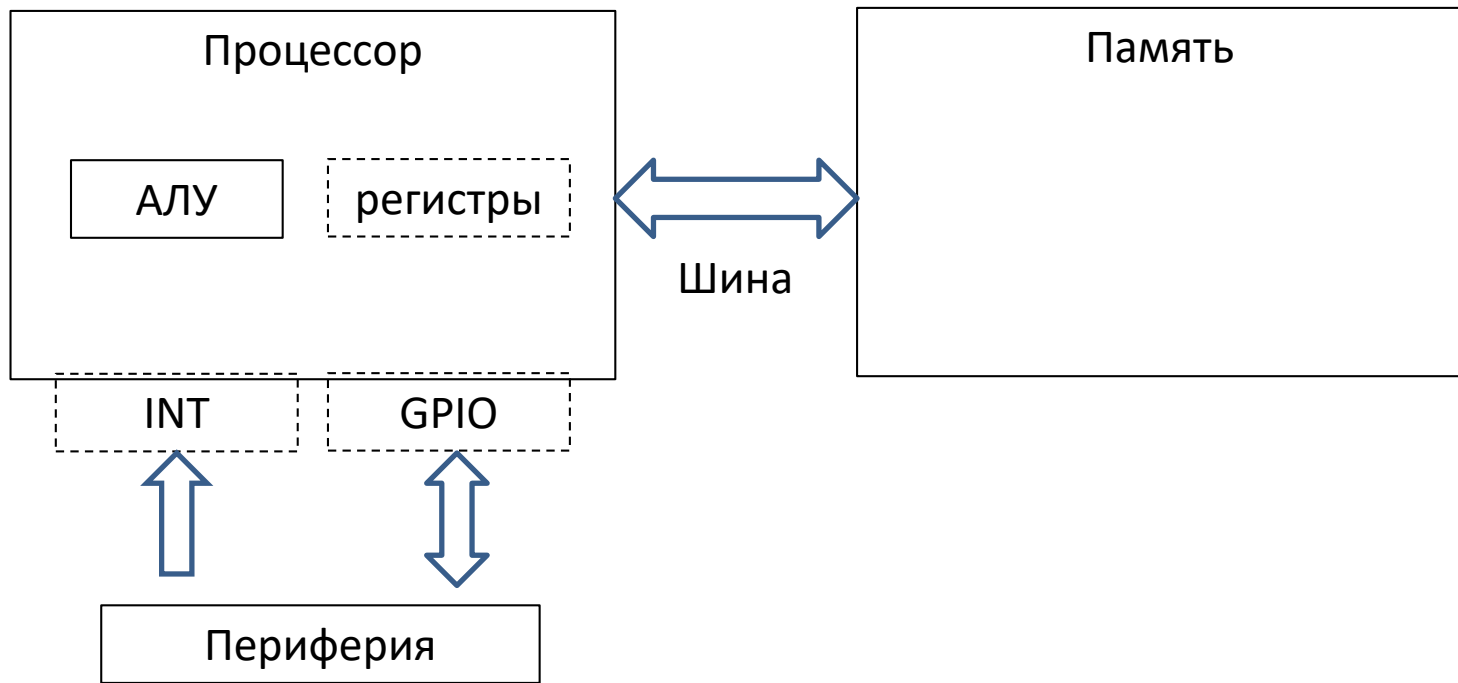
Линковка

```
ld -o mytest mytest.o
```

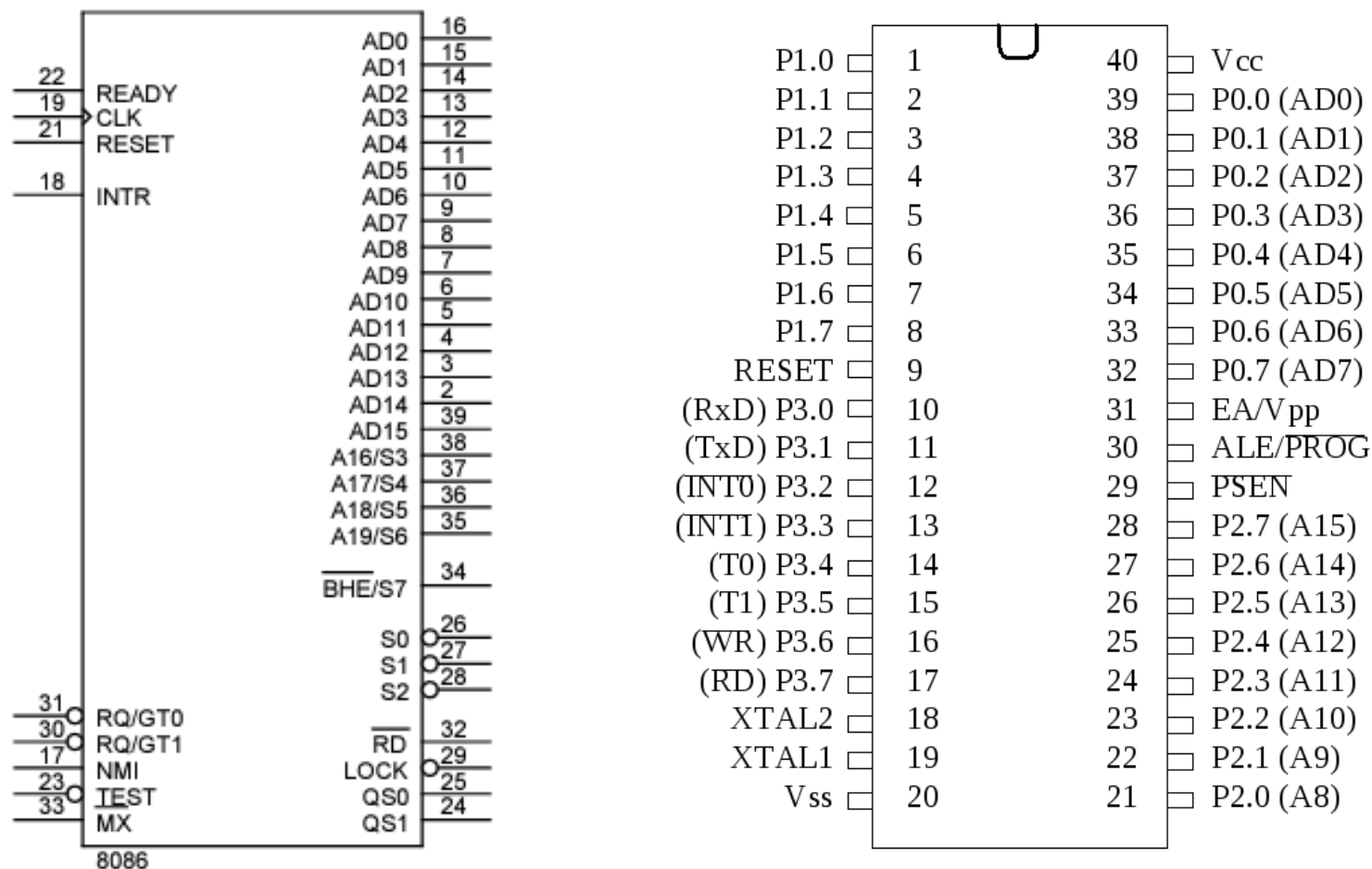
3. Программная архитектура процессора



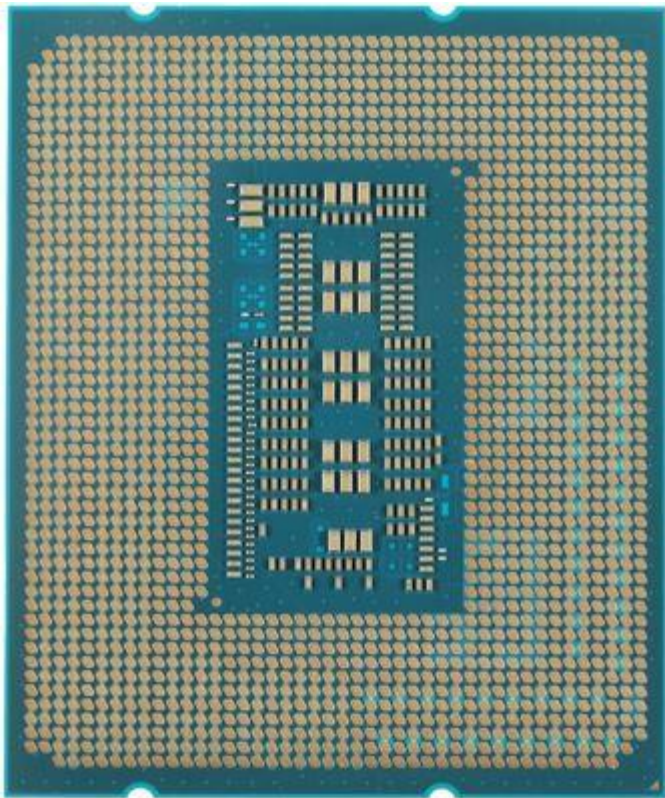
3. Программная архитектура процессора



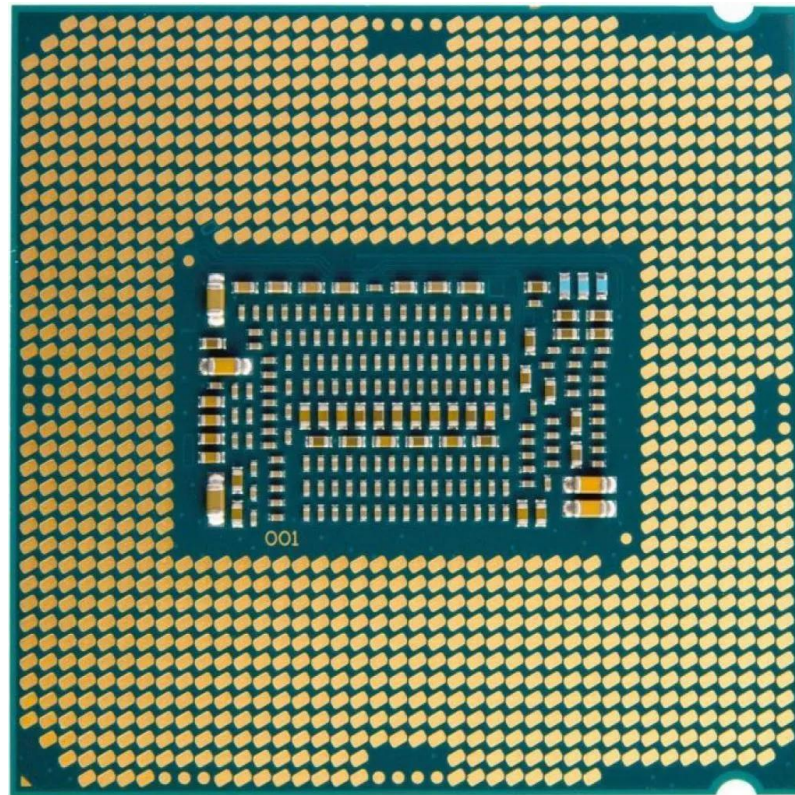
3. Программная архитектура процессора



3. Программная архитектура процессора

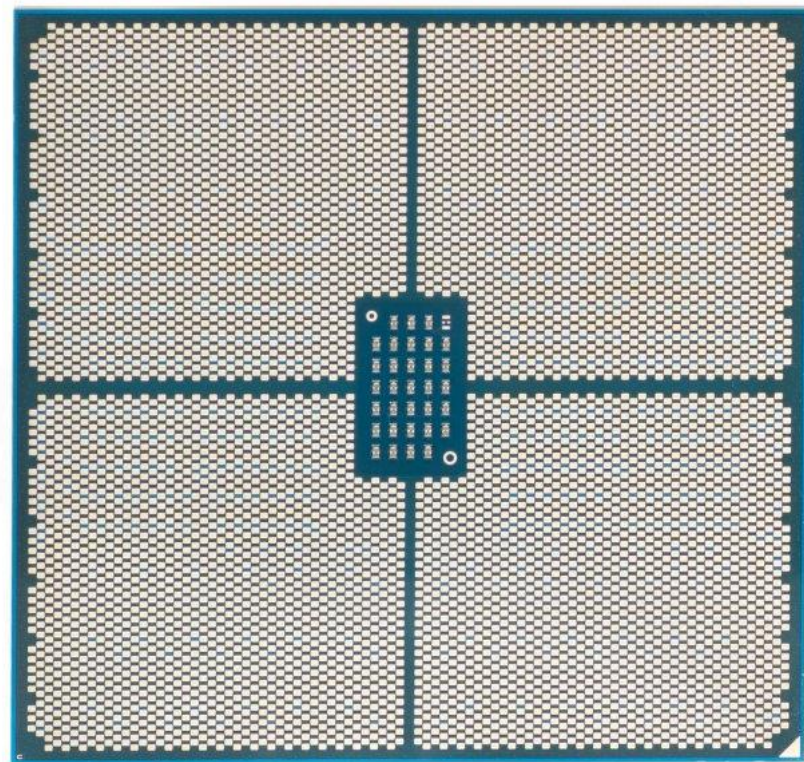
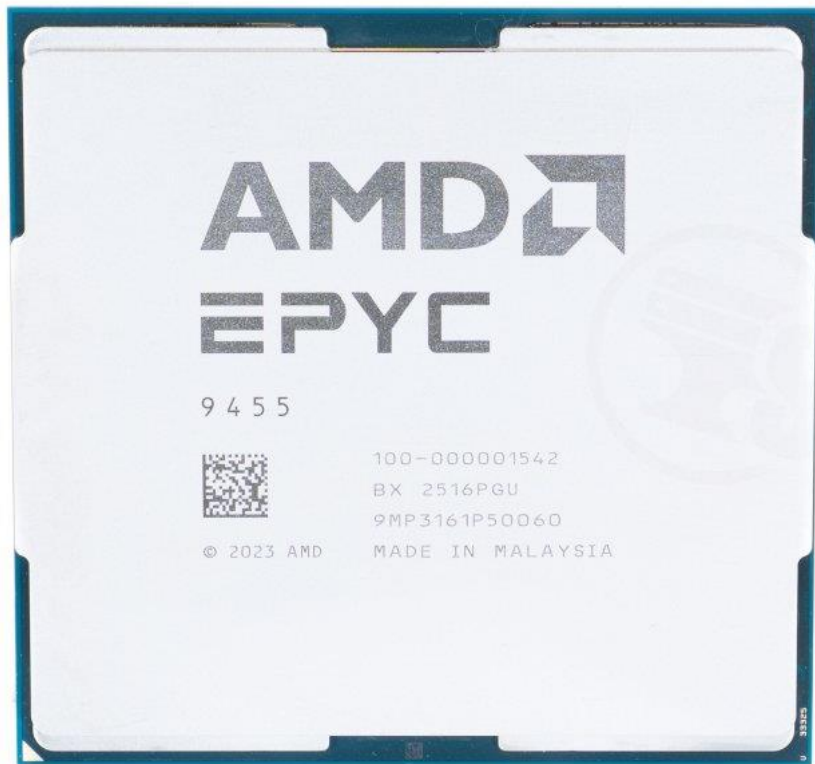


LGA 1700



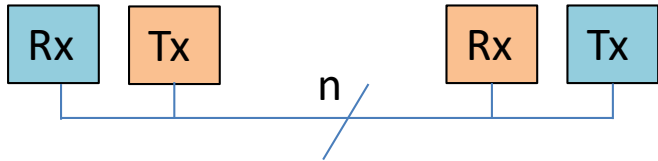
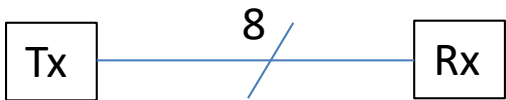
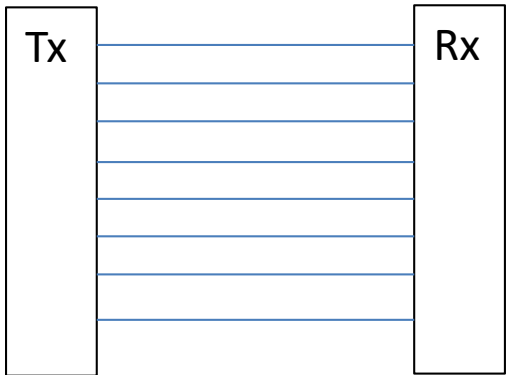
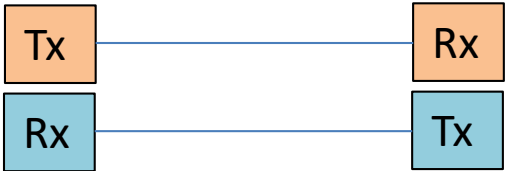
LGA 1151

3. Программная архитектура процессора

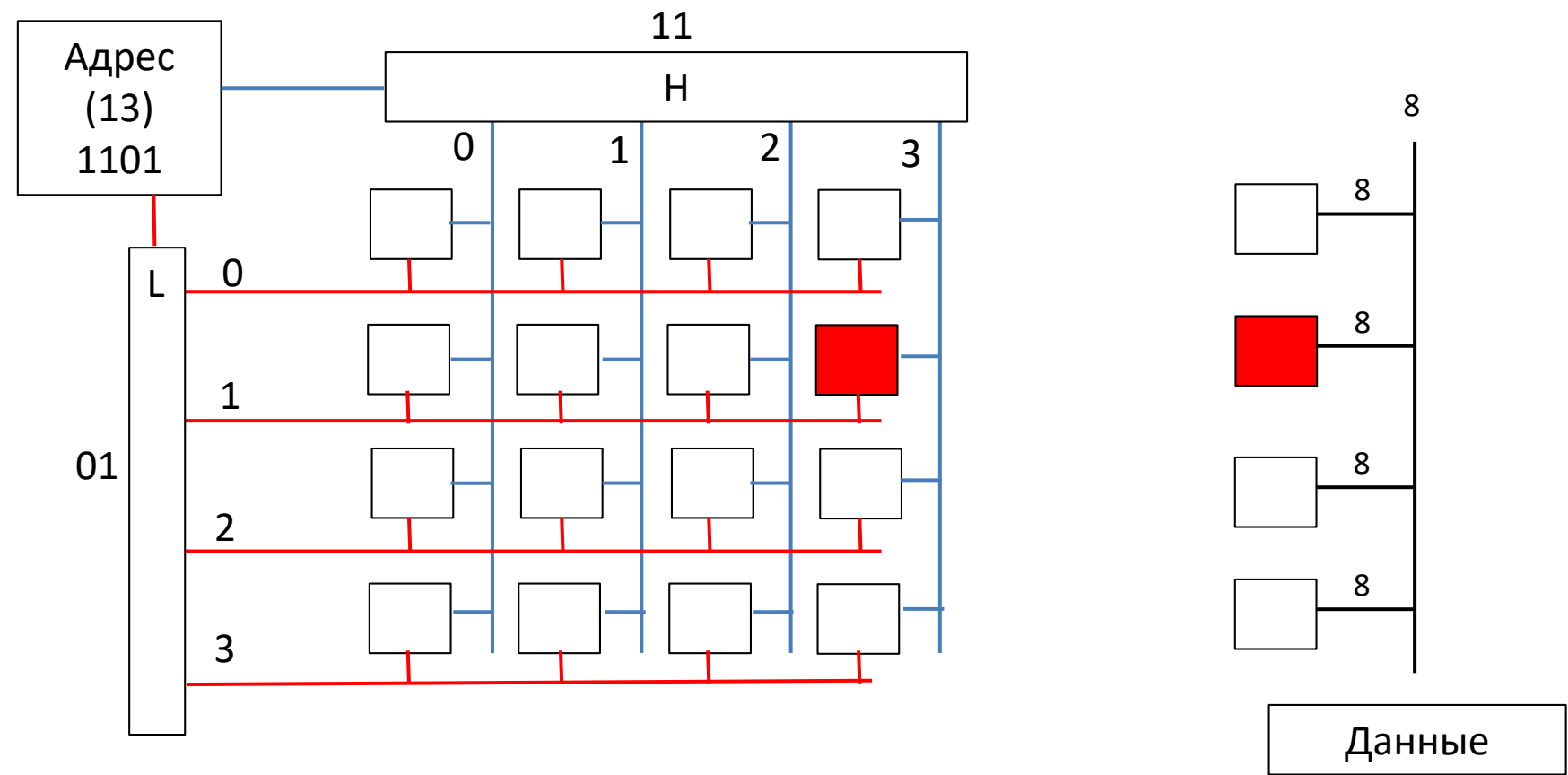


LGA 6096

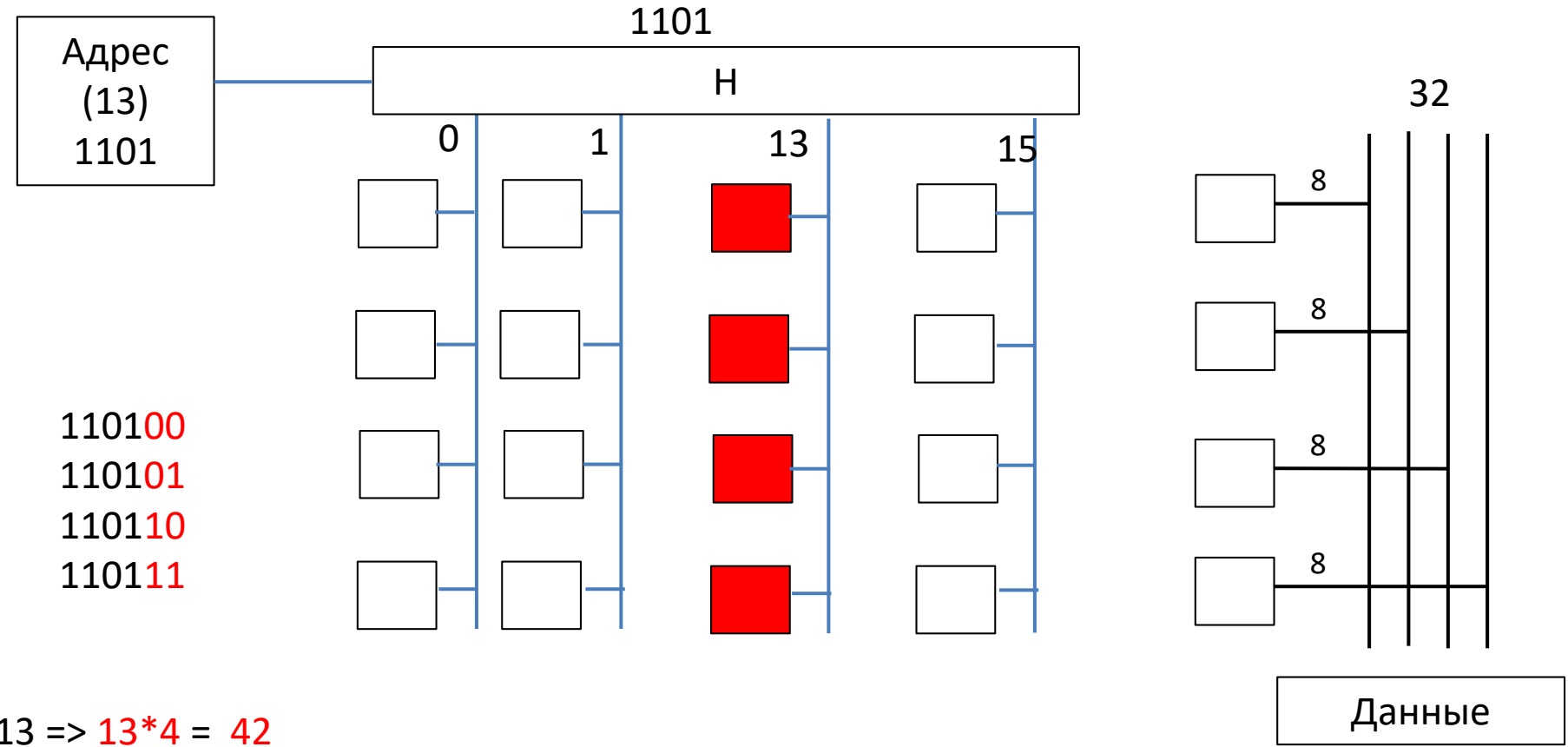
3. Программная архитектура процессора



3. Программная архитектура процессора (шина адреса и данных)

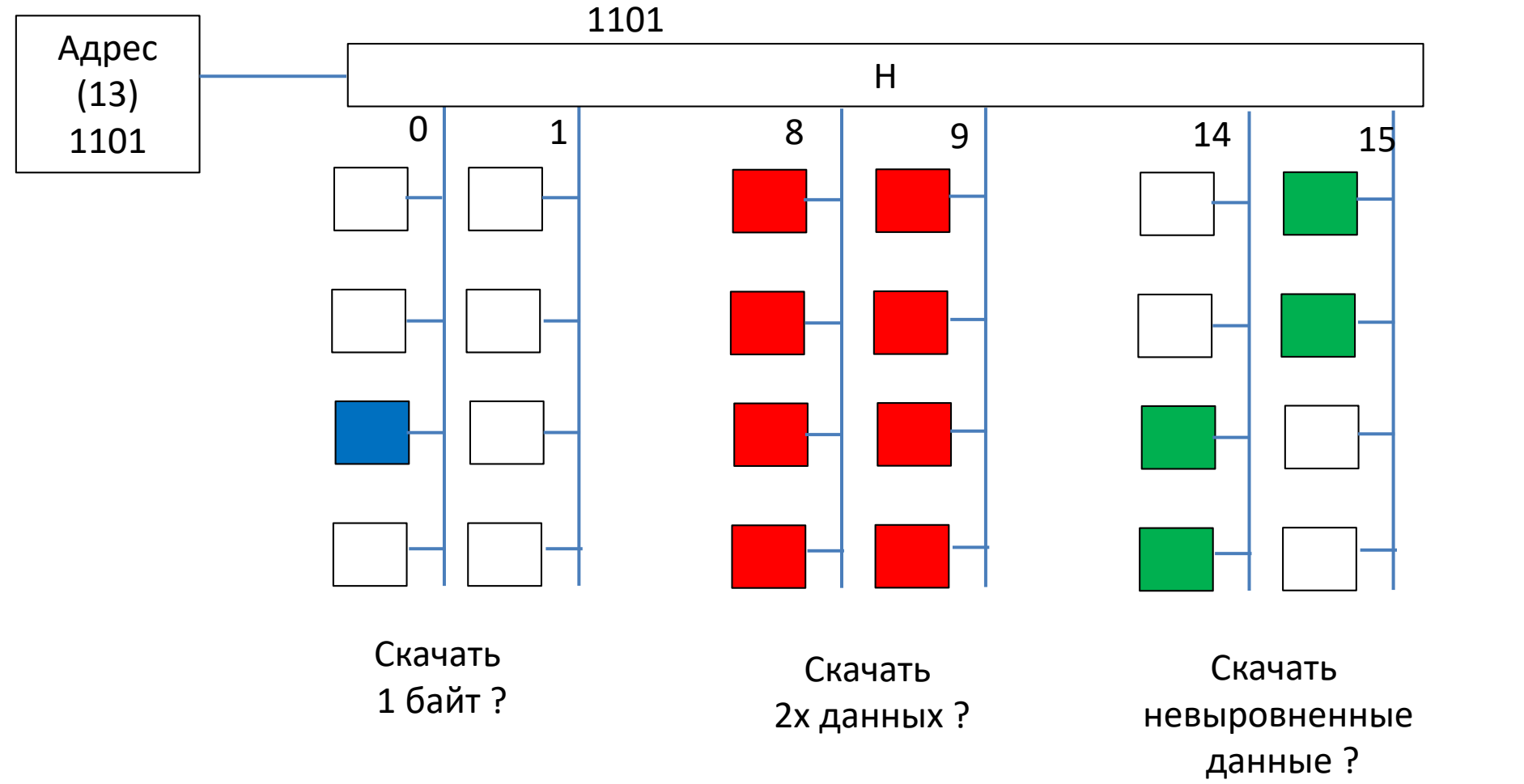


3. Программная архитектура процессора (машинное слово)



13 => 13*4 = 42

3. Программная архитектура процессора



3. Программная архитектура процессора

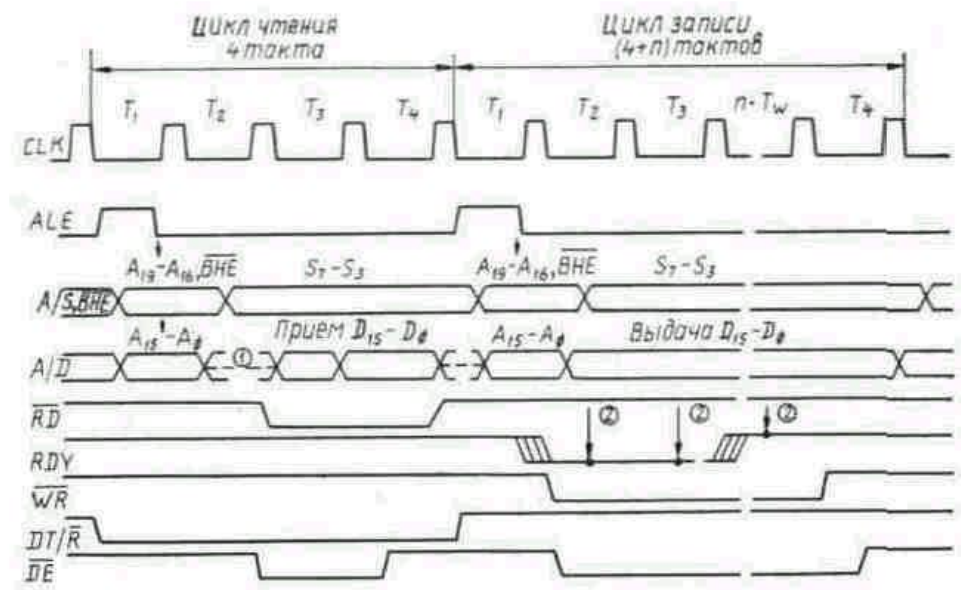
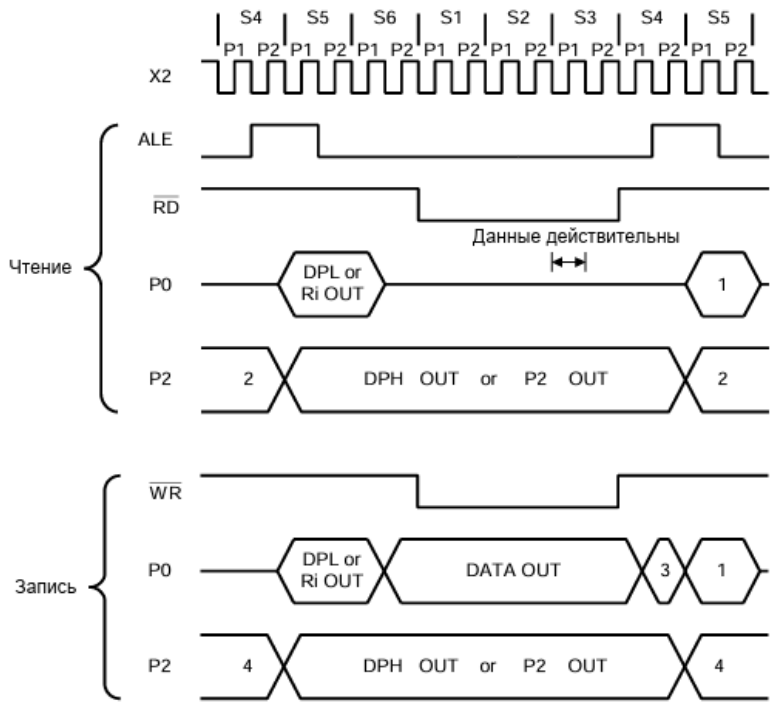


Рис. 2.8. Временная диаграмма циклов чтения и записи в системе с минимальной конфигурацией

4. Программная архитектура процессора

Память

- Что храним (данные, инструкции)

Фон-Неймановская архитектура

Гарвардская архитектура

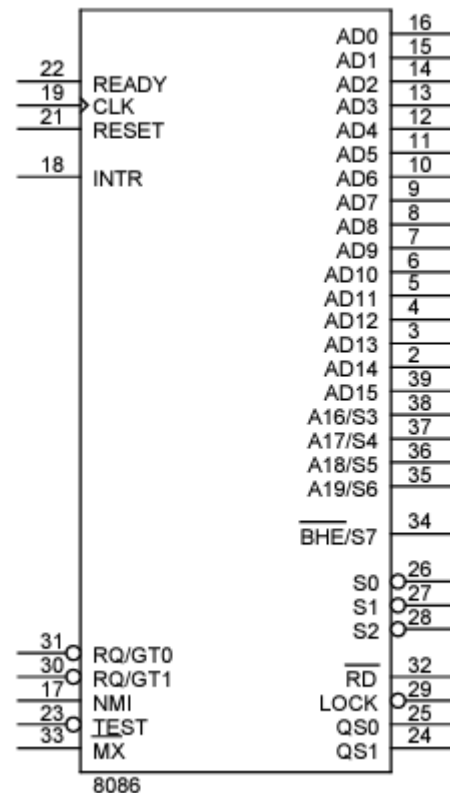
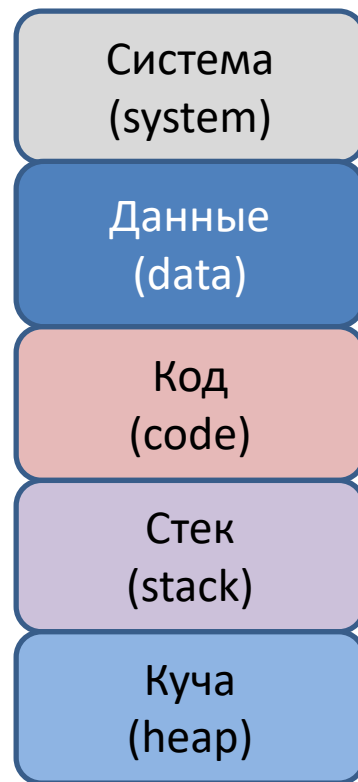
- Тип доступа (чтение, чтение-запись)

ROM, RAM, ПЗУ, ППЗУ

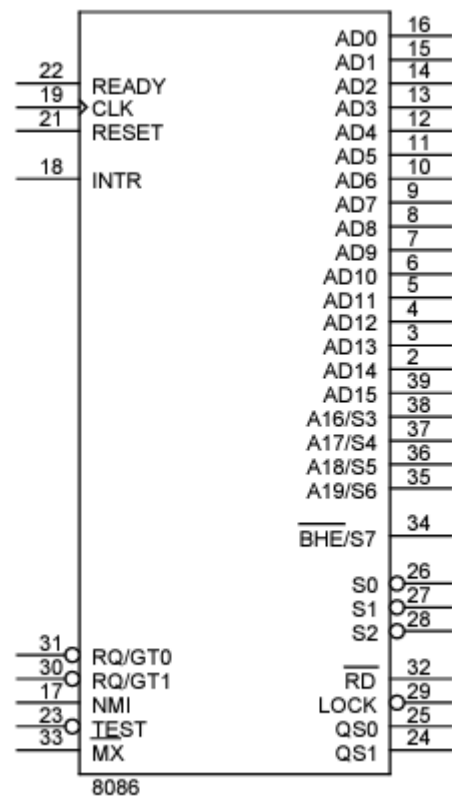
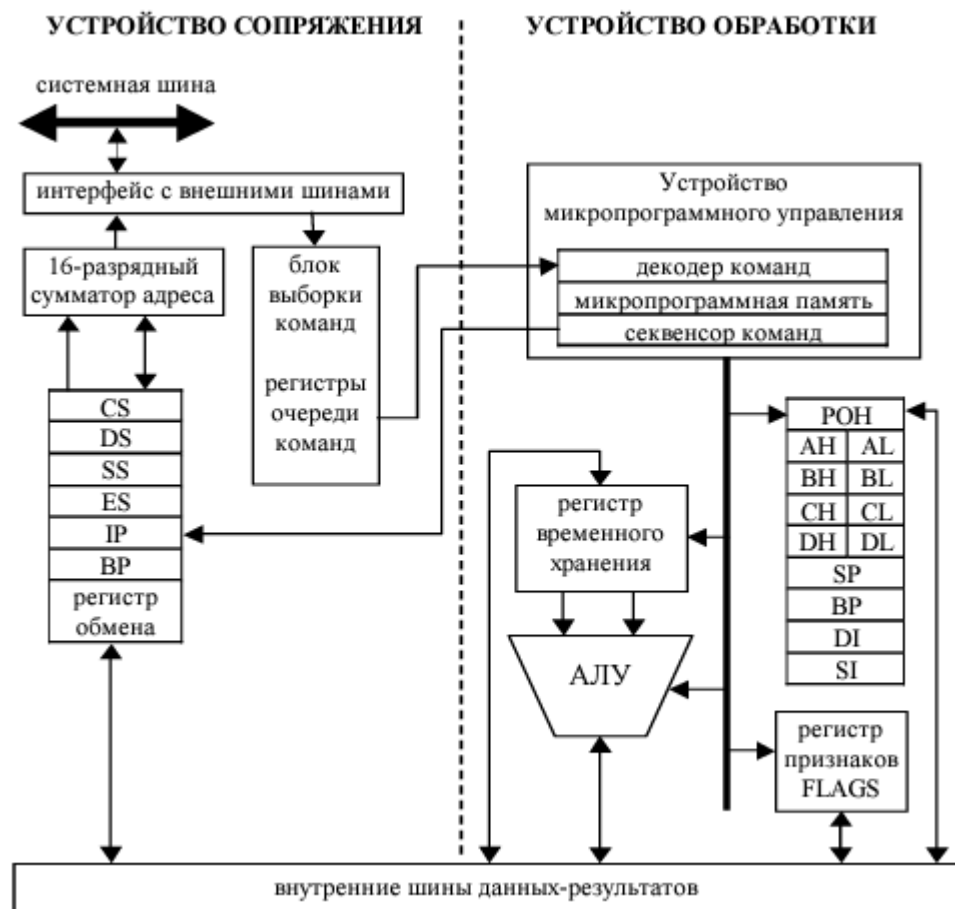
- Шина, адрес (размер), данные (размер)

- Сегменты

- Где храним код?



3. Программная архитектура процессора



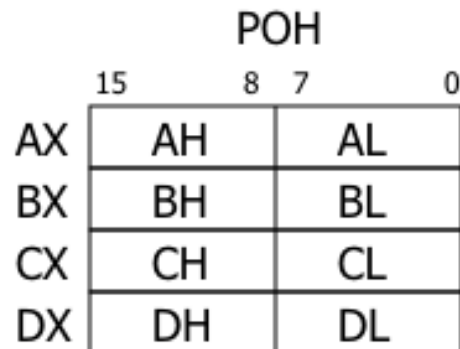
Для того, чтобы писать программы на ассемблере, нам необходимо знать, какие регистры процессора существуют и как их можно использовать.

Все процессоры архитектуры x86 (даже многоядерные, большие и сложные) являются дальними потомками древнего Intel 8086 и совместимы с его архитектурой.

Это значит, что программы на ассемблере 8086 будут работать и на всех современных процессорах x86.

Все внутренние регистры процессора Intel 8086 являются 16-битными:

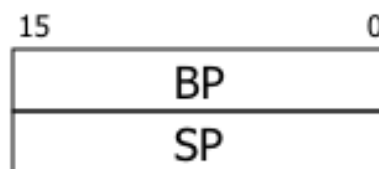
3. Программная архитектура процессора



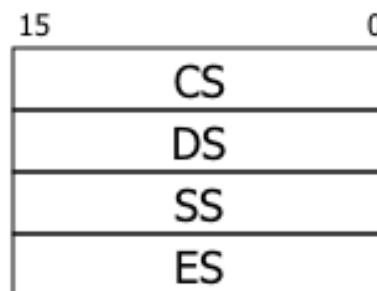
Индексные регистры



Регистры-указатели



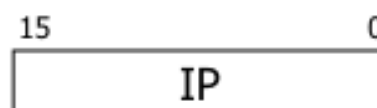
Сегментные регистры



Регистр флагов



Указатель команд



3. Программная архитектура процессора

Процессор 8086 имеет 14 шестнадцатиразрядных регистров, которые используются для управления исполнением команд, адресации и выполнения арифметических операций.

Регистры общего назначения.

К ним относятся 16-разрядные регистры AX, BX, CX, DX, каждый из которых разделен на 2 части по 8 разрядов:

- AX состоит из AH (старшая часть) и AL (младшая часть);
- BX состоит из BH и BL;
- CX состоит из CH и CL;
- DX состоит из DH и DL;

В общем случае функция, выполняемая тем или иным регистром, определяется командами, в которых он используется. При этом с каждым регистром связано некоторое стандартное его значение. Ниже перечисляются наиболее характерные функции каждого регистра:

Регистры общего назначения	
	15 8 7 0
AX	AH AL
BX	BH BL
CX	CH CL
DX	DH DL

Индексные регистры

15 0
SI
DI

Регистры-указатели

15 0
BP
SP

Сегментные регистры

15 0
CS
DS
SS
ES

Регистр флагов

15 0
FLAGS

Указатель команд

15 0
IP

3. Программная архитектура процессора

- **регистр AX** служит для временного хранения данных (регистр аккумулятор); часто используется при выполнении операций сложения, вычитания, сравнения и других арифметических и логических операции;
- **регистр BX** служит для хранения адреса некоторой области памяти (базовый регистр), а также используется как вычислительный регистр;
- **регистр CX** иногда используется для временного хранения данных, но в основном служит счетчиком; в нем хранится число повторений одной команды или фрагмента программы;
- **регистр DX** используется главным образом для временного хранения данных; часто служит средством пересылки данных между разными программными системами, в качестве расширителя аккумулятора для вычислений повышенной точности, а также при умножении и делении.
- **Регистры для адресации.** В микропроцессоре существуют четыре 16-битовых (2 байта или 1 слово) регистра, которые могут принимать участие в адресации операндов. Один из них одновременно является и регистром общего назначения — это регистр **BX**, или базовый регистр. Три другие регистра — это указатель базы **BP**, индекс источника **SI** и индекс результата **DI**. Отдельные байты этих трех регистров недоступны.

Любой из названных выше 4 регистров может использоваться для хранения адреса памяти, а команды, работающие с данными из памяти, могут обращаться за ними к этим регистрам. При адресации памяти базовые и индексные регистры могут быть использованы в различных комбинациях. Разнообразные способы сочетания в командах этих регистров и других величин называются способами или режимами адресации.

3. Программная архитектура процессора

Регистры сегментов. Имеются четыре регистра сегментов, с помощью которых память можно организовать в виде совокупности четырех различных сегментов. Этими регистрами являются:

- **CS** — регистр программного сегмента (сегмента кода) определяет местоположение части памяти, содержащей программу, т. е. выполняемые процессором команды;
- **DS** — регистр информационного сегмента (сегмента данных) идентифицирует часть памяти, предназначенной для хранения данных;
- **SS** — регистр стекового сегмента (сегмента стека) определяет часть памяти, используемой как системный стек;
- **ES** — регистр расширенного сегмента (дополнительного сегмента) указывает дополнительную область памяти, используемую для хранения данных.

Эти 4 различные области памяти могут располагаться практически в любом месте физической машинной памяти. Поскольку местоположение каждого сегмента определяется только содержимым соответствующего регистра сегмента, для реорганизации памяти достаточно всего лишь, изменить это содержимое.

3. Программная архитектура процессора

Регистр указателя стека. Указатель стека **SP** – это 16-битовый регистр, который определяет смещение текущей вершины стека. Указатель стека SP вместе с сегментным регистром стека SS используются микропроцессором для формирования физического адреса стека.

Стек всегда растёт в направлении меньших адресов памяти, т.е. когда слово помещается в стек, содержимое SP уменьшается на 2, когда слово извлекается из стека, микропроцессор увеличивает содержимое регистра SP на 2.

Стек (англ. stack — стопка) — абстрактный тип данных, представляющий собой список элементов, организованных по принципу **LIFO** (англ. last in — first out, «последним пришёл — первым вышел»).



4. Сегменты, данные, инструкции

```
section .data
section .text
section .bss

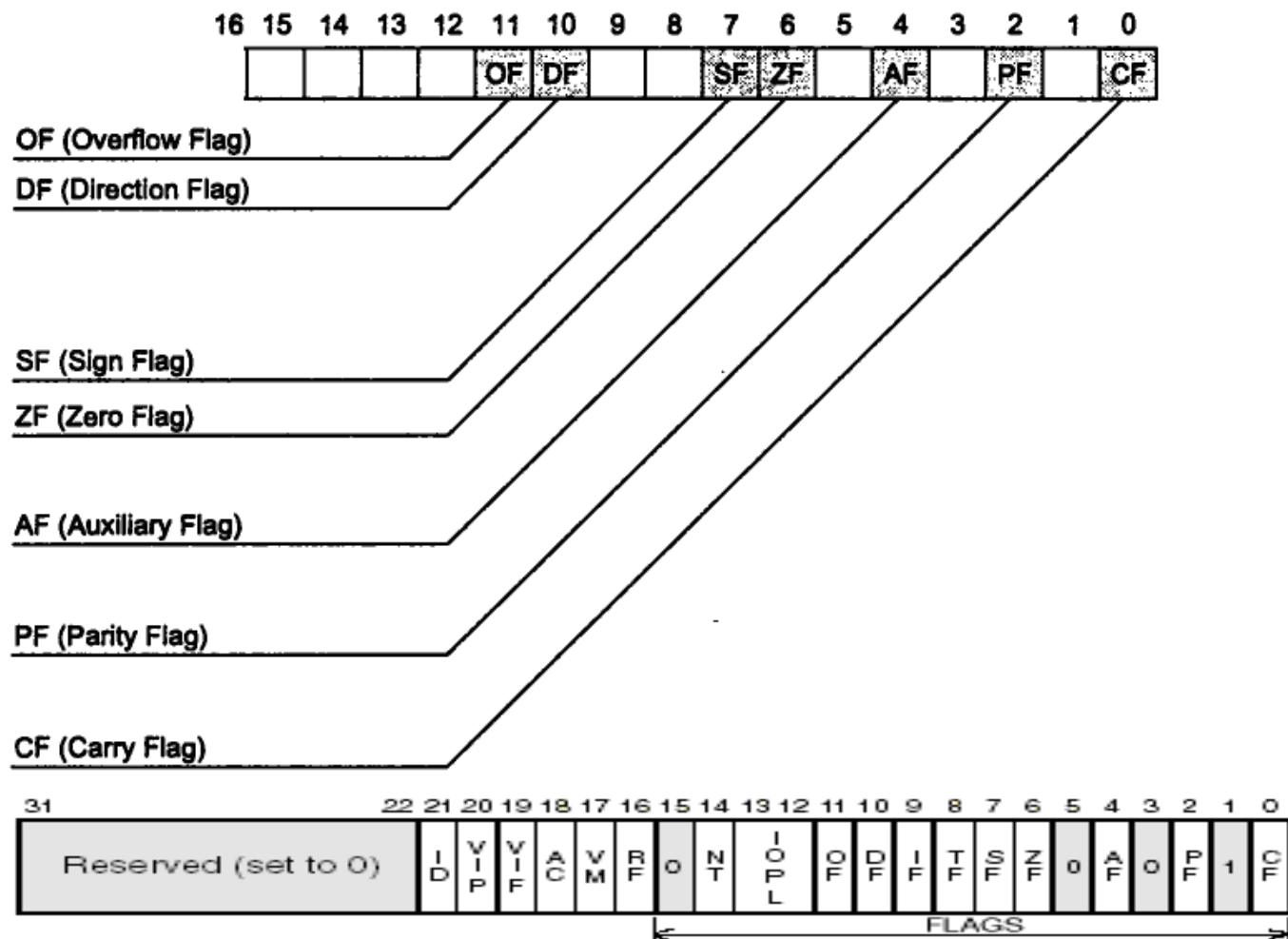
section .stack
section .rodata
```

```
.model tiny
CodeSg segment use16
    ASSUME ds:CodeSg, cs:CodeSg
    org 100h          ;Программа начинается с адреса 100h
start:
    mov ax, 4C00h    ;В регистр АХ помещаем 4Ch, в AL – 00h.
    int 21h          ;Завершение программы
    ;-----
    hello db 'Hello, world!$'
CodeSg ends
end start
```

3. Программная архитектура процессора

- **Регистр указателя команд IP.** Регистр указателя команд IP, иначе называемый регистром счетчика команд, имеет размер 16 бит и хранит адрес некоторой ячейки памяти – начало следующей команды. Микропроцессор использует регистр IP совместно с регистром CS для формирования 20-битового физического адреса очередной выполняемой команды, при этом регистр CS задает сегмент выполняемой программы, а IP – смещение от начала сегмента. По мере того, как микропроцессор загружает команду из памяти и выполняет ее, регистр IP увеличивается на число байт в команде. Для непосредственного изменения содержимого регистра IP служат команды перехода.
- **Регистр флагов.** Флаги – это отдельные биты, принимающие значение 0 или 1. Регистр флагов (признаков) содержит девять активных битов (из 16). Каждый бит данного регистра имеет особое значение, некоторые из этих бит содержат код условия, установленный последней выполненной командой. Другие биты показывают текущее состояние микропроцессора.

3. Программная архитектура процессора



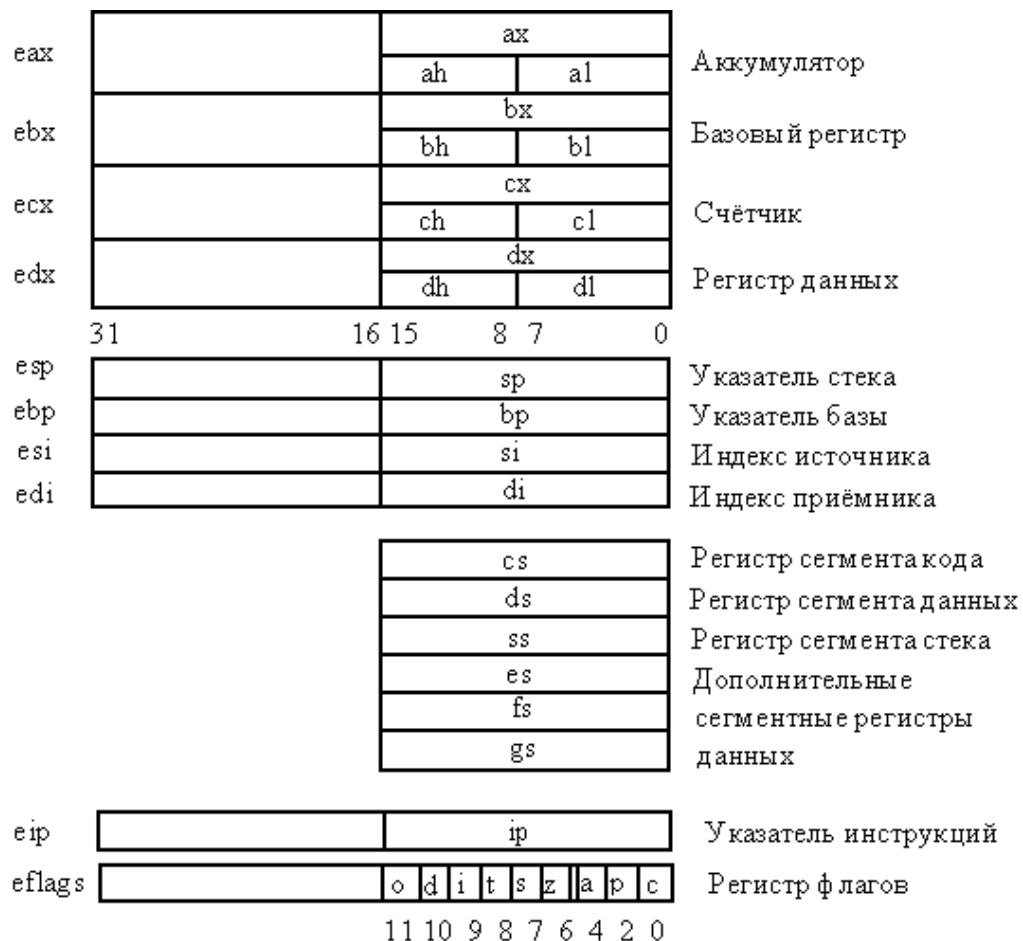
3. Программная архитектура процессора

Указанные на рисунке флаги наиболее часто используются в прикладных программах и сигнализируют о следующих событиях:

- OF (флаг переполнения) фиксирует ситуацию переполнения, то есть выход результата арифметической операции за пределы допустимого диапазона значений;
- DF (флаг направления) используется командами обработки строк. Если $DF = 0$, строка обрабатывается в прямом направлении, от меньших адресов к большим. Если $DF = 1$, обработка строк ведется в обратном направлении;
- SF (флаг знака) показывает знак результата операции, при отрицательном результате $SF = 1$;
- ZF (флаг нуля) устанавливается в 1, если результат операции равен 0;
- AF (флаг вспомогательного переноса) используется в операциях над упакованными двоично-десятичными числами. Этот флаг служит индикатором переноса или заема из старшей тетрады (бит 3);
- PF (флаг четности) устанавливается в 1, если результат операции содержит четное число двоичных единиц;
- CF (флаг переноса) показывает, был ли перенос или заем при выполнении арифметических операций.

Легко заметить, что все флаги младшего байта регистра флагов устанавливаются арифметическими или логическими операциями процессора. За исключением флага переполнения, все флаги старшего байта отражают состояние микропроцессора и влияют на характер выполнения программы. Флаги старшего байта устанавливаются и сбрасываются специально предназначенными для этого командами. Флаги младшего байта используются командами условного перехода для изменения порядка выполнения программы.

3. Программная архитектура процессора



3. Программная архитектура процессора

IP ([англ. Instruction Pointer](#)) — регистр, обозначающий смещение следующей команды относительно кодового сегмента.

IP — 16-битный (младшая часть EIP)

EIP — 32-битный аналог (младшая часть RIP)

RIP — 64-битный аналог

Сегментные регистры — Регистры указывающие на сегменты.

CS ([англ. Code Segment](#)), DS ([англ. Data Segment](#)), SS ([англ. Stack Segment](#)), ES, FS, GS

В [реальном режиме](#) работы процессора сегментные регистры содержат адрес начала 64Кб сегмента, смещенный вправо на 4 бита.

В [защищенном режиме](#) работы процессора сегментные регистры содержат селектор сегмента памяти, выделенного ОС.

CS — указатель на кодовый сегмент. Связка CS:IP (CS:EIP/CS:RIP — в защищенном/64-битном режиме) указывает на адрес в памяти следующей команды.

Регистры данных — служат для хранения промежуточных вычислений.

RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8 — R15 — 64-битные

EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, R8D — R15D — 32-битные (extended AX)

AX, CX, DX, BX, SP, BP, SI, DI, R8W — R15W — 16-битные

AH, AL, CH, CL, DH, DL, BH, BL, SPL, BPL, SIL, DIL, R8B — R15B — 8-битные (половинки 16-ти битных регистров)

например, AH — high AX — старшая половинка 8 бит

AL — low AX — младшая половинка 8 бит

RAX		RCX		RDX		RBX	
	EAX		ECX		EDX		EBX
	AX		CX		DX		BX
	AH AL		CH CL		DH DL		BH BL

RSP		RBP		RSI		RDI		R8	
	ESP		EBP		ESI		EDI		RxD
	SP		BP		SI		DI		RxW
	SPL		BPL		SIL		DIL		RxB

где x — 8..15.

Регистры RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W, SPL, BPL, SIL, DIL доступны только в 64-битном режиме работы процессора.

[Регистр флагов](#) FLAGS (16 бит) / EFLAGS (32 бита) / RFLAGS (64 бита) — содержит текущее состояние процессора.

4. Сегменты, данные, инструкции

```
section .data
```

```
    a dw 185
```

```
section .bss
```

```
    b resb
```

```
section .text
```

```
global _start
```

```
_start:
```

```
    mov ax,[a]
```

```
    not ax
```

```
    add ax,1
```

```
    mov eax,1
```

```
    mov ebx,0
```

```
    int 0x80
```

Данные

Неинициализированные
данные

Инструкции

Представление целых чисел в памяти

Представление отрицательных чисел в памяти

4. Сегменты, данные, инструкции

Для инициализации простых типов данных в Ассемблере используются специальные директивы **Dx**, являющиеся указаниями компилятору на выделение определенных объемов памяти с **инициализацией**. Для языка Ассемблера имеет значение только длина ячейки, в которой размещено данное, а какое это данное — зависит всецело от человека, пишущего программу его обработки.

Длина (бит)	Директива	Пример	Пример Си (*)
8	DB	<code>db 0</code> <code>Var1 db 5</code> <code>Var2 db 'a'</code> <code>Var3 db -1</code> <code>Var4 db 1.3</code> <code>MasQ DB 1, 2</code>	<code>char var1 = 5;</code> <code>unsigned char var2 = 'a';</code> <code>char var3 = -1;</code> <code>char MasQ[2] = {1, 2};</code>

4. Сегменты, данные, инструкции

Длина (бит)	Директива	Пример	Пример Си(*)
16	DW	<code>dw 0</code> <code>Var1 dw 5</code> <code>Var2 dw 'a'</code> <code>Var3 dw -1</code> <code>Var4 dw 1.3</code>	<code>short var1 = 5;</code> <code>unsigned short var2 = 'a';</code> <code>short var3 = -1;</code>
32	DD	<code>dd 0</code> <code>Var1 dd 5</code> <code>Var2 dd 'a'</code> <code>Var3 dd -1</code> <code>Var4 dd 1.3</code>	<code>int var1 = 5;</code> <code>unsigned int var2 = 'a';</code> <code>int var3 = -1;</code> <code>float var4 = 1.3;</code>

4. Сегменты, данные, инструкции

Длина (бит)	Директива	Пример	Пример Си (*)
64	DQ	<code>dq 0</code> <code>Var1 dq 5</code> <code>Var2 dq 'a'</code> <code>Var3 dq -1</code> <code>Var4 dq 1.3</code>	<code>long var1 = 5;</code> <code>unsigned long var2 = 'a';</code> <code>long = -1;</code> <code>double var4 = 1.3;</code>
80	DT	<code>dt 0</code> <code>Var1 dt 5</code> <code>Var2 dt 'a'</code> <code>Var3 dt -1</code> <code>Var4 dt 1.3</code>	<code>long double var4 = 1.3;</code>

4. Сегменты, данные, инструкции

(*) Важно!

В `asm` указание перед инструкцией `Dx` «имени переменной» не является именем переменной в понимании языка Си. Это адрес ячейки памяти, то есть это указатель!

```
MasQ DB 1, 2
```

```
char MasQ[2] = {1, 2};
```

4. Сегменты, данные, инструкции

Константы

```
mov ax, 200 ; decimal
mov ax, 0200 ; still decimal
mov ax, 0200d ; explicitly decimal
mov ax, 0d200 ; also decimal
mov ax, 0c8h ; hex
mov ax, $0c8 ; hex again: the 0 is required
mov ax, 0xc8 ; hex yet again
mov ax, 0hc8 ; still hex
mov ax, 310q ; octal
mov ax, 310o ; octal again
mov ax, 0o310 ; octal yet again
mov ax, 0q310 ; octal yet again
mov ax, 11001000b ; binary
mov ax, 1100_1000b ; same binary constant
mov ax, 1100_1000y ; same binary constant once more
mov ax, 0b1100_1000 ; same binary constant yet again
mov ax, 0y1100_1000 ; same binary constant yet again
```

```
db 'hello' ;           string constant
db 'h','e','l','l','o' ; equivalent character constants
```

And the following are also equivalent:

```
dd 'ninechars' ;           doubleword string constant
dd 'nine','char','s' ;     becomes three doublewords
db 'ninechars',0,0,0 ;     and really looks like this
```

```
db -0.2 ; "Quarter precision"
dw -0.5 ; IEEE 754r/SSE5 half precision
dd 1.2 ; an easy one
dd 1.222_222_222 ; underscores are permitted
dd 0x1p+2 ;  $1.0 \times 2^2 = 4.0$ 
dq 0x1p+32 ;  $1.0 \times 2^{32} = 4\,294\,967\,296.0$ 
dq 1.e10 ; 10 000 000 000.0
dq 1.e+10 ; synonymous with 1.e10
dq 1.e-10 ; 0.000 000 000 1
dt 3.141592653589793238462 ; pi
do 1.e+4000 ; IEEE 754r quad precision
```

Для хранения неинициализированных простых типов данных в Ассемблере используются специальные директивы **RESx**, являющиеся указаниями компилятору на выделение определенных объемов памяти **без инициализации**.

Директива	Пример	Пример Си
RESB	Var1 resb 5	char var1[5];
RESW	Var2 resw 5	short var2[5];
RESD	Var3 resd 5	int var3[5];
	...	

4. Сегменты, данные, инструкции

Инструкции управляют работой процессора, а директивы указывают ассемблеру и редактору связей, каким образом следует объединять инструкции для создания модуля, который и станет работающей программой.

Инструкция процессора на языке ассемблера состоит не более чем из четырех полей и имеет следующий формат:

[[метка:]] мнемоника [[операнды]] [[;комментарии]]

Единственное обязательное поле – **поле кода операции (мнемоника)**, определяющее инструкцию, которую должен выполнить микропроцессор. Поле операндов определяется кодом операции и содержит дополнительную информацию о команде. Каждому коду операции соответствует определенное число **операндов**.

Операнды представляют значения, регистры или адреса ячеек памяти, используемых определенным образом по контексту программы.

4. Сегменты, данные, инструкции

Метка служит для обозначения какого-то определенного места в памяти, т. е. содержит в символическом виде адрес, по которому храниться инструкция. Преобразование символических имен в действительные адреса осуществляется программой ассемблера. Часть строки исходного текста после символа «;» (если он не является элементом знаковой константы или строки знаков) считается **комментарием** и ассемблером игнорируется.

Пример:

Метка	Код операции	Операнды	;Комментарий
МЕТ:	MOV	AX, BX	;Пересылка

4. Сегменты, данные, инструкции

MOV r, r	2	1 0 0 0 1 0 d w	mod reg r/m	Pr←Pr
MOV r, mem	8+E			Pr←П
MOV mem, r	9+E			П←Pr
MOV mem, data	10+E	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	П←Д
		data L	data H(w=1)	
MOV r, data	4	1 0 1 1 w reg	data L	Pr←Д
		data H(w=1)		
MOV a, mem	10	1 0 1 0 0 0 d w	addr L	A←П
		addr H		
MOV mem, a	10			П←A

Integer instructions

Instruction	Operands	Ops	Latency	Reciprocal throughput	E u
Move instructions					
MOV	r,r	1	1	1/3	
MOV	r,i	1	1	1/3	
MOV	r8,m8	1	4	1/2	
MOV	r16,m16	1	4	1/2	
MOV	r32,m32	1	3	1/2	
MOV	m8,r8H	1	8	1/2	

4. Сегменты, данные, инструкции

```
1                                     section .data
2 00000000 B900                      a dw 185

3                                     section .text
4 global _start
5 _start:
6 00000000 668B0425[00000000]        mov ax,[a]
7 00000008 66F7D0                    not ax
8 0000000B 6683C001                 add ax,1
9
10 0000000F B801000000              mov eax,1
11 00000014 BB00000000              mov ebx,0
12 00000019 CD80                   int 0x80
```

5. Арифметические операции

- Арифметические операции
 - Перемещение данных
 - Сложение
 - Вычитание
 - Умножение
 - Деление

5. Арифметические операции. Перемещение данных

- MOV
- POP/PUSH,...
- XCNG
- CBW/CWD,...
- MOVSX
- MOVZX

5. Арифметические операции. Перемещение данных

MOV r, r	2	1 0 0 0 1 0 d w	mod reg r/m	$Pr \leftarrow Pr$
MOV r, mem	$8+E$			$Pr \leftarrow П$
MOV mem, r	$9+E$			$П \leftarrow Pr$
MOV mem, data	$10+E$	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	$П \leftarrow Д$
		data L	data H (w=1)	
MOV r, data	4	1 0 1 1 w reg	data L	$Pr \leftarrow Д$
		data H (w=1)		
MOV a, mem	10	1 0 1 0 0 0 d w	addr L	$A \leftarrow П$
		addr H		
MOV mem, a	10			$П \leftarrow A$

MOV seg, r	2	1 0 0 0 1 1 d 0	mod 0 seg r/m	Сегментный $Pr \leftarrow Pr$
MOV seg, mem	$8+E$			Сегментный $Pr \leftarrow П$
MOV r, seg	2			$Pr \leftarrow \text{сегментный } Pr$
MOV mem, seg	$9+E$			$П \leftarrow \text{сегментный } Pr$

5. Арифметические операции. Перемещение данных

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
88 /r	MOV r/m8,r8	MR	Valid	Valid	Move r8 to r/m8.
REX + 88 /r	MOV r/m8 ^{***} ,r8 ^{***}	MR	Valid	N.E.	Move r8 to r/m8.
89 /r	MOV r/m16,r16	MR	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32,r32	MR	Valid	Valid	Move r32 to r/m32.
REX.W + 89 /r	MOV r/m64,r64	MR	Valid	N.E.	Move r64 to r/m64.
8A /r	MOV r8,r/m8	RM	Valid	Valid	Move r/m8 to r8.
REX + 8A /r	MOV r8 ^{***} ,r/m8 ^{**}	RM	Valid	N.E.	Move r/m8 to r8.
8B /r	MOV r16,r/m16	RM	Valid	Valid	Move r/m16 to r16.
8B /r	MOV r32,r/m32	RM	Valid	Valid	Move r/m32 to r32.
REX.W + 8B /r	MOV r64,r/m64	RM	Valid	N.E.	Move r/m64 to r64.

5. Арифметические операции. Перемещение данных

section .data

ch1 db 0x11;	uint8_t ch1 = 0x11	void * ch1
ch2 db 0x22;	uint8_t ch2 = 0x22	void * ch2
sh1 dw 0xAABB;	uint16_t sh1 = 0xAABB	void * sh1
myint dd 0;		void * myint

section .text

global CMAIN

CMAIN:

```
xor eax, eax
mov al, [ch1]    ; uint8_t al = *(uint8_t*)ch1; al = 0x11
mov ax, [ch1]    ; uint16_t ax = *(uint16_t*)ch1; ax = 0x2211
mov eax, [ch1]   ; uint32_t eax = *(uint32_t*)ch1; eax = 0xAABB2211
mov [myint], eax ; *myint = 0x600042
mov eax, ch1     ; uint32_t eax = ch1; eax = 0x600042 – адрес!!!
```

```
mov ax, 3
```

5. Арифметические операции. Перемещение данных

```
section .data
    myint dd 0x11223344
section .text
global CMAIN
CMAIN:
    xor eax, eax
    mov ax, [myint]           ; ax == 0x3344
    mov bx, [myint + 2]       ; bx == 0x1122
    mov cl, [myint]           ; cl == 0x44
    mov ch, [myint + 1]       ; ch == 0x33
    mov dl, [myint + 2]       ; dl == 0x22
    mov dh, [myint + 3]       ; dh == 0x11
    ret
```

5. Арифметические операции. Перемещение данных

PUSH r	10	1 1 1 1 1 1 1 1	mod 1 1 0 r/m	Стек←Pг
PUSH mem	16+E			Стек←П
PUSH r	10	0 1 0 1 0 reg		Стек←Pг
PUSH seg	10	0 0 0 seg 1 1 0		Стек←сегментный Pг
POP r	8	1 0 0 0 1 1 1 1	mod 0 0 0 r/m	Pг←стек
POP mem	17+E			П←стек
POP r	8	0 1 0 1 1 reg		Pг←стек
POP seg	8	0 0 0 seg 1 1 1		Сегментный Pг←стек

```
section .data
```

```
    a dq 1
```

```
    b dq 2
```

```
section .text
```

```
global CMAIN
```

```
CMAIN:
```

```
    push qword [a]
```

```
    pop qword [b]
```

```
    ret
```

5. Арифметические операции. Перемещение данных

```
char var1 = 4;  
short var2 = va1; ????
```

```
CBW      ; ax <- al  
CWD      ; dx:ax <- ax  
CDQ      ; edx:eax <- eax
```

```
CWDE     ; eax <- ax  
CDQE     ; rax <- eax
```

```
MOVZX    ; для беззнаковых, заполнение 0
```

```
MOVSX    ; для знаковых, учет знака
```

5. Арифметические операции. Перемещение данных

```
section .data
```

```
  b db 55
```

```
  c db -10
```

```
section .text
```

```
global CMAIN
```

```
CMAIN:
```

```
  movsx bx, [c]      ; bx = c
```

```
  movzx ax, [b]      ; ax = b
```

```
  mov al, [c]
```

```
  cbw
```

```
  mov bx, ax
```

```
  mov al, [b]
```

```
  cbw
```

```
  ret
```

5. Арифметические операции. Сложение

Команды сложения ADD, ADC, INC.

Командам **БЕЗРАЗЛИЧНО** какие числа складываются (**знаковые или нет**).

Если в результате сложения результат **НЕ** поместился в отведенное место, устанавливается флаг переноса **CF=1**. Команда **ADC** как раз и реагирует на этот флаг. Вырабатываются еще **4** флага: **PF, SF, ZF, OF**.

5. Арифметические операции. Сложение

Команда **ADD** (ADDition)

Назначение: сложение двух операндов источник и приемник размерностью байт, слово или двойное слово.

Синтаксис: **ADD Приемник, Источник**

Алгоритм работы:

- сложить операнды источник и приемник;
- записать результат сложения в приемник;
- установить флаги.

Логика работы: **<Приемник> = < Приемник> + <Источник>**

Применение: Команда **add** используется для сложения двух целочисленных операндов. Результат сложения помещается по адресу первого операнда. Если результат сложения выходит за границы операнда приемник (возникает переполнение), то учесть эту ситуацию следует путем анализа флага **cf** и последующего возможного применения команды **adc**.

5. Арифметические операции. Сложение

Команда **ADC** (Addition with Carry)

Назначение: сложение двух операндов с учетом переноса из младшего разряда.

Синтаксис: **ADC Приемник, Источник**

Алгоритм работы:

- сложить два операнда;
- поместить результат в первый операнд;
- в зависимости от результата установить флаги.

Логика работы: **<Приемник> = <Приемник> + <Источник> + < CF >**

Применение: Команда **adc** используется при сложении длинных двоичных чисел. Ее можно использовать как самостоятельно, так и совместно с командой **add**. При совместном использовании команды **adc** с командой **add** сложение младших байтов/слов/двойных слов осуществляется командой **add**, а уже старшие байты/слова/двойные слова складываются командой **adc**, учитывающей переносы из младших разрядов в старшие. Таким образом, команда **adc** значительно расширяет диапазон значений складываемых чисел.

5. Арифметические операции. Сложение

$a = b + c$

$a = 10 + 22 = 32$ (0x20)

$a = b + c$

$al == al + bl$

```
section .data
```

```
b db 10
```

```
c db 22
```

```
a db 0
```

```
section .text
```

```
global CMAIN
```

```
CMAIN:
```

```
    mov bl, [c]        ; bl = c
```

```
    mov al, [b]        ; al = b
```

```
    add al, bl         ; al = al + bl = b + c
```

```
    mov [a], al        ; a = b + c
```

```
ret
```

5. Арифметические операции. Сложение

$$a = b + c$$

$$a = 255 + 255 = 510 \text{ (0x01FE)}$$

$$a = b + c$$

$$ax == ax + bx$$

```
section .data
```

```
  b db 255
```

```
  c db 255
```

```
  a dw 0
```

```
section .text
```

```
global CMAIN
```

```
CMAIN:
```

```
  movzx bx, [c]      ; bx = c
```

```
  movzx ax, [b]      ; ax = b
```

```
  add ax, bx          ; ax = ax + bx = b + c
```

```
  mov [a], ax         ; a = b + c
```

```
ret
```

5. Арифметические операции. Вычитание

Команды вычитания **SUB**, **SBB**, **DEC** ОБРАТНЫ соответствующим командам сложения **ADD**, **ADC** и **INC**.

Они имеют те же самые операнды.

SUB (SUBtract — Вычитание).

SBB (SuBtract with Borrow CF — Вычитание с заемом флага переноса CF).

DEC (DECrement operand by 1 — Уменьшение значения операнда на 1).

5. Арифметические операции. Вычитание

$$a = b - c$$

$$a = -100 - 100 = -200 \text{ (0xFF38)}$$

$$a = b - c$$

$$ax == ax - bx$$

```
section .data
    b db -100
    c db 100
    a dw 0
section .text
global CMAIN
CMAIN:
    mov al, [c]
    cbw                ; ax = c
    mov bx, ax          ; bx = c
    mov al, [b]
    cbw                ; ax = b
    sub ax, bx          ; ax = ax - bx = b - c
    mov [a], ax         ; a = b - c
    ret
```

5. Арифметические операции. Вычитание

CMP r1, r2	3	0 0 1 1 1 0 d w	mod reg r/m	Pr—Pr
CMP r, mem	9+E			Pr—П
CMP mem, r	16+E			П—Pr
CMP r, data	4	1 0 0 0 0 0 s w	mod 1 1 1 r/m	Pr—Д
		data L	data H(w=1)	
CMP mem, data	17+E			П—Д
CMP a, data	4	0 0 1 1 1 1 0 w	data L	А—Д
		data H(w=1)		

5. Арифметические операции. Умножение

MUL (MULTiply) - беззнаковое умножение

Назначение: операция умножения двух целых чисел без учета знака.

Синтаксис: **MUL сомножитель_1**

Алгоритм работы: Команда выполняет умножение двух целочисленных сомножителей. Один из сомножителей указан в качестве операнда. Расположение второго сомножителя указано НЕЯВНО и зависит от размера первого сомножителя.

- если операнд, указанный в команде — байт, то второй сомножитель должен располагаться в **AL**;
- если операнд, указанный в команде — слово, то второй сомножитель должен располагаться в **AX**;
- если операнд, указанный в команде — двойное слово, то второй сомножитель должен располагаться в **EAX**.

5. Арифметические операции. Умножение

От размера первого сомножителя также зависит размер результата. Результат умножения помещается также в фиксированное место, определяемое размером сомножителей:

- при умножении байтов результат помещается в AX;
- при умножении слов результат помещается в пару DX:AX;
- при умножении двойных слов результат помещается в пару EDX:EAX.

Логика работы: $\text{<Произведение>} = \text{<Сомножитель_1>} * \text{<Сомножитель_2>}$

Длина Произведения всегда в ДВА раза больше, чем у Множителя. Причем старшая часть Произведения находится либо в регистре AH, либо в DX.

Эти команды тоже вырабатывают флаги. Устанавливаются флаги CF=1 и OF=1, если результат слишком велик для отведенных ему регистров назначения.

5. Арифметические операции. Умножение

IMUL (Integer MULtiply) - умножение с учетом знака

Назначение: операция умножения двух целочисленных двоичных значений со знаком.

В отличие от MUL, команда может иметь 3 различных вида.

Синтаксис:

IMUL сомножитель_1

IMUL сомножитель_1, сомножитель_2

IMUL результат, сомножитель_1, сомножитель_2

Алгоритм работы: Алгоритм работы команды зависит от используемой формы команды. Форма команды с одним операндом требует явного указания местоположения только одного сомножителя, который может быть расположен в ячейке памяти или регистре. Местоположение второго сомножителя фиксировано и зависит от размера первого сомножителя:

- если операнд, указанный в команде, — байт, то второй сомножитель располагается в AL;
- если операнд, указанный в команде, — слово, то второй сомножитель располагается в AX;
- если операнд, указанный в команде, — двойное слово, то второй сомножитель располагается в EAX.

5. Арифметические операции. Умножение

Результат умножения для команды с одним операндом также помещается в строго определенное место, определяемое размером сомножителей:

- при умножении байтов результат помещается в AX;
- при умножении слов результат помещается в пару DX:AX;
- при умножении двойных слов результат помещается в пару EDX:EAX.

Команды с двумя и тремя операндами однозначно определяют расположение результата и сомножителей следующим образом:

- в команде с двумя операндами первый операнд определяет местоположение первого сомножителя. На его место впоследствии будет записан результат. Второй операнд определяет местоположение второго сомножителя;
- в команде с тремя операндами первый операнд определяет местоположение результата, второй операнд — местоположение первого сомножителя, третий операнд может быть непосредственно заданным значением размером в байт, слово или двойное слово.

Логика работы: $\text{<Произведение>} = \text{<Сомножитель_1>} * \text{<Сомножитель_2>}$

5. Арифметические операции. Умножение

$a = b * c$

$ax == al * bl$

```
section .data
```

```
b db -100
```

```
c db 100
```

```
a dw 0
```

```
section .text
```

```
global CMAIN
```

```
CMAIN:
```

```
    xor eax, eax
```

```
    xor ebx, ebx
```

```
    mov bl, [c]
```

```
    mov al, [b]
```

```
    imul bl                ; ax = al * bl
```

```
    mov [a], ax           ;
```

```
    ; mov [a], al
```

```
    ; mov [a + 1], ah
```

```
ret
```

5. Арифметические операции. Умножение

$a = b * c$

$dx:ax == ax * bx$

```
section .data
```

```
b dw -100
```

```
c dw 100
```

```
a dd 0
```

```
section .text
```

```
global CMAIN
```

```
CMAIN:
```

```
xor eax, eax
```

```
xor ebx, ebx
```

```
mov bx, [c]
```

```
mov ax, [b]
```

```
imul bx
```

```
; dx:ax = ax * bx
```

```
mov [a], ax
```

```
mov [a + 2], dx
```

```
ret
```

5. Арифметические операции. Умножение

$a = b * c$

$cx == cx * bx$

```
section .data
```

```
b dw 1000
```

```
c dw 1000
```

```
a dw 0
```

```
section .text
```

```
global CMAIN
```

```
CMAIN:
```

```
xor ecx, ecx
```

```
xor ebx, ebx
```

```
mov bx, [c]
```

```
mov cx, [b]
```

```
imul cx, bx ;
```

```
mov [a], cx
```

```
ret
```

$cx = cx * bx$, старшая часть игнорируется

5. Арифметические операции. Умножение

$a = b * \text{const}$

$dx == cx * X$

```
section .data
```

```
b dw 1000
```

```
a dw 0
```

```
section .text
```

```
global CMAIN
```

```
CMAIN:
```

```
xor ecx, ecx
```

```
xor edx, edx
```

```
mov cx, [b]
```

```
imul dx, cx, 4;
```

$dx = cx * 4$, старшая часть игнорируется

```
mov [a], dx
```

```
ret
```

5. Арифметические операции. Умножение двух больших чисел

1.	MOV EAX, DWORD [X]				
2.	MOV EBX,EAX	; сохраняем для шага 6			
3.	MUL DWORD [Y]	; перемножить младшие двойные слова			
4.	MOV DWORD [Z], EAX	; сохранить младшее слово произведения			
5.	MOV ECX,EDX	; сохранить старшее двойное слово			
6.	MOV EAX,EBX	; младшее слово "X" в eax			
7.	MUL DWORD [Y+4]	; умножить младшее слово на старшее			
8.	ADD EAX, ECX		1 2		[L:1] AX = 2
9.	ADC EDX,0	; добавить перенос	3 4		[L:2] BX = 2
10.	MOV EBX,EAX	; сохранить частичное произведение	0 8		[L:3-5] CX = DX = 0, AX = 8 (2*4)
11.	MOV ECX,EDX		0 6		[L:7] DX = 0, AX = 6 (2*3)
12.	MOV EAX,DWORD [X+4]		0 6	(AX+ CX)	[L:8](0 6 + 0 0)
13.	MUL DWORD [Y]	; умножить старшее слово на младшее	0 6	(+ CF)	[L:9] CX = DX = 0, BX = AX = 6
14.	ADD EAX,EBX	; сложить с частичным произведением	0 4		[L:13] DX = 0, AX = 4 (1 * 4)
15.	MOV DWORD [Z+4],EAX		0 A	(AX + BX)	[L:14-15]
16.	ADC ECX,EDX		0 A	(+ CF)	[L:16] CX = 0
17.	MOV EAX,DWORD [X+4]		0 3		[L:18] DX = 1, AX = 3 (1*3)
18.	MUL DWORD [Y+4]	; умножить старшие слова	0 3	(AX + CX)	[L:19]
19.	ADD EAX,ECX	; сложить с частичным произведением	0 3	(+ CF)	[L:20]
20.	ADC EDX,0	; и добавить перенос	0 3 A 8		[L:21-22]
21.	MOV DWORD [Z+8],EAX				
22.	MOV DWORD [Z+12],EDX				
23.	ret				

section .data
X DQ 0x00000000100000002
Y DQ 0x00000000300000004
Z DQ 0, 0

0 – результат умножения, 6 – результат сложения
Жирный шрифт (8, A, 0, 3) – сохранение результата [L:20] – номер строки кода

5. Арифметические операции. Деление

Команды деления **DIV** и **IDIV**

Назначение:

DIV выполнение операции деления двух двоичных беззнаковых значений.

IDIV операция деления двух двоичных значений со знаком.

Синтаксис: **DIV делитель**

IDIV делитель

Алгоритм работы: Для команды необходимо задание двух операндов — делимого и делителя. Делимое задается неявно и размер его зависит от размера делителя, который указывается в команде:

- если делитель размером в байт, то делимое должно быть расположено в регистре `ax`. После операции частное помещается в `AL`, а остаток — в `AH`;
- если делитель размером в слово, то делимое должно быть расположено в паре регистров `DX:AX`, причем младшая часть делимого находится в `AX`. После операции частное помещается в `ax`, а остаток — в `DX`;
- если делитель размером в двойное слово, то делимое должно быть расположено в паре регистров `EDX:EAX`, причем младшая часть делимого находится в `EAX`. После операции частное помещается в `EAX`, а остаток — в `EDX`.

Логика работы: **< Частное: Остаток > = <Делимое> / <Делитель>**

5. Арифметические операции. Деление

Команда **IDIV** реагирует на ЗНАК обрабатываемых чисел.

Результат состоит из Частного и Остатка, которым при обычных целочисленных вычислениях пренебрегают.

Длина Делимого всегда в ДВА раза больше, чем у Делителя. Причем старшая часть Делимого находится либо в регистре AH, либо в DX.

Эти команды тоже вырабатывают флаги. Но устанавливаются флаги **CF=1** и **OF=1**, если частное НЕ помещается в регистры AL или AX.

Здесь, в отличие от команд умножения, может генерироваться ПРЕРЫВАНИЕ "Деление на ноль".

*Применение: Команда выполняет целочисленное деление операндов с выдачей результата деления в виде частного и остатка от деления. При выполнении операции деления возможно возникновение исключительной ситуации: 0 — ошибка деления. Эта ситуация возникает в одном из двух случаев: делитель равен 0 или частное слишком велико для его размещения в регистре **eax/ax/al**.*

5. Арифметические операции. Деление

$$a = b / c$$

$dx:ax / bx \Rightarrow ax * dx / bx$, dx - остаток

```
section .data
```

```
b dw 100
```

```
c dw 48
```

```
a dw 0
```

```
section .text
```

```
global CMAIN
```

```
CMAIN:
```

```
mov bx, [c]
```

```
mov ax, [b]
```

```
cwd
```

```
div bx           ; dx:ax = ax / bx
```

```
                ; ax = 2, dx = 4
```

```
mov [a], ax      ; a = 2
```

```
ret
```

5. Арифметические операции. Сдвиг

Всего существует 8 команд сдвига: 4 команды обычного сдвига и 4 команды циклического сдвига.

команды логического сдвига вправо **SHR** и влево **SHL**;

команды арифметического сдвига вправо **SAR** и влево **SAL**;

команды циклического сдвига вправо **ROR** и влево **ROL**;

команды циклического сдвига вправо **RCR** и влево **RCL** с переносом;

Команды циклического сдвига переносят появляющийся в конце операнда бит в другой конец, а в случае обычного сдвига этот бит пропадает.

5. Арифметические операции. Сдвиг

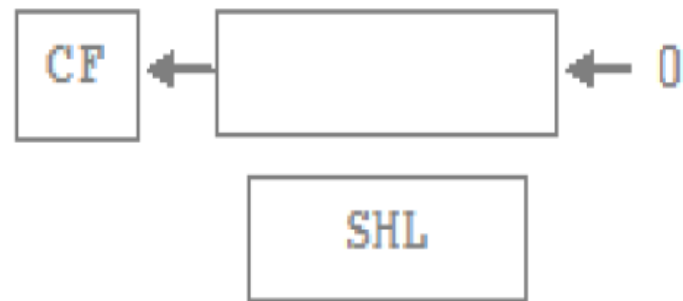


MOV CL, 03 AX:
MOV AX, 10110111B ; **10110111**
SHR AX, 1 ; **01011011** ; Сдвиг вправо на 1 , CF = 1
SHR AX, CL ; **00001011** ; Сдвиг вправо на 3, CF = 0

Первая команда SHR сдвигает содержимое регистра AX вправо на 1 бит. Выдвинутый в результате один бит попадает в флаг CF, а самый левый бит регистра AX заполняется нулем.

Вторая команда сдвигает содержимое регистра AX еще на три бита. При этом флаг CF последовательно принимает значения 1,1, 0, а в три левых бита в регистре AX заносятся нули.

5. Арифметические операции. Сдвиг



```
MOV CL, 04           AX:
MOV AX, 10110111B ; 10110111
SHL AX, 1             ; 01101110 ; Сдвиг влево на 1 , CF = 1
SHL AX, CL            ; 11100000 ; Сдвиг влево на 4, CF = 0
```

6. Логические операции

Логические операции являются важным элементом в проектировании микросхем и имеют много общего в логике программирования.

Команды **AND**, **OR**, **XOR**, **NOT** и **TEST** - являются командами логических операций.

Эти команды используются для сброса и установки отдельных бит.

Все эти команды обрабатывают один байт или одно слово в регистре или в памяти, и устанавливают флаги CF, OF, PF, SF, ZF.

AND: Если оба из сравниваемых битов равны 1, то результат равен 1; во всех остальных случаях результат - 0.

OR: Если хотя бы один из сравниваемых битов равен 1, то результат равен 1; если сравниваемые биты равны 0, то результат - 0.

XOR: Если один из сравниваемых битов равен 0, а другой равен 1, то результат равен 1; если сравниваемые биты одинаковы (оба - 0 или оба - 1) то результат - 0.

TEST: действует как **AND-устанавливает** флаги, но не изменяет биты.

6. Логические операции

Первый операнд в логических командах указывает на один байт или слово в регистре или в памяти и является единственным значением, которое может изменяться после выполнения команд.

Пример:	AND	OR	XOR
	0101	0101	0101
	0011	0011	0011
Результат:	0001	0111	0110

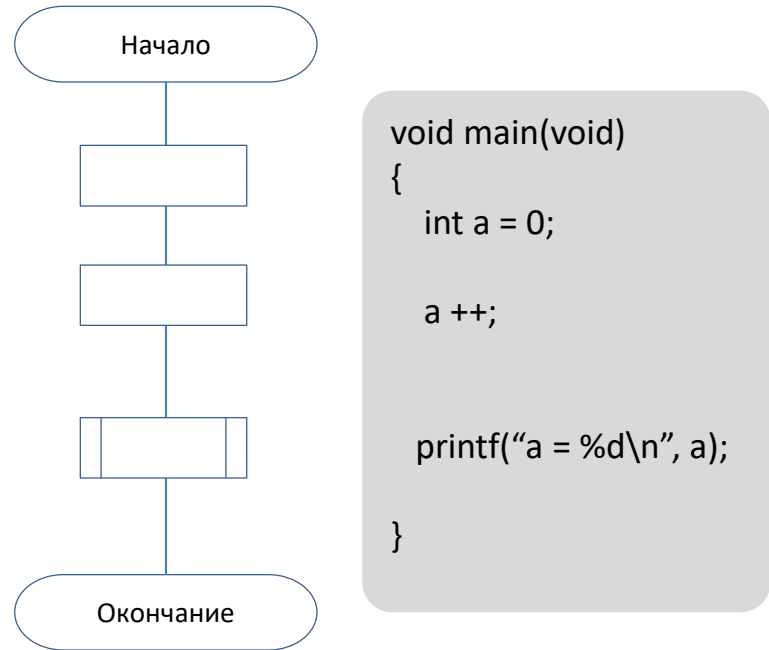
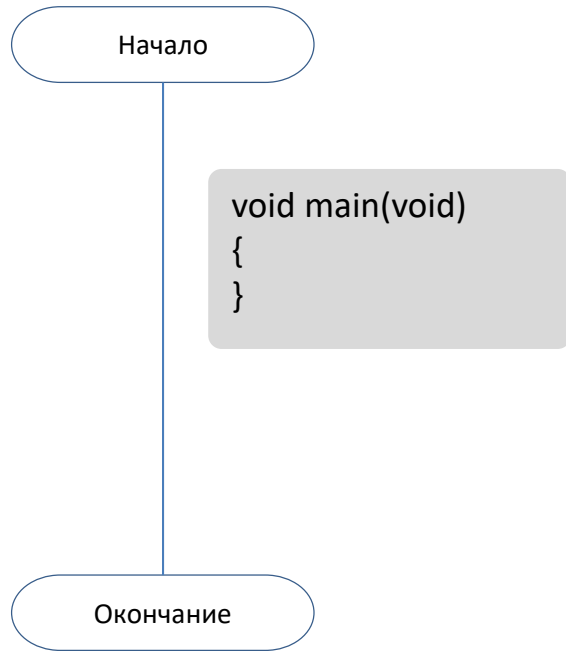
NOT – побитная инверсия

MOV ax, 0xAA55;
NOT ax ; AX = 0x55AA

```
mov dl, 5 ; Двоичное число  
  
OR dl, 30h ; Преобразуем в ASCII-код
```

```
AND AX, 111b ; Остаток от деления на 2^3
```

7. Условные операции



7. Условные операции

Команды передачи управления позволяют нарушить естественную последовательность выполнения команд.

Команды безусловной передачи управления:

JMP - Безусловная передача управления;

CALL - Вызов процедуры;

RET - Возврат из процедуры.

Команды условной передачи управления

Команды передачи управления НЕ меняют значения флагов.

7. Условные операции

Безусловный переход

Команда безусловного перехода имеет следующий синтаксис:

JMP <операнд>

Операнд указывает адрес перехода. Существует два способа указания этого адреса, соответственно различают прямой и косвенный переходы.

JMP L

...

...

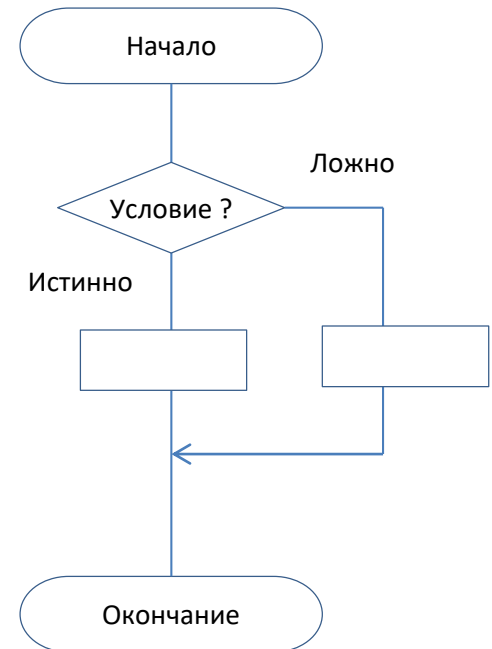
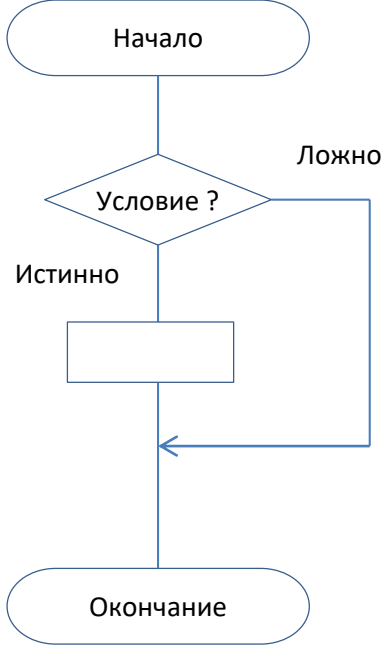
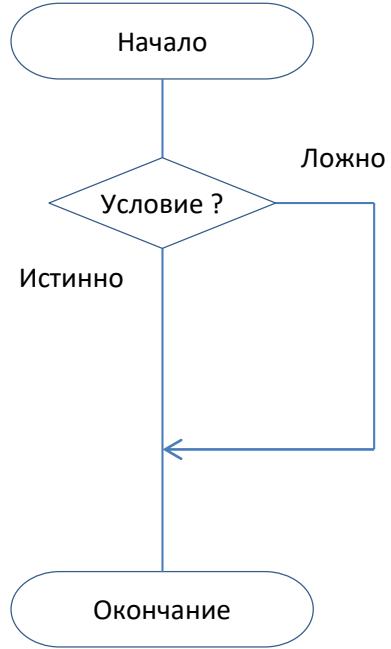
L: MOV eax, [x]

JMP EBX ; регистр

JMP \$

выполняет безусловный переход на саму себя, так что создается вечный цикл из одной команды.

7. Условные операции



if-else, switch, ?: (тернар)

7. Условные операции

В системе команд процессора архитектуры x86 не предусмотрена поддержка условных логических структур, характерных для языков высокого уровня. Однако на языке ассемблера с помощью набора команд сравнения и условного перехода вы можете реализовать логическую структуру любой сложности. В языке высокого уровня любой условный оператор выполняется в два этапа. Сначала вычисляется значение условного выражения, а затем, в зависимости от его результата, выполняются те или иные действия. Проводя аналогию с языком ассемблера, можно сказать, что сначала выполняются такие команды, как **CMP**, **AND** или **SUB**, влияющие на **флаги** состояния процессора. Затем выполняется команда условного перехода, которая анализирует значение нужных флагов, и в случае если они установлены, выполняют переход по указанному адресу.

7. Условные операции

Что касается команд условного перехода, то их достаточно много, но все они записываются единообразно:

Jxx <метка>

Все команды условного перехода можно разделить на три группы

E — Equal (равно).

N — Not (не, отрицание).

G — Greater (больше) — применяется для чисел со **ЗНАКОМ**.

L — Less (меньше) — применяется для чисел со **ЗНАКОМ**.

A — Above (выше, больше) — применяется для чисел **БЕЗ ЗНАКА**.

B — Below (ниже, меньше) — применяется для чисел **БЕЗ ЗНАКА**.

7. Условные операции

Мнемокод	Название	Условие перехода после команды CMP op1, op2	Значения флагов	Примечание
JE	Переход если равно	$op1 = op2$	$ZF = 1$	Для всех чисел
JNE	Переход если не равно	$op1 \neq op2$	$ZF = 0$	
JL/JNGE	Переход если меньше	$op1 < op2$	$SF \neq OF$	Для чисел со знаком
JLE/JNG	Переход если меньше или равно	$op1 \leq op2$	$SF \neq OF$ или $ZF = 1$	
JG/JNLE	Переход если больше	$op1 > op2$	$SF = OF$ и $ZF = 0$	
JGE/JNL	Переход если больше или равно	$op1 \geq op2$	$SF = OF$	
JB/JNAE	Переход если ниже	$op1 < op2$	$CF = 1$	Для чисел без знака
JBE/JNA	Переход если ниже или равно	$op1 \leq op2$	$CF = 1$ или $ZF = 1$	
JA/JNBE	Переход если выше	$op1 > op2$	$CF = 0$ и $ZF = 0$	
JAЕ/JNB	Переход если выше или равно	$op1 \geq op2$	$CF = 0$	

7. Условные операции

Мnemonic	Condition	Мnemonic	Condition
JZ	ZF = 1	JNZ	ZF = 0
JS	SF = 1	JNS	SF = 0
JC	CF = 1	JNC	CF = 0
JO	OF = 1	JNO	OF = 0
JP	PF = 1	JNP	PF = 0

7. Условные операции

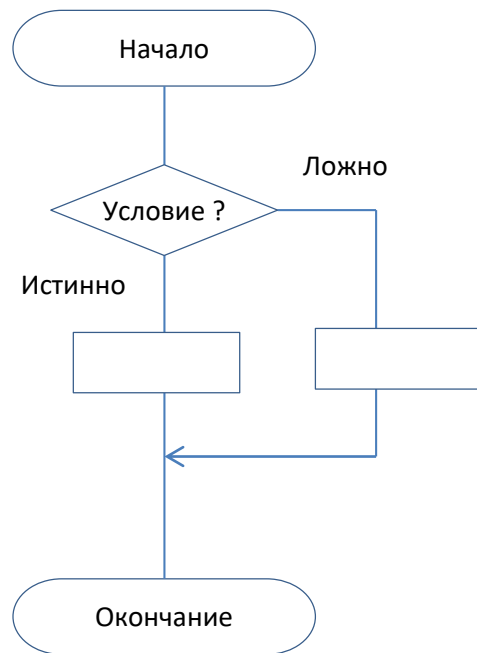
Три команды условного перехода, проверяющие не флаги, а значение регистра RCX, ECX или CX:

JCXZ <метка> ; Переход, если значение регистра CX равно 0

JECXZ <метка> ; Переход, если значение регистра ECX равно 0

JRCXZ <метка> ; Переход, если значение регистра RCX равно 0

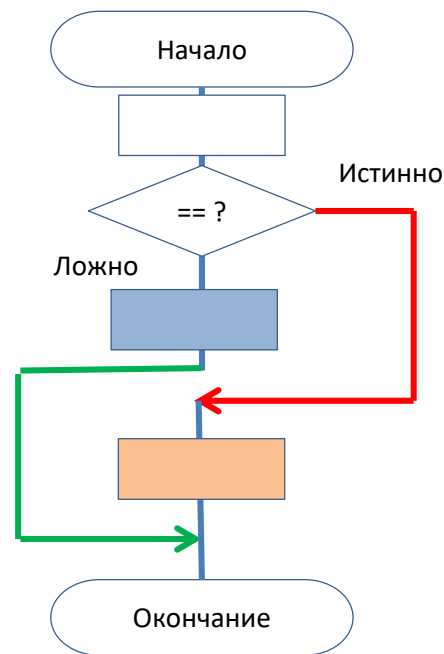
7. Условные операции



```
if( a == 0 )  
    {}  
else  
    {}
```

```
section .data  
a dd 5
```

```
section .text  
mov eax, [a]  
cmp eax, 0           ; a == 0 ?  
je @a_is_0           ; a == 0 ?  
nop                  ; a != 0  
jmp @end_if  
@a_is_0:  
nop                  ; a == 0  
; jmp @end_if  
@end_if:  
ret
```



7. Условные операции

JL/JNGE	Переход если меньше	$op1 < op2$	$SF \neq OF$
---------	---------------------	-------------	--------------

```
a db -1
b db 127
```

```
xor eax, eax
mov al, [a]
mov bl, [b]
cmp al, bl           ; SF = 1, OF = 0
```

```
jl @a_less_b
```

```
  nop ; a >= b
```

```
  jmp @end_if
```

```
@a_less_b:
```

```
  nop ; a < b           ; !!!!!!! OK
```

```
@end_if:
```

```
  ret
```

```
a db 255
b db 127
```

```
xor eax, eax
mov al, [a]
mov bl, [b]
cmp al, bl           ; SF = 1, OF = 0
```

```
jl @a_less_b
```

```
  nop ; a >= b
```

```
  jmp @end_if
```

```
@a_less_b:
```

```
  nop ; a < b           ; !!!!!!! err
```

```
@end_if:
```

```
  ret
```

7. Условные операции

JB/JNAE	Переход если ниже	$op1 < op2$	CF = 1
---------	-------------------	-------------	--------

a db 255

b db 127

xor eax, eax

mov al, [a]

mov bl, [b]

cmp al, bl ; SF = 1, OF = 0, CF = 0

jl @a_less_b

nop ; a >= b

jmp @end_if

@a_less_b:

nop ; a < b !!!!! err

@end_if:

ret

a db 255

b db 127

xor eax, eax

mov al, [a]

mov bl, [b]

cmp al, bl ; SF = 1, OF = 0, CF = 0

jb @a_less_b

nop ; a >= b !!! OK

jmp @end_if

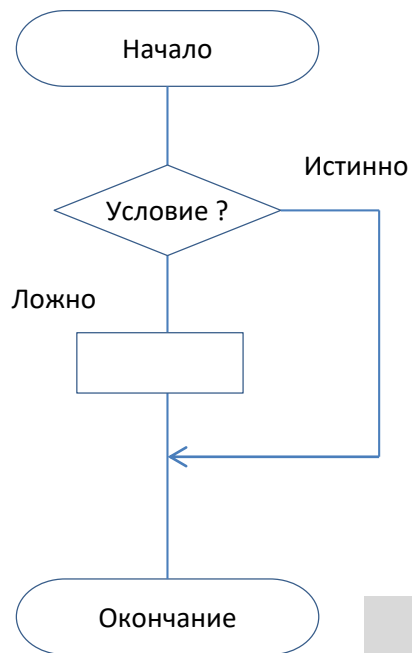
@a_less_b:

nop ; a < b

@end_if:

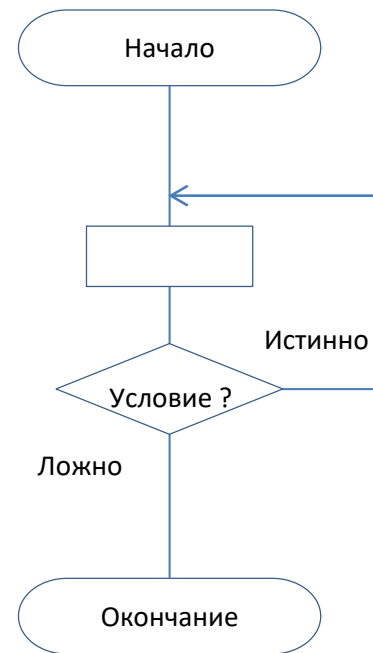
ret

8. Организация цикла



```
if(<условие>)  
    {  
.....  
}
```

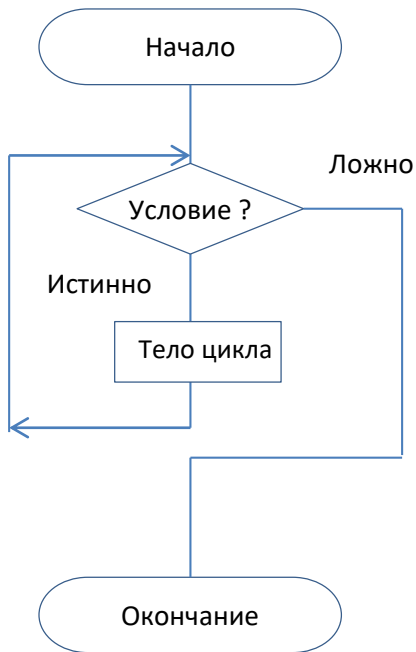
```
mov eax, [a]  
cmp eax, 0;  
je @a_is_0  
; .....  
jmp @end_if  
@a_is_0:  
@end_if:  
ret
```



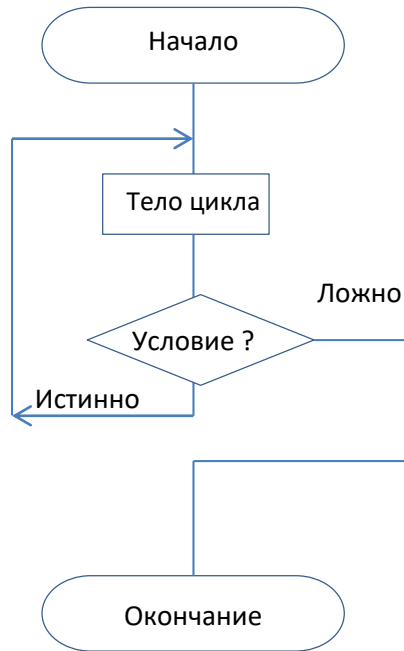
```
do {  
.....  
} while (<условие>)
```

8. Организация цикла

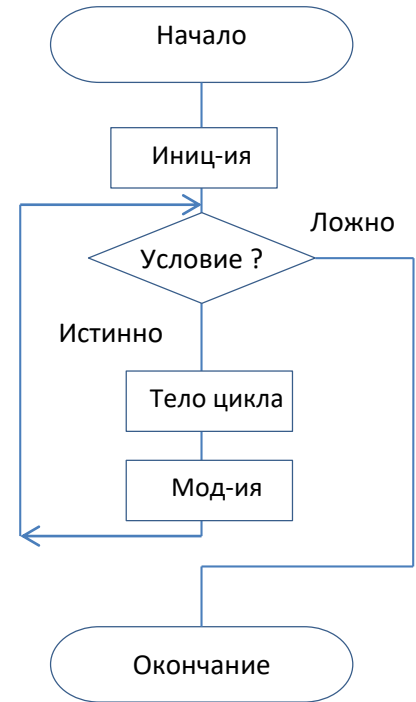
С предусловием



С постусловием



Параметрический



while, do-while, for, (if-else + goto)

8. Организация цикла

```
jmp $
```

```
@label
```

```
....
```

```
jmp @label
```

```
@label: jmp @label
```

8. Организация цикла

Организация цикла с помощью команд условного перехода и регистра ECX/CX .

Если некоторую группу команд необходимо повторить определенное количество раз, цикл можно организовать следующим образом:

- 1) поместить в регистр ecx/cx количество повторений;
- 2) первую команду тела цикла отметить меткой;
- 3) после выполнения тела цикла уменьшить содержимое регистра ecx/cx на 1;
- 4) выполнить сравнение содержимого регистра ecx/cx с нулем;
- 5) в случае, если $ecx/cx \neq 0$, осуществить переход на метку, иначе выполнять следующую за телом цикла команду.

Схема реализации будет выглядеть таким образом:

```
mov cx, [N]
```

```
CYCL:
```

```
<тело цикла>
```

```
dec cx
```

```
cmp cx,0
```

```
jne CYCL
```

```
...
```

8. Организация цикла

Необходимо обращать внимание на выполнение «пустого» цикла. Если вдруг начальное значение в регистре CX равно нулю, то после уменьшения на единицу, содержимым регистра станет число **FFFFh** (-1 в десятичной системе счисления). В результате цикл повторится еще 65535 раз (или 4 294 967 295 раз при использовании регистра ecx). Поэтому следует всегда проверять содержимое регистра ecx/cx для предотвращения выполнения «пустого» цикла.

Такую проверку легко организовать с помощью команды условного перехода **jcxz/jecxz**

```
mov cx, [N]
```

```
jcxz Exit
```

```
CYCL:
```

```
<тело цикла>
```

```
dec cx
```

```
cmp cx,0
```

```
jne CYCL
```

```
Exit: ...
```

8. Организация цикла

Организация цикла с помощью команды безусловного перехода `jmp` и регистра `ecx/cx`.

Цикл можно организовать и следующим образом:

- 1) поместить в регистр `ecx/cx` количество повторений;
- 2) осуществить проверку на «пустой» цикл командой `jcxz/jcxz <метка перехода>`
- 3) эту команду проверки отметить меткой начала цикла;
- 4) после выполнения тела цикла уменьшить содержимое регистра `ecx/cx` на 1;
- 5) выполнить безусловный переход на начало цикла.

Схема реализации будет выглядеть таким образом:

```
mov cx, [N]
```

```
CYCL:
```

```
    jcxz Exit
```

```
    <тело цикла>
```

```
    dec cx
```

```
    jmp CYCL
```

```
Exit: ...
```

Обратим внимание на то, что здесь команда `jcxz` играет двойную роль. Во-первых, предотвращает выполнение «пустого» цикла, и, во-вторых, отслеживает окончание цикла.

8. Организация цикла

Организация цикла с помощью специальных команд.

Заметим, что в описанных первых двух способах организации цикла большинство операций выполняются “вручную” (**декремент регистра *sx*, сравнение *sx* с нулем**, переход на начало цикла).

Учитывая важность циклов (практически ни одна программа не обходится без таких конструкций) разработчики системы команд микропроцессора ввели специальные команды, облегчающие (сокращающие) программирование циклических участков программ.

Это команды

loop <метка перехода>

loope/loopz <метка перехода>

loopne/loopnz <метка перехода>

Обратим внимание, эти команды также используют регистр *esx/sx* как параметр (счетчик) цикла.

8. Организация цикла

Организация цикла с помощью команды loop

В переводе Loop означает «петля», или другими словами «повторить цикл».

Синтаксис команды:

loop <метка перехода>

Работа команды заключается в выполнении следующих действий:

- 1) уменьшение значения регистра **есх/сх** на 1;
- 2) сравнение регистра **есх/сх** с нулем:

если **есх/сх = 0**, то осуществляется выход из цикла,
т.е. управление передается на следующую после loop команду;
иначе — управление передается на метку перехода (цикл повторяется).

8. Организация цикла

Порядок организации цикла с помощью команды loop :

- 1) в регистр cx поместить значение, равное количеству повторений;
- 2) установить метку на первую команду тела цикла;
- 3) выполнить команду loop <метка перехода>.

Схема реализации выглядит следующим образом:

```
mov cx, [N]
jcxz Exit
CYCL :
    <тело цикла>
    loop CYCL
Exit :    ...
```

Обратим внимание также на то, что после выхода из цикла содержимое регистра **cx** всегда равно нулю..

!!! Проверяйте содержимое регистров

8. Организация цикла

Организация цикла с помощью команд `loope` / `loopz`.

Команды `loope` и `loopz` — абсолютные синонимы. Использовать можно любую из них.

Синтаксис команд:

`loope <метка перехода>`

`loopz <метка перехода>`

При использовании этих команд цикл повторяется до тех пор, пока **`cx ≠ 0`** **и** **`zf = 1`**.

Выход из цикла осуществляется при выполнении одного из двух условий: **`cx = 0`** **или** **`zf = 0`**.

Работа команд заключается в выполнении следующих действий:

- 1) уменьшение значения регистра `ecx/cx` на 1;
- 2) сравнение регистра `ecx/cx` с нулем: если $(ecx/cx) = 0$, осуществляется выход из цикла;
- 3) анализ состояния флага нуля `zf`: если `zf = 0`, осуществляется выход из цикла.

Таким образом, цикл повторяется, **если `cx ≠ 0` и `zf = 1`**.

Поэтому, используя флаг `ZF` в качестве индикатора удобно организовать досрочный (до того, как значение `cx` станет 0) выход из цикла.

8. Организация цикла

Порядок организации цикла с помощью команды `loope` :

- 1) в регистр **cx** поместить значение, равное количеству повторений;
- 2) установить метку на первую команду тела цикла;
- 3) использовать команду **cmp** для сравнения операндов (здесь будет установлен **флаг нуля**);
- 4) выполнить команду **loope** <метка перехода>.

Схема реализации выглядит следующим образом:

```
mov cx, [N]
```

```
jcxz Exit
```

```
CYCL :
```

```
<тело цикла>
```

```
cmp <оп1>,<оп2>
```

```
loope CYCL
```

```
Exit: ...
```

Совет. Так как флаг нуля принимает значение 0 в случае несовпадения операндов и, соответственно, выход из цикла осуществляется именно при несовпадении операндов, команды `loope/loopz` удобно использовать для поиска **первого элемента** последовательности, **отличного** от заданной величины.

8. Организация цикла

Организация цикла с помощью команд `loopne/loopnz`.

Команды `loopne/loopnz` являются обратными командам `loope/loopz`.

loopne и **loopnz** также абсолютные синонимы, использовать можно любую из них.

Синтаксис команд:

loopne <метка перехода>

loopnz <метка перехода>

При использовании этих команд цикл повторяется до тех пор, пока **cx** $\neq 0$ **и** **zf** = 0.

Выход из цикла осуществляется при выполнении одного из двух условий: **cx** = 0 **или** **zf** = 1.

Работа команд заключается в выполнении следующих действий:

- 1) уменьшение значения регистра `ecx/cx` на 1;
- 2) сравнение регистра `ecx/cx` с нулем: если (`ecx/cx`) = 0, осуществляется выход из цикла;
- 3) анализ состояния флага нуля `zf` : если `zf` = 1, осуществляется выход из цикла.

Таким образом, цикл повторяется, если `cx` $\neq 0$ и `zf` = 0.

8. Организация цикла

Порядок организации цикла с помощью команды `loopne` :

- 1) в регистр **cx** поместить значение, равное количеству повторений;
- 2) установить метку на первую команду тела цикла;
- 3) использовать команду **cmp** для сравнения операндов (здесь будет установлен **флаг нуля**);
- 4) выполнить команду **loopne** <метка перехода>.

Схема реализации выглядит следующим образом:

```
mov cx , [N]
```

```
jcxz Exit
```

```
CYCL :
```

```
<тело цикла>
```

```
cmp <оп1>,<оп2>
```

```
loopne CYCL
```

```
Exit: ...
```

Совет. Так как флаг нуля принимает значение 1 в случае совпадения операндов и, соответственно, выход из цикла осуществляется именно при совпадении операндов, команды `loopne/loopnz` удобно использовать для **поиска первого элемента последовательности, имеющего заданную величину**.

8. Организация цикла

Форма записи цикла while на языке C/C++:

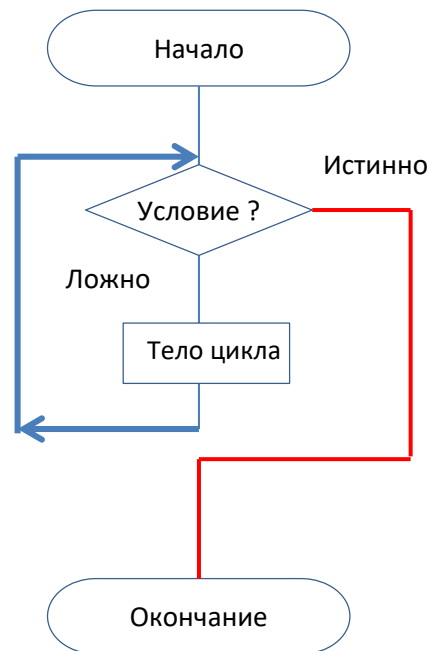
```
while (/*условие продолжения цикла while*/)  
{  
    /*блок операторов*/;  
}
```

Типичная структура такого цикла WHILE на языке ассемблера:

Cont:

```
cmp X, Condition  
jxx Exit    ; условие ВЫХОДА!!  
..... ;тело цикла  
jmp Cont
```

Exit: ...



8. Организация цикла

```
int i = 0;  
while ( i <= 10) {  
    i++;  
}
```

```
i dw 0
```

```
...
```

```
mov ax, [i];
```

Cont:

```
cmp ax, 10
```

; сравнение

```
jg Exit
```

; условие выхода!!

```
inc ax
```

;тело цикла

```
jmp Cont
```

Exit: ...

8. Организация цикла

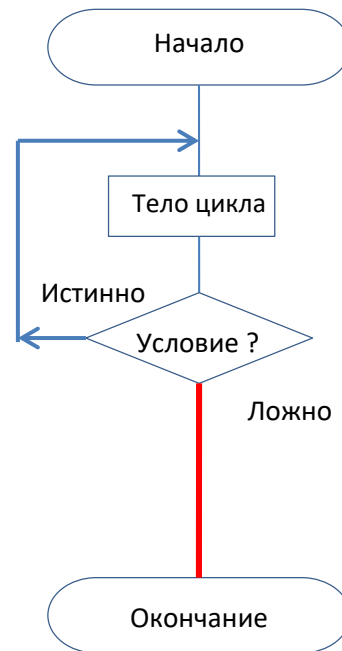
Форма записи цикла while на языке C/C++:

```
do  
{  
    /*блок операторов*/;  
} while (/*условие продолжения цикла*/)
```

Типичная структура такого цикла DO WHILE на языке ассемблера:

Cont:

...	;тело цикла
CMP X, Condition	
Jxx Cont;	условие ПРОДОЛЖЕНИЯ ЦИКЛА



8. Организация цикла

Форма записи на языке C/C++:

```
for (инициализация; условие; модификация) {  
    тело цикла;  
}
```

Типичная структура такого цикла на языке ассемблера:

`mov ax, [N] ; Инициализация`

Cont:

`cmp ax, Condition`

`jxx Exit ; условие ВЫХОДА!!`

`..... ;тело цикла`

`inc ax ; Модификация`

`jmp Cont`

Exit: ...

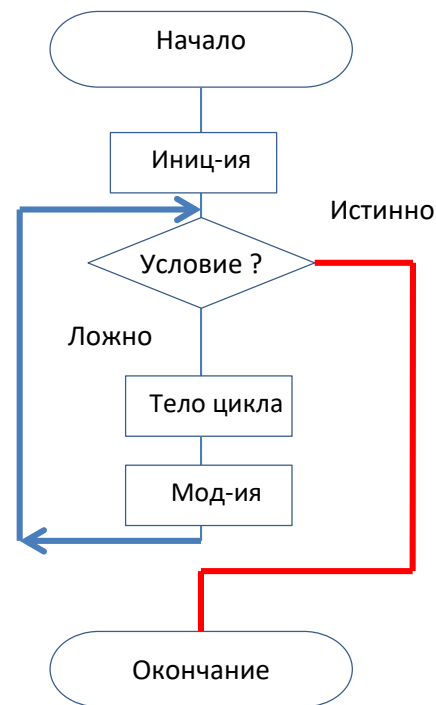
Структура цикла с командой loop (число повторений N):

`mov cx, [N] ; Инициализация`

Cont:

`..... ;тело цикла`

`loop Cont`



8. Организация цикла

Написать программу, которая сумму значений функции $y = x^2 + 4x - 5$ в диапазоне от 1 до 5 с шагом 1.

```
x dw 0
```

```
y dw 0
```

```
mov ecx, 1 ; ecx = X
```

```
cycle:
```

```
mov eax, ecx ; eax = X
```

```
imul ecx ; edx:eax = ecx*ecx = X^2
```

```
mov ebx, ecx ; ebx = X
```

```
sal ebx, 2 ; ebx = X << 2 == 4*X
```

```
add eax, ebx ; eax = X^2 + 4X
```

```
sub eax, 5 ; eax = X^2 + 4X - 5
```

```
mov [y], eax ; y = X^2 + 4X - 5
```

```
inc ecx ; ecx = ecx + 1
```

```
cmp ecx, 5
```

```
jng cycle
```

- Цепочечные примитивы
 - MOVS*, LODS*, SCAS*, CMPS*
 - REP / REPE(Z) / REPNE(Z)

9. Работа с массивами

```
void *memcpy(void *dest, const void *src, size_t n);
```

Функция `memcpy()` копирует `n` байтов из области памяти `src` в область памяти `dest`. Области памяти не могут пересекаться. Используйте `memmove(3)`, если области памяти перекрываются.

```
void *memcpy(void *dest, const void *src, size_t n)
{
    for (size_t i = 0; i < n; ++i )
    {
        *(dest + i) = *(src + i);
        // dest[i] = src[i];
    }
    return dest;
}
```

9. Работа с массивами

```
section .data
    src  dd 0x11223344
    dest dd 0

section .text
    xor eax, eax
    mov ecx, 0
    mov edi, dest
    mov esi, src
next:
    mov al, [esi]      ; al = *(src + i)
    mov [edi], al      ; *(dest+i) = al
    inc ecx            ; i++
    inc esi            ; src + i
    inc edi            ; dest + i
    cmp ecx, 4         ; sizeof(dd) == 4
    jne next
    ret
```

9. Работа с массивами

Синтаксис:	MOVSB
Операнды:	Нет
Назначение:	Копирование строк байтов
Процессор:	8086+
Флаги:	Не изменяются
Комментарий:	Команда MOVSB копирует один байт из ячейки памяти по адресу DS:SI в ячейку памяти по адресу ES:DI. После выполнения команды, регистры SI и DI увеличиваются на 1, если флаг DF = 0, или уменьшаются на 1, если DF = 1. Если команда используется в 32-разрядном режиме адресации, то используются регистры ESI и EDI соответственно.
Ограничения:	Нет
Примеры:	<pre>mov si, str1 mov di, str2 cld movsb</pre>

9. Работа с массивами

```
section .data
    src    dd 0x11223344
    dest   dd 0

section .text

    xor    eax, eax
    mov    ecx, 0
    mov    edi, dest
    mov    esi, src
    cld                                ; направление
next:
    movsb                                ; *(dest+i) = *(src + i)
    inc    ecx                          ; i++
    cmp    ecx, 4                       ; sizeof(dd) == 4
    jne    next                         ; jle next
```

MOVSb, MOVSw, MOVSD, MOVsq

9. Работа с массивами

Синтаксис:	SCASB
Операнды:	Нет
Назначение:	Сканирование строки байт
Процессор:	8086+
Флаги:	Флаги OF, SF, ZF, AF, PF, CF устанавливаются в соответствии с результатом.
Комментарий:	Команда SCASB сравнивает регистр AL с байтом в ячейке памяти по адресу ES:DI и устанавливает флаги аналогично команде CMP. После выполнения команды, регистр DI увеличивается на 1, если флаг DF = 0, или уменьшается на 1, если DF = 1. Если команда используется в 32-разрядном режиме адресации, то используется регистр EDI.
Ограничения:	Нет
Примеры:	<pre>mov di, str1 cld scasb</pre>

9. Работа с массивами

```
section .data
    dest db "hello", 0

section .text

    xor eax, eax
    mov ecx, 0
    mov edi, dest
    dec ecx
    cld
next:
    inc ecx
    mov al, 0
    scasb
    ; *(dest+i) == al ?
    jne next
```

```
size_t strlen(const char *s)
{
    int cnt = -1;

    do
    {
        cnt ++;
    } while( *(s + cnt) != 0 );

    return (size_t)cnt;
}
```

9. Работа с массивами

интаксис:	LODSB
Операнды:	Нет
Назначение:	Чтение байта из строки
Процессор:	8086+
Флаги:	Не изменяются
Комментарий:	Команда LODSB копирует слово из памяти по адресу DS:SI в регистр AX. После выполнения команды, регистр SI увеличивается на 1, если флаг DF = 0, или уменьшается на 1, если DF = 1. Если команда используется в 32-разрядном режиме адресации, то используется регистр ESI.
Ограничения:	Нет
Примеры:	<code>mov si, str1</code> <code>cld</code> <code>Lodsb</code>

9. Работа с массивами

```
:: toUpper ( const char *src, size_t n);
```

```
section .data
    src    db "hello"
    n      db $ - src

section .text

    xor eax, eax
    mov ecx, [n] ; strlen("hello")
    mov edi, dest
    mov esi, dest
    cld
step:
    lodsb                ; al = *(src + i)
    and al, 0xDF         ; to Uppercase (sub al, 0x20)
    mov [edi], al
    inc edi              ;

    dec ecx
    cmp ecx, 0
    jne step
```

9. Работа с массивами

Синтаксис:	STOSB
Операнды:	Нет
Назначение:	Запись байта в строку
Процессор:	8086+
Флаги:	Не изменяются
Комментарий:	Команда STOSB сохраняет регистр AL в ячейке памяти по адресу ES:DI. После выполнения команды, регистр DI увеличивается на 1, если флаг DF = 0, или уменьшается на 1, если DF = 1. Если команда используется в 32-разрядном режиме адресации, то используется регистр EDI.
Ограничения:	Нет
Примеры:	<pre>mov di, offset str1 cld stosb</pre>

9. Работа с массивами

```
void *memset (void *s, int c, size_t n);
```

```
section .data
    dest  db  "hello"
    n      db  $ - src
    c      db  0

section .text

    xor  eax, eax
    mov  ecx, 0
    mov  edi, dest
    cld                                ; направление
next:
    mov  al, [c]
    stosb                               ; *(dest+i) = c
    inc  ecx                            ; i++
    cmp  ecx, [n]                       ; strlen()
    jne  next                           ; jle next
```

```
void *memset(void *s, int c, size_t n);
{
    for (size_t i = 0; i < n; ++i )
        *(s + i) = c;

    return s;
}
```

9. Работа с массивами

Синтаксис:	CMPSB
Операнды:	Нет
Назначение:	Сравнение строк байтов
Процессор:	8086+
Флаги:	Флаги OF, SF, ZF, AF, PF и CF устанавливаются в соответствии с результатом.
Комментарий:	Команда CMPSB сравнивает один байт из памяти по адресу DS:SI с байтом по адресу ES:DI. Аналогична по действию команде CMP. После выполнения команды, регистры SI и DI увеличиваются на 1, если флаг DF = 0, или уменьшаются на 1, если DF = 1. Если команда используется в 32-разрядном режиме адресации, то используются регистры ESI и EDI соответственно.
Ограничения:	Нет
Примеры:	<pre>mov si,offset str1 mov di,offset str2 cld cmpsb</pre>

9. Работа с массивами

```
int memcmp (const void *s1, const void *s2, size_t n);
```

```
    mov     esi, src      ; Указатели
    mov     edi, dest
    xor     ecx, ecx      ; счетчик
    xor     eax, eax      ; результат
    cld                      ; направление
@next:
    inc     ecx
    cmp     ecx, [n]
    je     @end
    cmpsb                   ; *(dest+i) == *(src + i)
    je     @next
    inc     eax;           ; не равны
@end:
```

Префиксы повторений REPx

REP	Повторять команду , пока (E)CX $\neq 0$
REPE / REPZ	Повторять команду, пока равно (флаг ZF = 1) и (E)CX $\neq 0$
REPNE / REPNZ	Повторять команду, пока НЕ равно (флаг ZF = 0) и (E)CX $\neq 0$

9. Работа с массивами

```
section .data
    src  dd 0x11223344
    dest dd 0

section .text

    xor eax, eax
    mov ecx, 0
    mov edi, dest
    mov esi, src
    cld                      ; направление
next:
    movsb                    ; *(dest+i) = *(src + i)
    inc ecx                  ; i++
    cmp ecx, 4               ; sizeof(dd) == 4
    jne next                 ; jle next
```

```
section .data
    src  dd 0x11223344
    dest dd 0

section .text

    xor eax, eax
    mov ecx, 4
    mov edi, dest
    mov esi, src

    cld
    rep movsb

    ; !!! ecx == 0
```

Объявление, инициализация

```
section .data
```

```
    arrByte db 0 ; uint8_t arrByte[4] = {0, 1, 2, 3};
```

```
        db 1
```

```
        db 2
```

```
        db 3
```

```
    arrByte1 db 0, 1, 2, 3 ; uint8_t arrByte1[4] = {0, 1, 2, 3};
```

```
section .bss
```

```
    arrByte2 resb 4 ; uint8_t arrByte2[4];
```

Объявление, инициализация

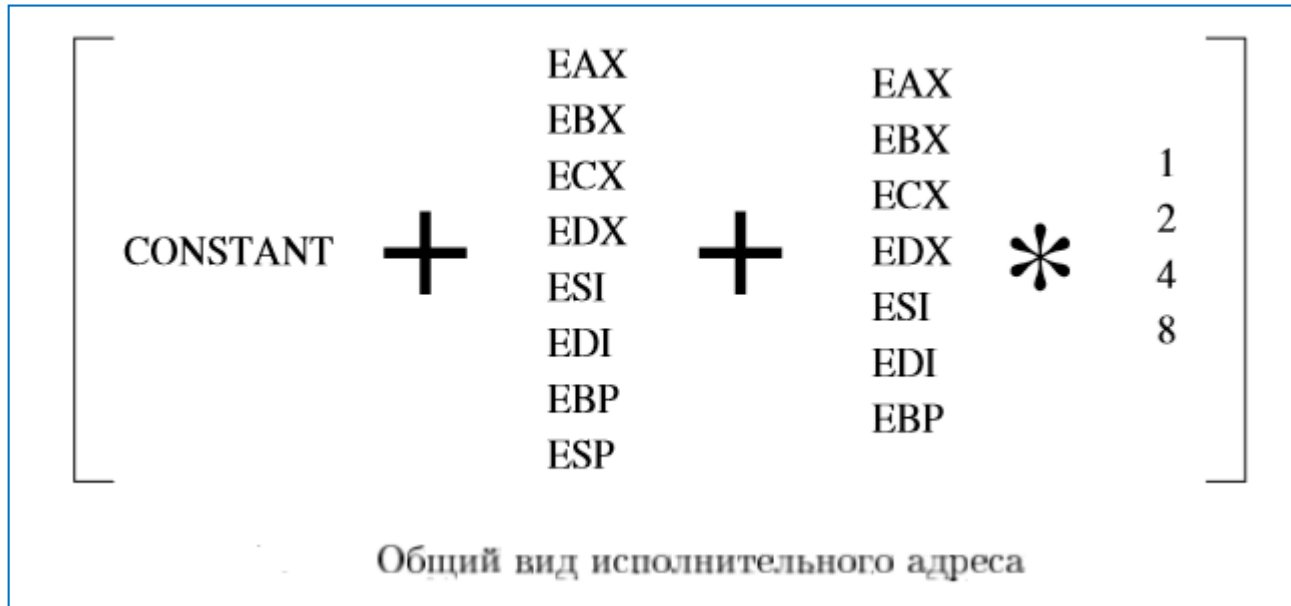
```
section .data
```

```
arrInt times 4 dd 0 ; uint32_t arrInt[4] = {0};
```

```
section .bss
```

```
arrInt2 resd 4 ; uint32_t arrInt2[4];
```

9. Работа с массивами



1. «Имя» переменной (метка) == статический адрес == const
2. `arr[0] == *(arr + 0) => [arr + 0]`
3. `arr[3] == *(arr + 3) => [arr + 3]`
4. `arr[i] == *(arr + i) =>` если `mov ebx, [i]`, то `[arr + ebx]`

Предупреждение: справедливо только для массивов переменных типа byte

9. Работа с массивами

```
section .data
    arrByte times 5 db 0          ; uint8_t arrByte[5] = {0, 0, 0, 0, 0}
    len      db $-arrByte        ; len = 5, так как массив физически занимает 5 байт
section .text
...
    xor eax, eax
    movzx ebx, byte [len]        ; условие выхода из цикла
    mov ecx, 0                   ; счетчик цикла
@lab:
    mov [arrByte + ecx], cl      ; *(arrByte + ecx) = cl – инициализируем значением
                                ; младшего байта счетчика
    inc ecx                     ; увеличиваем счетчик
    cmp ecx, ebx                ; проверяем условие выхода из цикла
    jb @lab
```

Итог: `arrByte[5] = {0, 1, 2, 3, 4}`

9. Работа с массивами

section .data

arrByte times 5 db 0

len db \$-arrByte

section .text

...

xor eax, eax

mov eax, arrByte

movzx ebx, byte [len]

mov ecx, 0

@lab:

mov [eax + ecx], cl

inc ecx

cmp ecx, ebx

jb @lab

; uint8_t arrByte[5] = {0, 0, 0, 0, 0}

; len = 5, так как массив физически занимает 5 байт

; eax= arrByte – заносим адрес массива, &arrByte[0]

; условие выхода из цикла

; счетчик цикла

; *(arrByte + ecx) = cl – инициализируем значением
; младшего байта счетчика

; увеличиваем счетчик

; проверяем условие выхода из цикла

Итог: arrByte[5] = {0, 1, 2, 3, 4}

9. Работа с массивами

```
uint8_t arr[3] = {1, 2, 3}; //sizeof(uint8_t) == 1
```

```
for(int i = 0; i < 3; i++)  
{  
    printf("Адрес 0x%x = %d\n", &arr[i], *(arr + i));  
}
```

Output:

0x0acd0126 = 1;

0x0acd0127 = 2;

0x0acd0128 = 3;

1) Адреса ячеек отличаются на 1

2) i++ – изменяет адрес на 1

9. Работа с массивами

```
uint32_t arr[3] = {1, 2, 3}; //sizeof(uint32_t) == 4
```

```
for(int i = 0; i < 3; i++)  
{  
    printf("Адрес 0x%x = %d\n", &arr[i], *(arr + i));  
}
```

Output:

0x0acd0126 = 1;

0x0acd012a = 2;

0x0acd012e = 3;

1) Адреса ячеек отличаются на 4

2) i++ – изменяет адрес на 4 (!)

В ассемблере (в отличии от Си) адресную арифметику надо делать вручную

9. Работа с массивами

section .data

arrInt times 5 dd 0 ; uint32_t arrInt[5] = {0, 0, 0, 0, 0}

len db \$-arrInt ; len = 5*4 = 20

section .text

...

movzx ebx, byte [len] ; условие выхода из цикла

mov ecx, 0 ; счетчик цикла

@lab:

mov eax, dword [arrInt + ecx] ; eax = *(arrInt+ ecx) – читаем элемент массива

add eax, ecx ; увеличиваем на значение счетчика

; (игнорируем переполнение)

mov [arrInt + ecx], eax ; *(arrInt + ecx) = eax – сохраняем элемент массива

add ecx, 4 ; увеличиваем счетчик на sizeof(uint32_t)

cmp ecx, ebx ; проверяем условие выхода из цикла

jb @lab

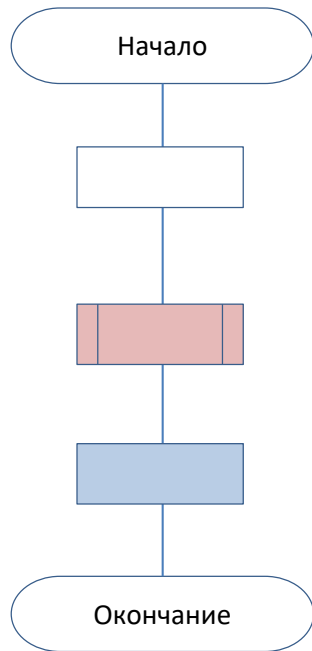
Итог: arrInt[5] = {0, 4, 8, 12, 16}

9. Работа с массивами

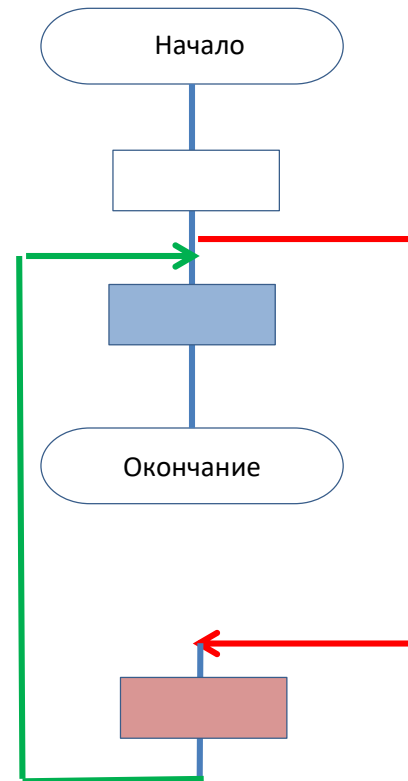
```
section .data
    arrInt times 5 dd 0          ; uint32_t arrInt[5] = {0, 0, 0, 0, 0}
    len      db $-arrInt        ; len = 5*4 = 20,
section .text
...
    movzx ebx, byte [len]        ; условие выхода из цикла
    shr ebx, 2                   ; делим на sizeof(uint32_t) => 20/4 = 5
    mov ecx, 0                   ; счетчик цикла
@lab:
    mov eax, dword [arrInt + ecx * 4] ; eax = *(arrInt + ecx) – читаем элемент массива
                                    ; (адресная арифметика)
    add eax, ecx                 ; увеличиваем на значение счетчика (игнорируем переполнение)
    mov [arrInt + ecx * 4], eax   ; *(arrInt + ecx) = eax – сохраняем элемент массива
    inc ecx                      ; увеличиваем счетчик (адресная арифметика)
    cmp ecx, ebx                 ; проверяем условие выхода из цикла
    jb @lab
```

Итог: arrInt[5] = {0, 1, 2, 3, 4}

10. Функции



Вызов некоторой подпрограммы выглядит как структура «следование». Но учитывая, что инструкции располагаются в памяти последовательно, вызов подпрограммы – это **переход** по какому-то адресу вне основной функции и **возврат** назад, после окончания ее работы.



10. Функции

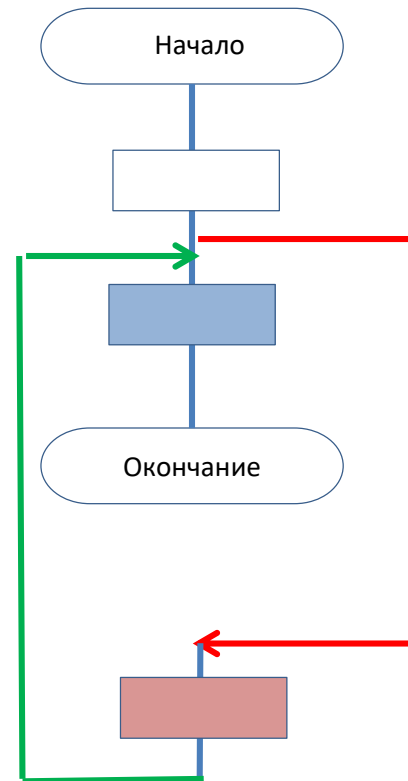
Для того, чтобы иметь возможность вернуться в точку вызова подпрограммы, следует при вызове подпрограммы запоминать этот адрес возврата.

Вызов функции: CALL <метка>

По команде CALL **в стек сохраняется адрес** инструкции, куда следует вернуться после завершения работы подпрограммы.

Возврат: RET

Команда RET завершает работу подпрограммы, вычитывает из стека адрес возврата и осуществляет переход на него.



10. Функции

CMAIN:

l1:

```
xor rax, rax
xor rbx, rbx
```

l2:

```
call myfunc
```

l3:

```
inc rbx
ret
```

myfunc:

```
inc rax
ret
```

> x /x l1

0x4014e3 <l1>: 0x48e58948

> x /x l2

0x4014ec <l2>: 0x000004e8

> x /x l3

0x4014f1 <l3>: 0xc3c3ff48

> x /x myfunc

0x4014f5 <myfunc>: 0xc3c0ff48

До CALL

> x /2x \$rsp

0x23fe68: 0x004013a5 0x00000000

После CALL

> x /4x \$rsp

0x23fe60: 0x004014f1 0x00000000

0x004013a5 0x00000000

После RET

> x /2x \$rsp

0x23fe68: 0x004013a5 0x00000000

```
section .text
    xor eax, eax
    call myfunc          ; в стек помещается адрес
                        ; следующей инструкции
    xor ebx, ebx

myfunc:                  ; Начало подпрограммы
    ...
    ret                  ; возврат, очистка стека
```

Основное требование к подпрограмме: не портить регистры.
Это означает, что если в подпрограмме изменяются значения каких-то регистров, то их надо сохранить на входе в подпрограмму и восстановить на выходе

10. Функции

Основное требование к подпрограмме: не портить регистры.

Это означает, что если в подпрограмме изменяются значения каких-то регистров, то их надо сохранить на входе в подпрограмму и восстановить на выходе, используя push/pop команды

```
section .text
    xor eax, eax
    mov ecx, 6
    call myfunc          ; в стек помещается адрес
                        ; следующей инструкции
    xor ebx, ebx
    add ebx, ecx

myfunc:                 ; Начало подпрограммы
    push ecx
    mov ecx, 3          ; Изменяем регистр
    pop ecx
```

2. Передача параметров в функцию

1. Глобальные переменные
2. Регистры
3. Стек

2. Передача параметров в функцию

1. Глобальные переменные

Используем

- Переменные в секции .DATA и BSS
- Extern переменные при взаимодействии с другими модулями

```
section .data
```

```
extern var  
extern var1  
extern result
```

```
section .text
```

```
global asmfunc
```

```
asmfunc:
```

```
    mov rax, 0  
    mov ax, 32  
    add ax, [var]  
    mov [result], ax
```

```
    mov [var1], dword 45
```

```
    ret
```

```
#include <stdio.h>  
#include <stdint.h>  
#include <stdlib.h>
```

```
int32_t var = 3;  
int16_t var1 = 1;  
int32_t result = 34;
```

```
extern int asmfunc(void);
```

```
int main (void)  
{  
    asmfunc();  
    printf("hello: res = %d, var1 = %d\n", result, var1);  
    return 0;  
}
```

2. Передача параметров в функцию

2. Регистры

Самый быстрый способ обмена данными между функциями

```
int i = 10;  
register int* a = &i;
```

```
section .text  
    xor eax, eax  
    mov eax, 5  
    mov ebx, 3  
    call myfunc  
    ; используем eax как результат  
  
myADD:  
    add eax, ebx  
    ret      ;
```

2. Fastcall

Для передачи параметров используем регистры

Пример: `int 0x80` И `syscall`

Это самый быстрый вызов подпрограммы

Arch/ABI	Instruction					System call #	Ret val	Ret val2	Error	Notes
i386	int \$0x80					eax	eax	edx	-	
x86-64	syscall					rax	rax	rdx	-	

Arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7	Notes
i386	ebx	ecx	edx	esi	edi	ebp	-	
x86-64	rdi	rsi	rdx	r10	r8	r9	-	

2. Передача параметров в функцию

```
int c = 0;
```

```
int func(int _a, int _b)
{
    return (_a + _b);
}
```

```
int main (void)
{
    c = func(1, 2);
    return 0;
}
```

```
int func(int _a, int _b)
{
401102:  55                push rbp
401103:  48 89 e5          mov  rbp,rsp
401106:  89 7d fc          mov  DWORD PTR [rbp-0x4], edi
401109:  89 75 f8          mov  DWORD PTR [rbp-0x8], esi
    return (_a + _b);
40110c:  8b 55 fc          mov  edx,DWORD PTR [rbp-0x4]
40110f:  8b 45 f8          mov  eax,DWORD PTR [rbp-0x8]
401112:  01 d0            add  eax,edx
}
401114:  5d                pop  rbp
401115:  c3                ret
0000000000401116 <main>:
int main (void)
{
401116:  55                push rbp
401117:  48 89 e5          mov  rbp,rsp
    c = func(1, 2);
40111a:  be 02 00 00 00    mov  esi,0x2
40111f:  bf 01 00 00 00    mov  edi,0x1
401124:  e8 d9 ff ff ff    call 401102 <func>
401129:  89 05 fd 2e 00 00 mov  DWORD PTR [rip+0x2efd], eax    # 40402c <c>
```

2. Передача параметров в функцию

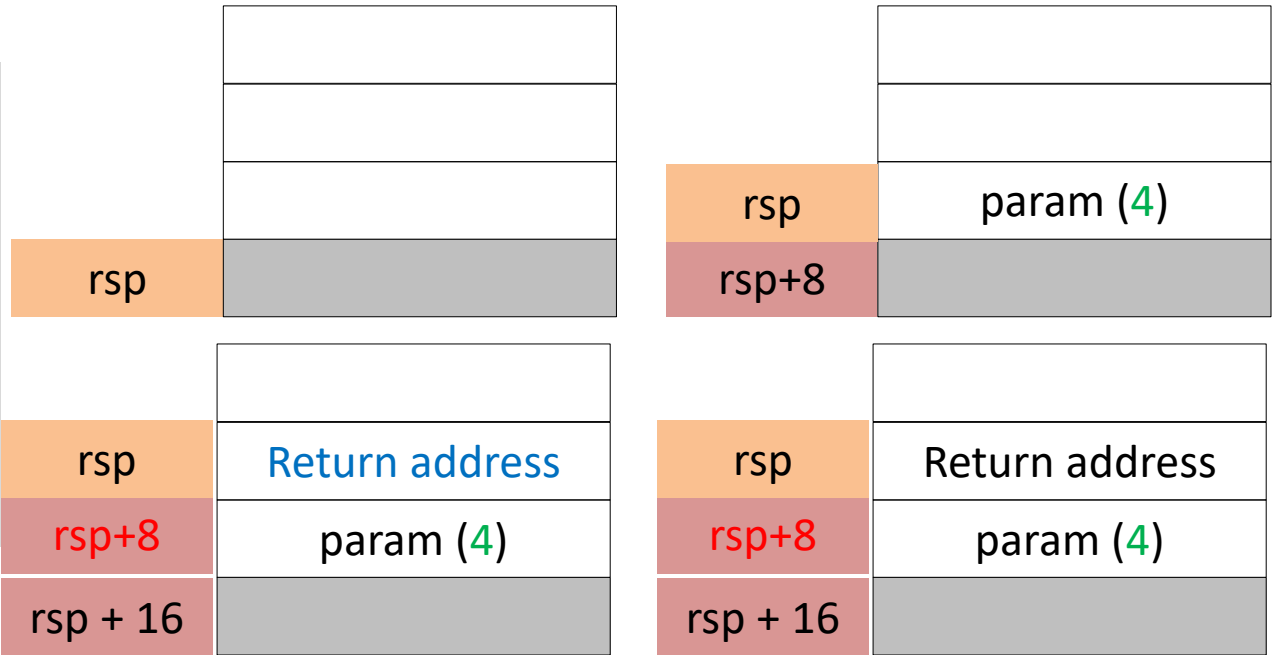
3. Стек

До вызова функции все параметры, используя `push`, загружаются в стек.
В функции из стека эти параметры выгружаются в регистр по мере надобности.

Основная проблема #1 – нельзя использовать `pop`

```
section .text
    mov rax, 4
    push rax
    call myfunc

myfunc:
    ; pop ecx - error
    mov ecx, [rsp+8]
    ret
```



2. Передача параметров в функцию

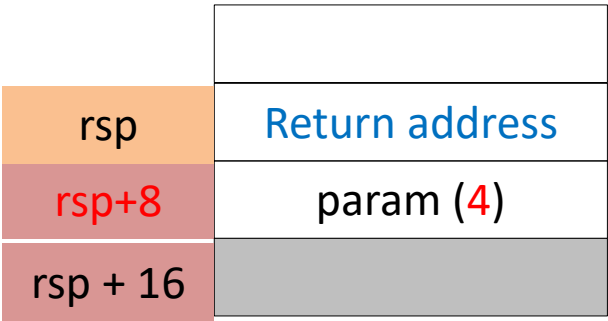
3. Стек

До вызова функции все параметры, используя push, загружаются в стек.
В функции из стека эти параметры выгружаются в регистр по мере надобности.

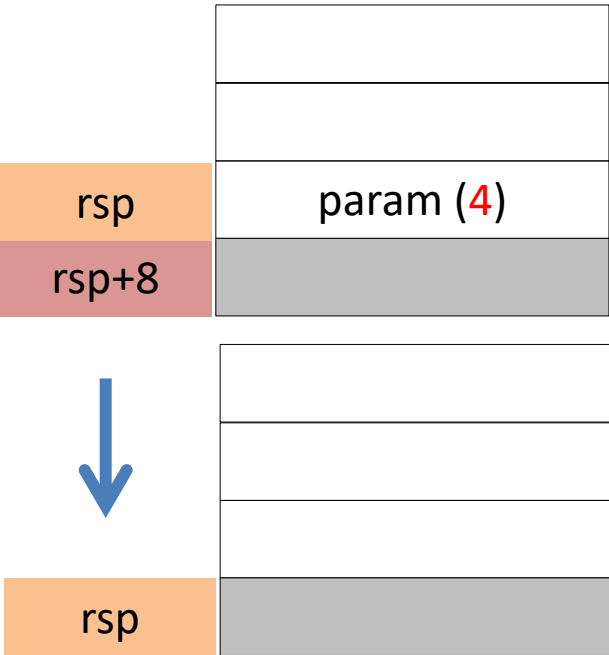
Основная проблема №2 – как извлечь аргумент после окончания вызова функции.

```
section .text
    mov rax, 4
    push rax
    call myfunc
    ?????

myfunc:
    mov ecx, [rsp+8]
    ret
```



- 1) pop eax
- 2) add rsp, 8



2. Передача параметров в функцию

```
section .text
    mov rax, 4
    push rax
    call myfunc
    add rsp, 8

myfunc:
    push rbp
    ...
    pop rbp
    ret
```

```
section .text
    mov rax, 4
    push rax
    call myfunc
    ;

myfunc:
    push rbp
    ...
    pop rbp
    ret 8
```

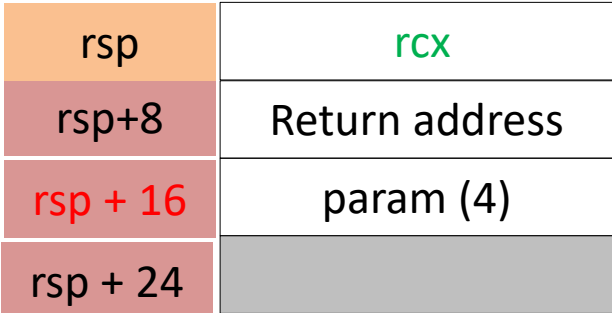
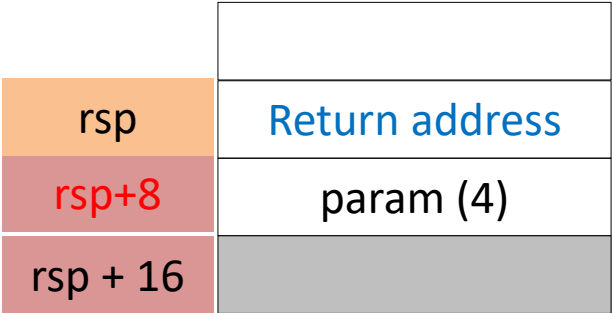
Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C3	RET	ZO	Valid	Valid	Near return to calling procedure.
CB	RET	ZO	Valid	Valid	Far return to calling procedure.
C2 iw	RET imm16	I	Valid	Valid	Near return to calling procedure and pop imm16 bytes from stack.
CA iw	RET imm16	I	Valid	Valid	Far return to calling procedure and pop imm16 bytes from stack.

2. Передача параметров в функцию

3. Стек

Основная проблема № 3 – как сопрячь требования передачи через стек и требование сохранения регистров на входе в функцию? Ведь смещается указатель на вершину стека (rsp).

```
section .text
    mov rax, 4
    push rax
    call myfunc
    ...
myfunc:
    push rcx
    mov ecx, [rsp+16]
    ...
    pop rcx
    ret
```



2. Передача параметров в функцию

3. Стек

Использование вспомогательного регистра для сохранения указателя на вершину стека

myfunc:

mov rax, 4

push rax

call myfunc

push rbp

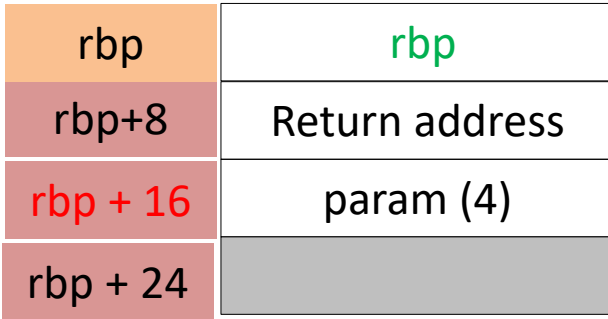
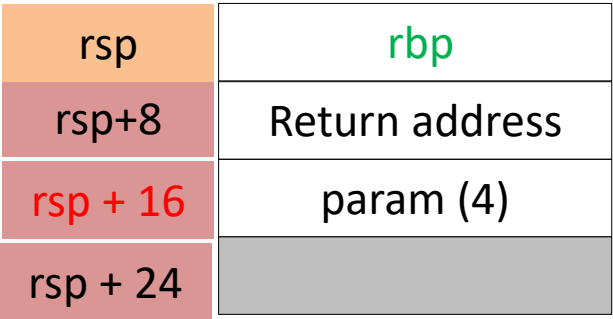
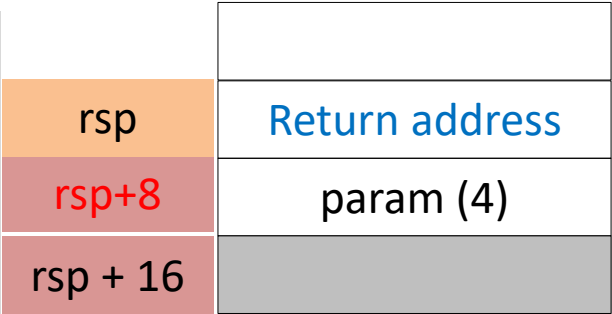
mov rbp, rsp

mov ecx, [rbp+16]

mov rsp, rbp ;!!!

pop rbp

ret

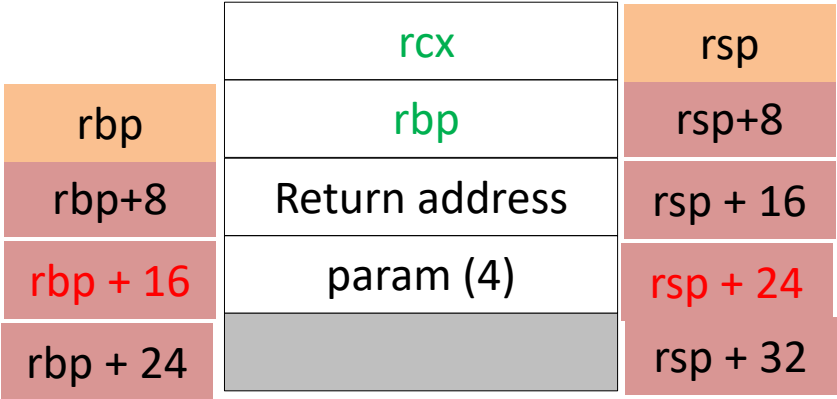
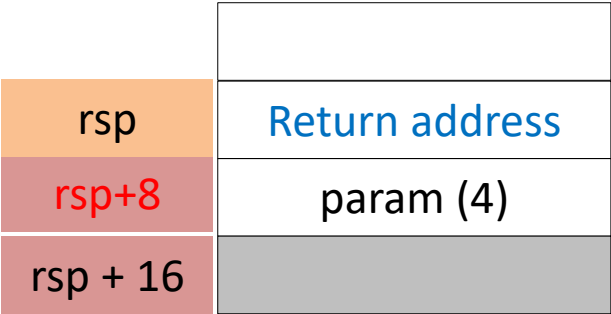


2. Передача параметров в функцию

3. Стек

Использование вспомогательного регистра для сохранения указателя на вершину стека

```
mov rax, 4
push rax
call myfunc
myfunc:
push rbp
mov rbp, rsp
push rcx
mov ecx, [rbp+16]
pop rcx
mov rsp, rbp ;!!!
pop rbp
ret
```

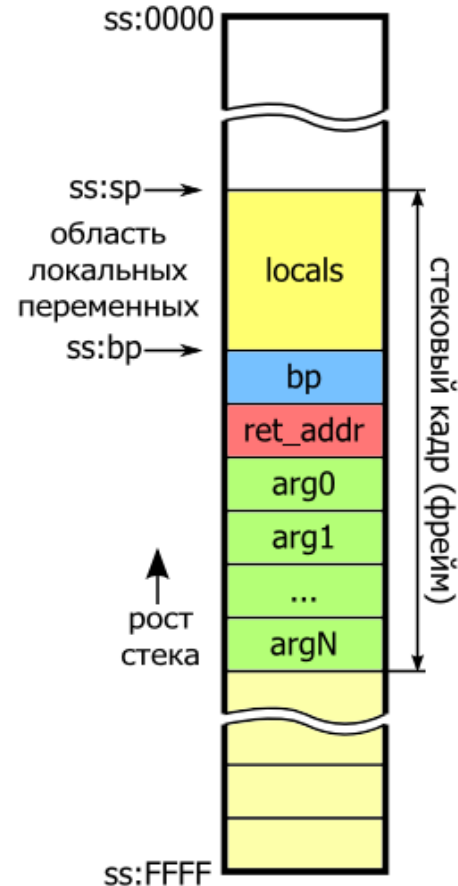


3. Локальные переменные

В отличие от глобальных переменных, размещаемых в секциях DATA и BSS, локальные переменные располагаются в стеке.

Выделение памяти под переменную производится смещением значения регистра ESP в меньшую сторону (стек растет в сторону уменьшения адресов) на размер локальных переменных.

Так как регистр ESP «занят» при использовании команд push/pop, то для навигации внутри выделенного пространства на стеке используется вспомогательный регистр EBP.



3. Локальные переменные

```
int func(int _a, int _b)
{
```

```
401102: 55
401103: 48 89 e5
401106: 89 7d fc
401109: 89 75 f8
    return (_a + _b);
40110c: 8b 55 fc
40110f: 8b 45 f8
401112: 01 d0
```

```
}
```

```
401114: 5d
401115: c3
```

```
push rbp
```

```
mov rbp, rsp
```

```
mov DWORD PTR [rbp-0x4], edi
```

```
mov DWORD PTR [rbp-0x8], esi
```

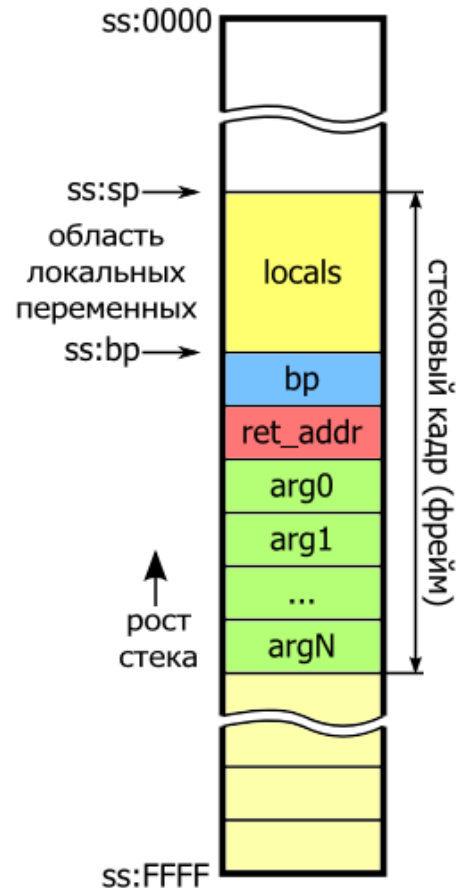
```
mov edx, DWORD PTR [rbp-0x4]
```

```
mov eax, DWORD PTR [rbp-0x8]
```

```
add eax, edx
```

```
pop rbp
```

```
ret
```



3. Локальные переменные

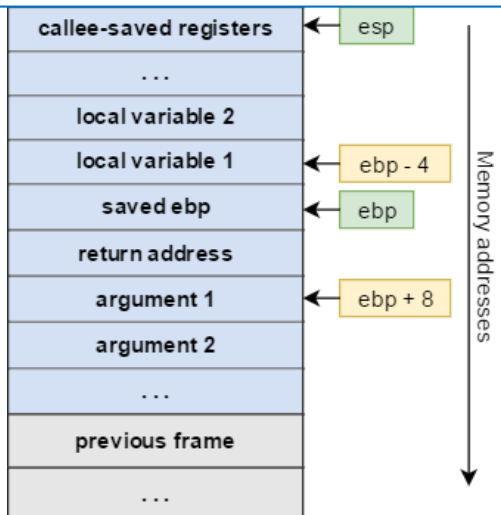
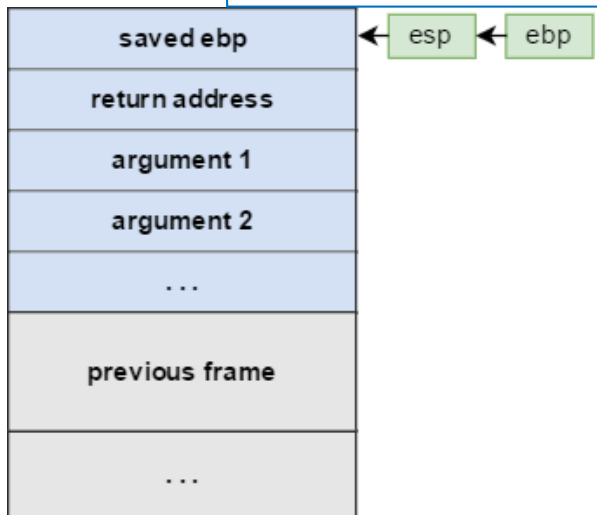
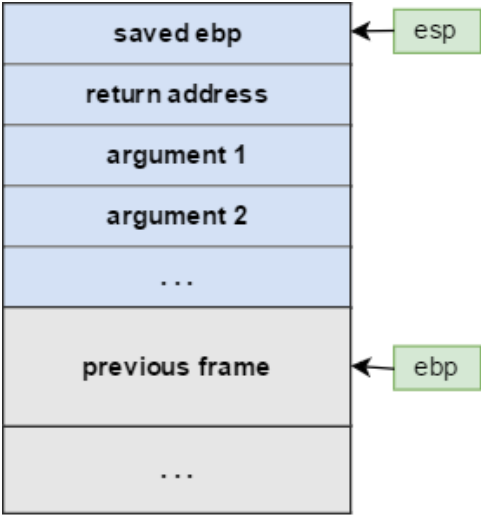
! Пример для 32 бит (4 байта) !

```
push 2 ; пушим аргументы в обратном порядке
push 10

call _subtract ;

add esp, 8 ; удаляем 8 байт со стека (два аргумента по 4 байта)
```

```
_subtract:
push ebp ; сохранение указателя базы предыдущего фрейма
mov ebp, esp ; настройка ebp
sub esp, 8 ; В стеке выделяем место под две переменные
mov eax, [ebp+8] ; копирование первого аргумента функции
mov [ebp-4], eax; копирование в локальную переменную 1
mov ebx [ebp+12] ; копирование второго аргумента
mov [ebp-8], eax; копирование в локальную переменную 2
..... работа с регистрами
add esp, 8 ; или mov esp, ebp - «удаление» локальных переменных
pop ebp ; восстановление указателя базы предыдущего фрейма
ret
```



10. Функции

section .text

```
mov ecx, 6  
push ecx  
call myfunc  
pop ecx
```

myfunc:

```
push ebp                ; сохраняем изменяемое  
                        ;  
mov ebp, esp            ; запоминаем границу  
                        ; стека  
sub esp, 4              ; выделили 4 байта  
                        ; под локальную пер-ю  
push eax                ; сохраняем изменяемое  
  
mov [ebp-4], 10          ; записали в локальную  
                        ; переменную  
mov eax, [ebp + 8]       ; прочитали значение  
                        ; переданного параметра  
  
pop eax                 ; «чистим» от локальной  
mov esp, ebp             ; переменной, аналог add esp, 4  
pop ebp                 ; восстанавливаем измененное  
ret
```

Состояние стека

esp ->

eax

Local var (10)

ebp ->

saved ebp

esp ->

Return address

param (6)

Увеличение
адреса памяти

```
user@host: ~$ man syscall
```

NAME

syscall - indirect system call

SYNOPSIS

```
#include <sys/syscall.h>    /* Definition of SYS_* constants */
```

```
#include <unistd.h>
```

```
long syscall(long number, ...);
```

11. Ввод/вывод

Arch/ABI	Instruction	System call #	Ret val	Ret val2	Error	Notes
i386	int \$0x80	eax	eax	edx	-	
x86-64	syscall	rax	rax	rdx	-	

Arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7	Notes
i386	ebx	ecx	edx	esi	edi	ebp	-	
x86-64	rdi	rsi	rdx	r10	r8	r9	-	

11. Ввод/вывод

%eax	Имя системного вызова	%ebx	%ecx	%edx
2	sys_fork	struct pt_regs	—	—
3	sys_read	unsigned int	char *	size_t
4	sys_write	unsigned int	const char *	size_t
5	sys_open	const char *	int	int
6	sys_close	unsigned int	—	—
8	sys_creat	const char *	int	—
19	sys_lseek	unsigned int	off_t	unsigned int

<https://filippo.io/linux-syscall-table/>

Файловый дескриптор

Файловый дескриптор (или «дескриптор файла») — это 16-битное целое число, присваиваемое файлу в качестве идентификатора. Когда создается новый файл или открывается существующий, дескриптор файла используется для доступа к нему.

Файловый дескриптор стандартных файловых потоков:

`stdin` — 0;

`stdout` — 1;

`stderr` — 2.

Открытие существующего файла

Для открытия существующего файла выполняются следующие шаги:

- поместите системный вызов `sys_open()` — номер 5 в регистр EAX;
- поместите имя файла в регистр EBX;
- поместите режим доступа к файлу в регистр ECX;

Системный вызов возвращает файловый дескриптор открытого файла в регистр EAX. В случае ошибки, код ошибки также будет находиться в регистре EAX.

Среди **режимов доступа к файлам** чаще всего используются:

- **O_RDONLY** только чтение (0);
- **O_WRONLY** только запись (1);
- **O_RDWR** чтение-запись (2).

```
int open(const char *pathname, int flags);
```

Чтение файла

Для чтения данных из файла выполняются следующие шаги:

- поместите системный вызов `sys_read()` — номер 3 в регистр EAX;
- поместите файловый дескриптор в регистр EBX;
- поместите указатель на входной буфер в регистр ECX;
- поместите размер буфера, т.е. количество байтов для чтения, в регистр EDX.

Системный вызов возвращает количество прочитанных байтов в регистр EAX.

В случае ошибки, код ошибки также будет находиться в регистре EAX.

```
ssize_t read(int fd, void *buf, size_t count);
```

Запись в файл

Для записи в файл выполняются следующие шаги:

- поместите системный вызов `sys_write()` — номер 4 в регистр EAX;
- поместите файловый дескриптор в регистр EBX;
- поместите указатель на выходной буфер в регистр ECX;
- поместите размер буфера, т.е. количество байтов для записи, в регистр EDX.

Системный вызов возвращает фактическое количество записанных байтов в регистр EAX. В случае ошибки, код ошибки также будет находиться в регистре EAX.

```
ssize_t write(int fd, const void *buf, size_t count);
```

11. Ввод/вывод

```
section .text
global _start
```

```
_start:
```

```
    mov     edx, len
    mov     ecx, msg
    mov     ebx, 1      ; file descriptor (stdout)
    mov     eax, 4      ; system call number (sys_write)
    int     0x80        ; call kernel
```

```
    mov     eax, 1      ; system call number (sys_exit)
    int     0x80        ; call kernel
```

```
section .data
```

```
msg     db 'Hello, world!',0xa
len     equ $ - msg
```

11. Ввод/вывод

```
section .text
global _start
```

```
_start:
```

```
    mov     rdx, len           ;message length
    mov     rsi, msg           ;message to write
    mov     rdi, 1             ;file descriptor (stdout)
    mov     rax, 1             ;system call number (sys_write)
    syscall
```

```
    mov     rax, 60            ;system call number (sys_exit)
    syscall
```

```
section .data
```

```
msg     db 'Hello, world!',0xa
len     equ $ - msg
```

Заккрытие файла

Для закрытия файла выполняются следующие шаги:

- поместите системный вызов `sys_close()` — номер 6 в регистр EAX;
- поместите файловый дескриптор в регистр EBX.

В случае ошибки, системный вызов возвращает код ошибки в регистр EAX.

```
int close(int fd) ;
```

12. Сопроцессор

float	4	dd
double	8	dq
long double	10	dt

```
section .data
    mvar4    dd 12.14
    mvar8    dq 12.14
    mvar10   dt 12.14
section .text
global CMAIN
CMAIN:
    mov ebp, esp; for correct debugging
    ;write your code here
    finit
    fld dword [mvar4]
    fld qword [mvar8]
    fld tword [mvar10]
```

```
mvar4    0x41423d71
mvar8    0x402847ae147ae148
mvar10   0x4002c23d70a3d70a3d71
```

st0	12.1400000000000001	(raw 0x4002c23d70a3d70a3d71)
st1	12.1400000000000001	(raw 0x4002c23d70a3d70a4000)
st2	12.140000343322754	(raw 0x4002c23d710000000000)

12. Сопроцессор

123

= 1*10^2 + 2*10^1 + 3*10^0

123.456

= 1*10^2 + 2*10^1 + 3*10^0 + 4*10^-1 + 5*10^-2 + 6*10^-3

	Делитель	Час.	Ост.
123	10	12	3
12	10	1	2
1	10	0	1

123

	Делитель	Час.	Ост.
123	2	61	1
61	2	30	1
30	2	15	0
15	2	7	1
7	2	3	1
3	2	1	1
1	2	0	1
			0

01111011 = 0x7b

12. Сопроцессор

$$123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

$$123.456 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2} + 6 \cdot 10^{-3}$$

	Множ	Произ	Целое
0,456	2	0,912	0
0,912	2	1,824	1
0,824	2	1,648	1
0,648	2	1,296	1
0,296	2	0,592	0
0,592	2	1,184	1
0,184	2	0,368	0
0,368	2	0,736	0
0,736	2	1,472	1
0,472	2	0,944	0
0,944	2	1,888	1
0,888	2	1,776	1
0,776	2	1,552	1
...	...	...	...



$$123.456 = 0111011.0111010010111..... = 1.1110110111010010111..... \cdot 2^6$$

1. Плавающая точка

32 бита (float)

S (1 бит) для хранения знака

E (8 бит) для экспоненты

M (23 бита) для мантиссы



Внутренности числа с плавающей запятой.



Три части числа с плавающей запятой.

$$(-1)^S * 1, M * 2^{(E-127)}$$

1. Плавающая точка

Тип	Размер	Е	М	Смещение
db	8	4	3	7
dw	16	5	10	15
dd	32	8	23	127
dq	64	11	53	1023
dt	80	15	63	16383

12. Сопроцессор

$$(-1)^S * 1, M * 2^{(E-127)}$$

Экспонента как окно (Window) или интервал между двумя соседними целыми степенями двойки.

Мантисса - смещение (Offset) в этом окне.



Три части числа с плавающей запятой.

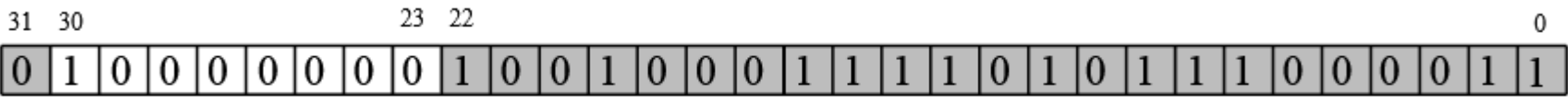
3,14.

- Число 3,14 положительно $\rightarrow S = 0$.
- Число 3,14 находится между степенями двойки 2 и 4, то есть окно числа с плавающей запятой должно начинаться с $2^1 \rightarrow E = 128$ (см. формулу, где окно — это $2^{(E-127)}$).
- Наконец, есть 2^{23} смещений, которыми можно выразить расположение 3,14 внутри интервала [2-4]. Оно находится в $\frac{3,14 - 2}{4 - 2} = 0,57$ внутри интервала, что даёт нам смещение $M = 2^{23} * 0,57 = 4781507$

$S = 0 = 0b$

$E = 128 = 10000000b$

$M = 4781507 = 10010001111010111000011b$



3,14 аппроксимируется как 3,1400001049041748046875

12. Сопроцессор



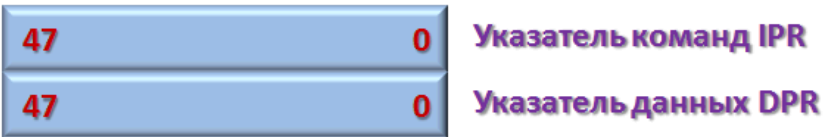
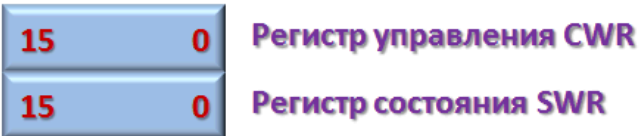
В программной модели сопроцессора можно выделить три группы регистров:

Восемь регистров **r0...r7**, составляющих основу программной модели сопроцессора — **стек сопроцессора**. Размерность каждого регистра 80 битов. Такая организация характерна для устройств, специализирующихся на обработке вычислительных алгоритмов.

Три служебных регистра:

- регистр состояния сопроцессора **swr** (Status Word Register — регистр слова состояния) — отражает информацию о текущем состоянии сопроцессора;
- управляющий регистр сопроцессора **cwr** (Control Word Register — регистр слова управления) — управляет режимами работы сопроцессора;
- регистр тегов **twr** (Tags Word Register — слово тегов) — используется для контроля за состоянием каждого из регистров стека.

Два регистра указателей — данных **dpr** (Data Point Register) и команд **ipr** (Instruction Point Register). Они предназначены для запоминания информации об адресе команды, вызвавшей исключительную ситуацию и адресе ее операнда. Эти указатели используются при обработке исключительных ситуаций (но не для всех команд).



12. Сопроцессор



Регистр состояния swr (FSTAT) — отражает текущее состояние сопроцессора после выполнения последней команды. В регистре swr содержатся поля, позволяющие определить: какой регистр является текущей вершиной стека сопроцессора, какие исключения возникли после выполнения последней команды, каковы особенности выполнения последней команды (некий аналог регистра флагов основного процессора).

Структурно регистр swr состоит из:

- 6 флагов исключительных ситуаций **PE, OE, UE, ZE, DE, IE**.

Исключения — это разновидность прерываний, с помощью которых процессор информирует программу о некоторых особенностях ее реального исполнения. Сопроцессор также обладает способностью возбуждения подобных прерываний при возникновении определенных ситуаций (не обязательно ошибочных). Все возможные исключения сведены к би типам, каждому из которых соответствует 1 бит в регистре swr. Программисту не обязательно писать обработчик для реакции на ситуацию, приведшую к некоторому исключению.

Сопроцессор умеет самостоятельно реагировать на многие из них. Это так называемая обработка исключений по умолчанию. Для того чтобы вызвать обработку определенного типа исключения по умолчанию, необходимо это исключение оставить не маскированным. Такое действие выполняется с помощью установки в 1 соответствующего бита в управляющем регистре сопроцессора swr.

12. Сопроцессор



- бита ошибки работы стека сопроцессора SF (Stack Fault). Бит устанавливается в 1, если возникает одна из трех исключительных ситуаций — PE, UE или IE. В частности, его установка информирует о попытке записи в заполненный стек, или, напротив, попытке чтения из пустого стека. После того как значение этого бита проанализировано, его нужно снова сбросить в 0, вместе с битами PE, UE и IE (если они были установлены);
- бита суммарной ошибки работы сопроцессора ES (Error Summary). Бит устанавливается в 1, если возникает любая из шести перечисленных выше исключительных ситуаций;
- четырех битов c0...c3 (Condition Code) — кода условия. Назначение этих битов аналогично флагам в регистре EFLAGS основного процессора — отразить результат выполнения последней команды сопроцессора.
- трехбитного поля TOP. Поле содержит указатель регистра текущей вершины стека.
- бита В занятости сопроцессора.

12. Сопроцессор



Регистр управления работой сопроцессора *cwr* (FCTRL)— определяет особенности обработки числовых данных. С помощью полей в регистре ***cwr*** можно регулировать точность выполнения численных вычислений, управлять округлением, маскировать исключения.

Регистр управления сопроцессора CWR состоит из:

- шести масок исключений PM, UM, OM, ZM, DM, IM;
- поля управления точностью PC (Precision Control);
- поля управления округлением RC (Rounding Control).

Маски исключений предназначены для маскирования исключительных ситуаций, возникновение которых фиксируется с помощью шести бит регистра *swr*. Если какие-то маскирующие биты исключений в регистре *swr* установлены в 1, то соответствующие исключения будут обрабатываться самим сопроцессором. Если для какого-либо исключения в соответствующем бите маски исключений регистра *swr* содержится 0, то при возникновении исключения этого типа будет возбуждено прерывание int 16 (10h). Операционная система должна содержать (или программист должен написать) обработчик этого прерывания. Он должен выяснить причину прерывания, после чего, если это необходимо, исправить ее, а также выполнить другие действия.

12. Сопроцессор



2-битовое поле управления точностью **PC** предназначено для выбора длины мантиссы. Возможные значения в этом поле означают:

PC=00 — длина мантиссы 24 бита;

PC=10 — длина мантиссы 53 бита;

PC=11 — длина мантиссы 64 бита.

По умолчанию устанавливается значение поля PC=11.

12. Сопроцессор



Поле управления округлением **RC** позволяет управлять процессом округления чисел в процессе работы сопроцессора. Необходимость операции округления может появиться в ситуации, когда после выполнения очередной команды сопроцессора получается не представимый результат, например, периодическая дробь. Установив одно из значений в поле **RC**, можно выполнить округление в необходимую сторону.

Значения поля **RC** с соответствующим алгоритмом округления:

- 00 — значение округляется к ближайшему числу, которое можно представить в разрядной сетке регистра сопроцессора;
- 01 — значение округляется в меньшую сторону;
- 10 — значение округляется в большую сторону;
- 11 — производится отбрасывание дробной части числа. Используется для приведения значения к форме, которая может использоваться в операциях целочисленной арифметики.

Бит 12 в регистре `swr` физически отсутствует и считывается равным 0.

12. Сопроцессор

Регистр тегов twr (FTAGS)— представляет собой совокупность двухбитовых полей. Каждое поле соответствует определенному физическому регистру стека и характеризует его текущее состояние. Команды сопроцессора используют этот регистр, например, для того, чтобы определить возможность записи значений в эти регистры. Изменение состояния любого регистра стека отражается на содержимом соответствующего этому регистру 2-битового поля регистра тега. Возможны следующие значения в полях регистра тега:

- 00 — регистр стека сопроцессора занят допустимым ненулевым значением;
- 01 — регистр стека сопроцессора содержит нулевое значение;
- 10 — регистр стека сопроцессора содержит одно из специальных численных значений, за исключением нуля;
- 11 — регистр пуст и в него можно производить запись. Это значение в двухбитовом поле регистра тегов не означает, что все биты соответствующего регистра стека должны быть обязательно нулевыми.

Регистр
тегов TWR

1	0
3	2
5	4
7	6
9	8
11	10
13	12
15	14

2. Обратная польская запись

$A = (1 + 2) * 4 + 3$

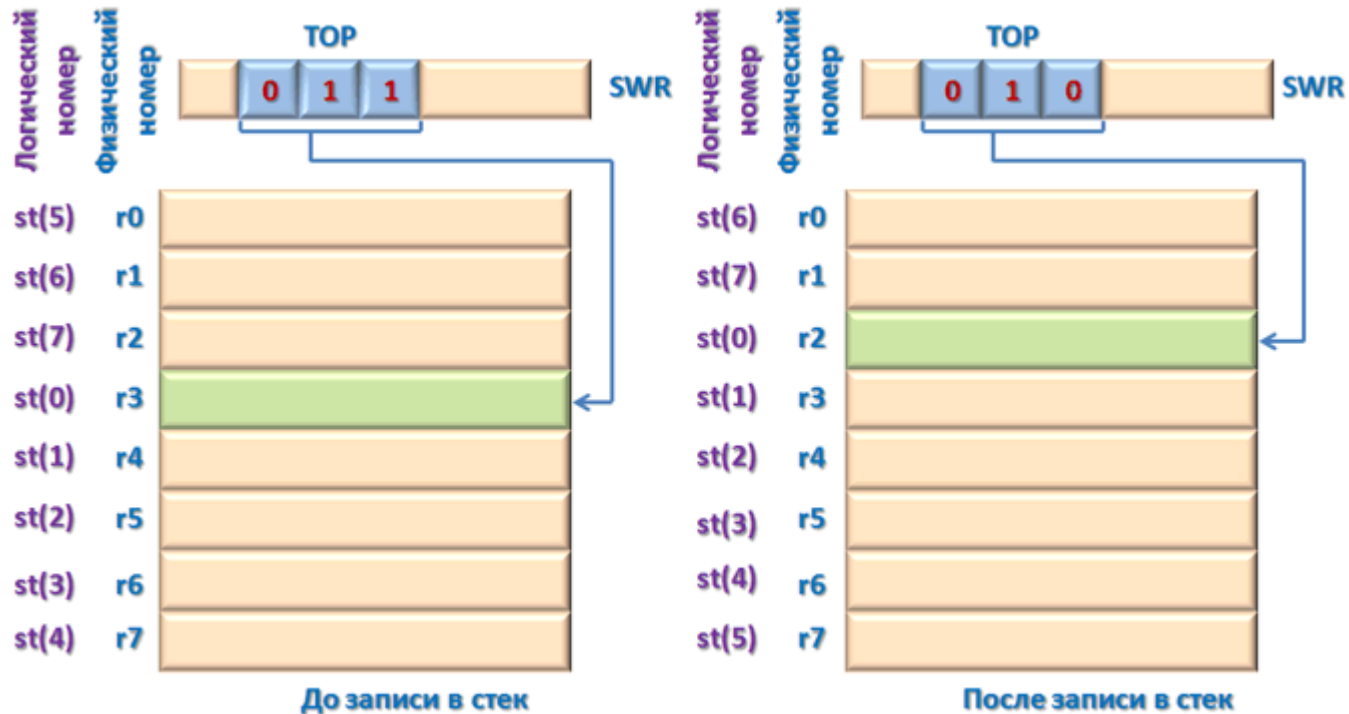
$1\ 2 + 4 \times 3 + A =$

Шаг	Позиция	Инструкции	Содержимое стека
1	1 2 + 4 * 3 + A =	Push 1	1
2	1 2 + 4 * 3 + A =	Push 2	2, 1
3	1 2 + 4 * 3 + A =	pop 2, pop 1, сложить, push 3	3
4	1 2 + 4 * 3 + A =	Push 4	4, 3
5	1 2 + 4 * 3 + A =	Pop 4, pop 3, умножить, push 12	12
6	1 2 + 4 * 3 + A =	Push 3	3, 12
7	1 2 + 4 * 3 + A =	Pop 3, pop 12 сложить, push 15	15
8	1 2 + 4 * 3 + A =	Push A	A, 15
9	1 2 + 4 * 3 + A =	Pop A, pop 15 переместить 15 в ячейку A	-
10		окончание алгоритма	

12. Сопроцессор

Принцип работы сопроцессора

Регистровый стек сопроцессора организован по принципу кольца. Среди восьми регистров, составляющих стек, нет такого, который является вершиной стека. Все регистры стека с функциональной точки зрения абсолютно одинаковы и равноправны. Вершина в кольцевом стеке сопроцессора является плавающей. Контроль текущей вершины осуществляется аппаратно с помощью 3-битового поля top регистра swr.



12. Сопроцессор

В поле **top** фиксируется номер регистра стека **r0...r7**, являющегося в данный момент текущей вершиной стека.

Команды сопроцессора оперируют не физическими номерами регистров стека **r0...r7**, а их логическими номерами **st(0)...st(7)**. С помощью логических номеров реализуется относительная адресация регистров стека сопроцессора. Например, если текущей вершиной до записи в стек является физический регистр стека **r3**, то после записи в стек текущей вершиной становится физический регистр стека **r2**. То есть, по мере записи в стек, указатель его вершины движется по направлению к младшим номерам физических регистров (уменьшается на единицу). Если текущей вершиной является **r0**, то после записи очередного значения в стек сопроцессора его текущей вершиной станет физический регистр **r7**. Что касается логических номеров регистров стека **st(0)...st(7)**, то они перемещаются вместе с изменением текущей вершины стека.

Логическая вершина стека всегда имеет имя st(0).

Поскольку при написании программы разработчик манипулирует не абсолютными, а относительными номерами регистров стека, у него могут возникнуть трудности при попытке интерпретации содержимого регистра тегов **twr**, с соответствующими физическими регистрами стека. В качестве связующего звена необходимо привлекать информацию из поля **top** регистра **swr**. Таким образом реализуется принцип **кольца**.

12. Сопроцессор

Типы команд

- команды передачи данных;
- команды сравнения данных;
- арифметические команды;
- команды трансцендентных функций;
- команды управления сопроцессором.

Команды передачи данных вещественного типа

Используются в случае если операнд, применяемый в команде, имеет вещественный тип (4, 8 или 10-байтный).

Команда	Операнды	Пояснение	Описание
FLD	src	$TOP_{SWR} -= 1; ST(0) = src;$	Загрузка операнда в вершину стека
FST	dst	$dst = ST(0);$	Сохранение вершины стека в память
FSTP	dst	$dst = ST(0); TOP_{SWR} += 1;$	Сохранение вершины стека в память с выталкиванием
FXCH	$ST(i)$	$ST(0) \leftrightarrow ST(i)$	Обмен значений $ST(0)$ и $ST(i)$

12. Сопроцессор

section .data

mvar4 dd 3.14

mvar8 dq 3.14

mvar10 dt 3.14

section .text

global CMAIN

CMAIN:

finit

fld **dword** [mvar4]

fld **qword** [mvar8]

fld **tword** [mvar10]

fld st2

ret

st0	3.1400001049041748	(raw 0x4000c8f5c30000000000)
st1	3.140000000000000001	(raw 0x4000c8f5c28f5c28f5c3)
st2	3.140000000000000001	(raw 0x4000c8f5c28f5c28f800)
st3	3.1400001049041748	(raw 0x4000c8f5c30000000000)
st4	0	(raw 0x00000000000000000000)
st5	0	(raw 0x00000000000000000000)
st6	0	(raw 0x00000000000000000000)
st7	0	(raw 0x00000000000000000000)

12. Сопроцессор

```
finit  
fld dword [mvar4]  
fld qword [mvar8]  
fld tword [mvar10]  
fld st2  
  
fstp dword [mvar4]  
fstp qword [mvar8]  
fstp tword [mvar10]  
fst st0  
ret
```

st0	3.1400001049041748	(raw 0x4000c8f5c30000000000)
st1	0	(raw 0x00000000000000000000)
st2	0	(raw 0x00000000000000000000)
st3	0	(raw 0x00000000000000000000)
st4	0	(raw 0x00000000000000000000)
st5	3.1400001049041748	(raw 0x4000c8f5c30000000000)
st6	3.14000000000000001	(raw 0x4000c8f5c28f5c28f5c3)
st7	3.14000000000000001	(raw 0x4000c8f5c28f5c28f800)
fctrl	0x37f	895
fstat	0x3820	14368
ftag	0x3fff	16383

12. Сопроцессор

Команды передачи данных целого типа Используются в случае если операнд, применяемый в команде, имеет целый тип (1, 2, 4 или 8-байтный).

Команда	Операнды	Пояснение	Описание
FILD	src	$TOP_{SWR} -= 1; ST(0) = src;$	Загрузка операнда в вершину стека
FIST	dst	$dst = ST(0);$	Сохранение вершины стека в память
FISTP	dst	$dst = ST(0); TOP_{SWR} += 1;$	Сохранение вершины стека в память с выталкиванием

12. Сопроцессор

Команда	Операнды	Пояснение	Описание
FADD	dst, src	$dst = dst + src;$	Сложение вещественное
FADDP	ST(i), ST(0)	$ST(i) = ST(i) + ST(0); TOP_{SWR} += 1;$	Сложение вещественное с выталкиванием
FSUB	dst, src	$dst = dst - src;$	Вычитание вещественное
FSUBP	ST(i), ST(0)	$ST(i) = ST(i) - ST(0); TOP_{SWR} += 1;$	Вычитание вещественное с выталкиванием
FSUBR	dst, src	$dst = src - dst;$	Вычитание вещественное реверсивное
FSUBRP	ST(i), ST(0)	$ST(i) = ST(0) - ST(i); TOP_{SWR} += 1;$	Вычитание вещественное реверсивное с выталкиванием
FMUL	dst, src	$dst = dst * src;$	Умножение вещественное
FMULP	ST(i), ST(0)	$ST(i) = ST(i) * ST(0); TOP_{SWR} += 1;$	Умножение вещественное с выталкиванием
FDIV	dst, src	$dst = dst / src;$	Деление вещественное
FDIVP	ST(i), ST(0)	$ST(i) = ST(i) / ST(0); TOP_{SWR} += 1;$	Деление вещественное с выталкиванием
FDIVR	dst, src	$dst = src / dst;$	Деление вещественное реверсивное
FDIVRP	ST(i), ST(0)	$ST(i) = ST(0) / ST(i); TOP_{SWR} += 1;$	Деление вещественное реверсивное с выталкиванием

12. Сопроцессор

Команда	Операнды	Пояснение	Описание
FCOM FUCOM	src	ST(0) — src	Вещественное сравнение
FCOMP FUCOMP	src	ST(0) — src; TOP _{SWR} += 1;	Вещественное сравнение с выталкиванием
FCOMPP FUCOMPP	—	ST(0) — ST(1); TOP _{SWR} += 2;	Вещественное сравнение с двойным выталкиванием
FCOMI FUCOMI	ST, ST(i)	ST(0) — ST(i)	Вещественное сравнение с модификацией EFLAGS
FCOMIP FUCOMIP	ST, ST(i)	ST(0) — ST(i); TOP _{SWR} += 1;	Вещественное сравнение с выталкиванием с модификацией EFLAGS
FXAM	—		Анализ ST(0)

12. Сопроцессор

$$x^2 + 7x + 6 = 0$$

section .data

four DD 4.

two DD 2.

section .bss

a resd 1

b resd 1

c resd 1

x1 resd 1

x2 resd 1

d resd 1

ac resd 1

bb resd 1

section .text

global CMAIN

CMAIN:

mov dword [a], 1

mov dword [b], 7

mov dword [c], 6

INIT ; иниц. 8087

FILD dword [b]

FMUL ST0,ST0

FST dword [bb]

FILD dword [a]

FMUL dword [four]

FIMUL dword [c]

FST dword [ac]

FSUBP ST1,ST0

FST dword [d]

FSQRT

FLD ST0

FCHS

FIADD dword [b]

FCHS

FXCH ST1

FIADD dword [b]

FCHS

FIDIV dword [a]

FDIV dword [two]

FSTP dword [x2]

FIDIV dword [a]

FDIV dword [two]

FSTP dword [x1]

;-----ST(0)----- ! -----ST(1)-----!

; b ! ?

; b*b ! ?

; копирование вершины стека ==> bb

; a ! b*b

; 4*a ! b*b

; 4*a*c ! b*b

; копирование вершины стека ==> ac

; **d=b*b-4*a*c** ! ?

; копирование вершины стека ==> d

; sqrt(d) ! ?

; sqrt(d) ! sqrt(d)

; -sqrt(d) ! sqrt(d)

; b-sqrt(d) ! sqrt(d)

; -b+sqrt(d) ! sqrt(d)

; sqrt(d) ! -b+sqrt(d)

; b+sqrt(d) ! -b+sqrt(d)

; -b-sqrt(d) ! -b+sqrt(d)

; -b-sqrt(d)/a ! -b+sqrt(d)

; **-b-sqrt(d)/a/2** ! -b+sqrt(d)

; -b+sqrt(d) ! ?

; -b+sqrt(d)/a ! -b+sqrt(d)

; **-b+sqrt(d)/a/2** ! ?

1. Команды сопроцессора

$$x^2 + 7x + 6 = 0$$

$$x1 = -6, x2 = -1$$

x1	-1	Float ▾	d ▾
x2	-6	Float ▾	d ▾

st0	0	(raw 0x000000000000000000000000)
st1	0	(raw 0x000000000000000000000000)
st2	0	(raw 0x000000000000000000000000)
st3	0	(raw 0x000000000000000000000000)
st4	0	(raw 0x000000000000000000000000)
st5	0	(raw 0x000000000000000000000000)
st6	-6	(raw 0xc001c0000000000000000000)
st7	-1	(raw 0xbffff8000000000000000000)
fctrl	0x37f	895
fstat	0x0	0
ftag	0xffff	65535

Спасибо