Машинно-зависимые языки программирования

Савельев Игорь Леонидович

План

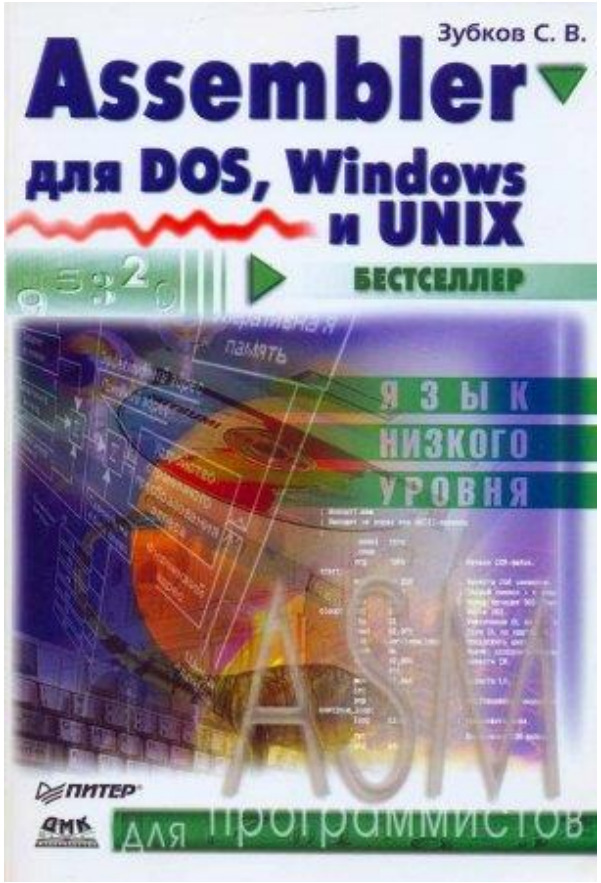
- Обучение
- Литература, введение
- Элементы схемотехники
- Программная модель процессоров Intel. Регистры процессоров и их
- назначение, способы адресации

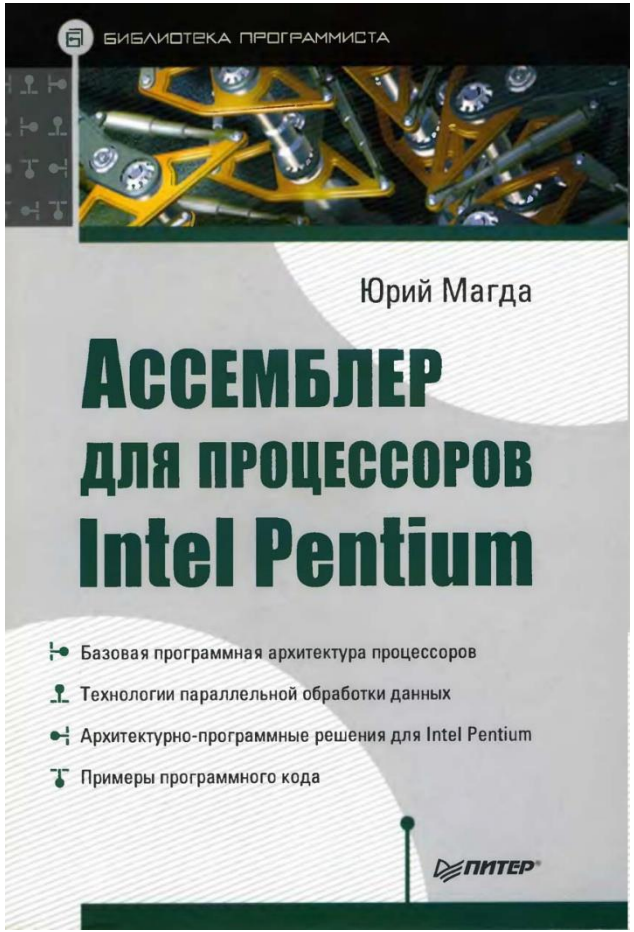
1. Обучение

- 2 семестра
- (1с) 5 лабораторных, 80x86
  - Арифметические операции
  - Условные переходы
  - Ввод-вывод на АССЕМБЛЕРЕ
  - Обработка массивов
  - Сопроцессор

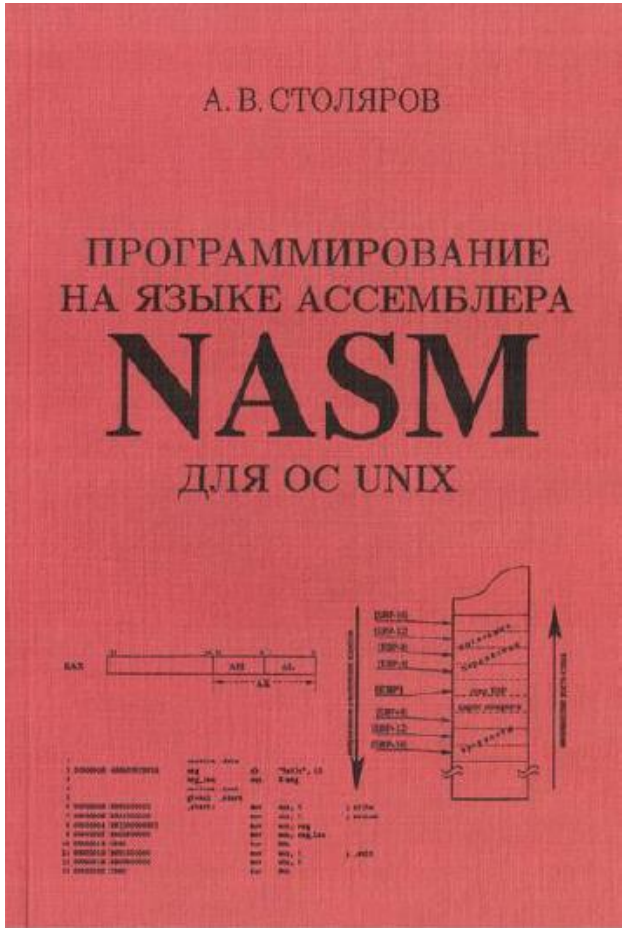
2. Литература

2. Литература





2. Литература



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

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

https://www.opennet.ru/

https://stackoverflow.com/

Денис Юричев. Reverse engineering для начинающих. (https://beginners.re/)

https://dzen.ru/a/X2uFSG0z7E66hLAG

Old Programmer
5,9K подписчиков

Подписаться

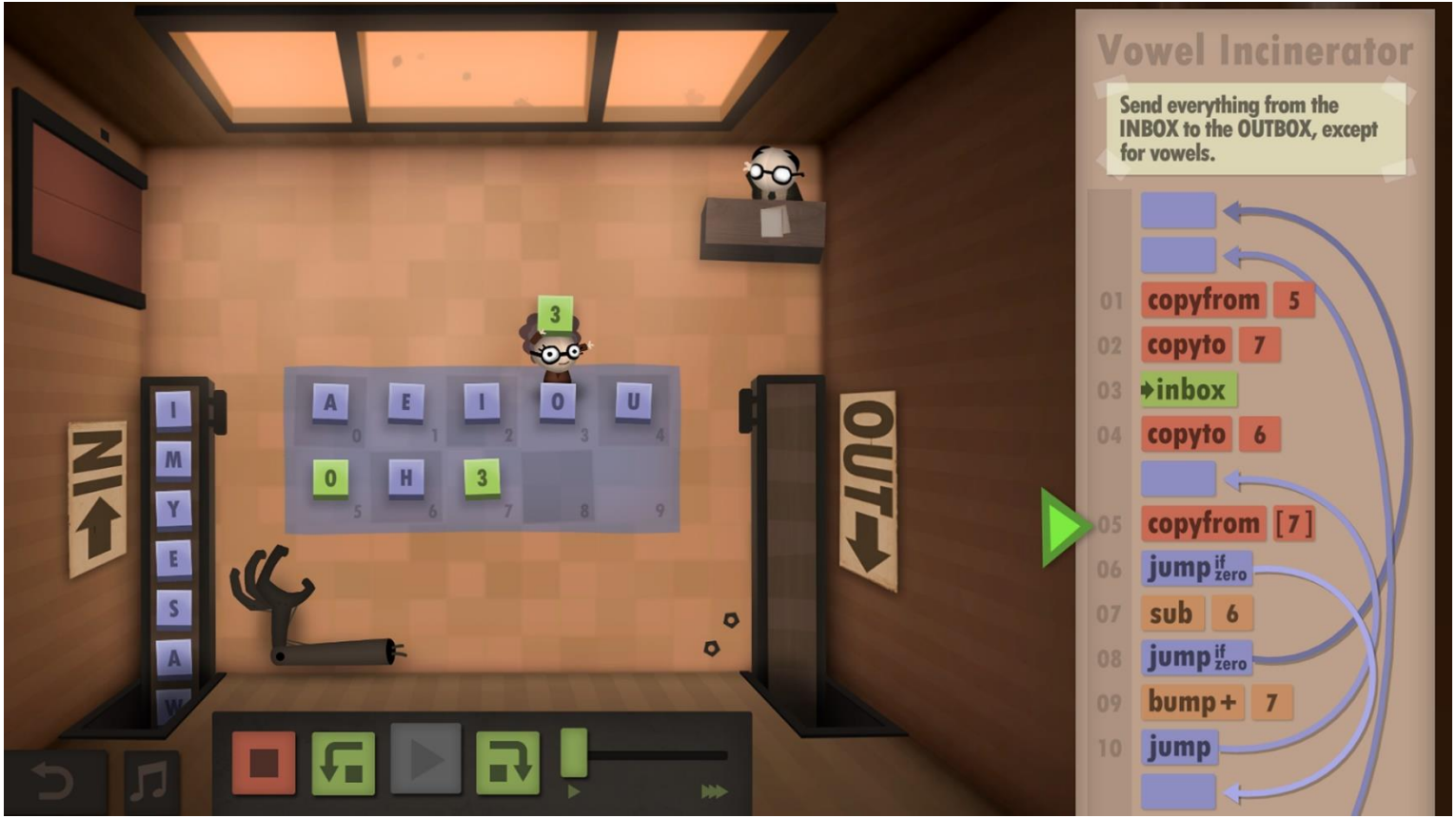
Язык ассемблера (x86-64, Linux). Путеводитель по ресурсам канала Old Programmer

23 сентября 2020 · 1K прочитали

Весь мой канал Old Programmer о программировании и программистах представлен по темам здесь.

О моих ресурсах по языку ассемблер

Human Resource Machine

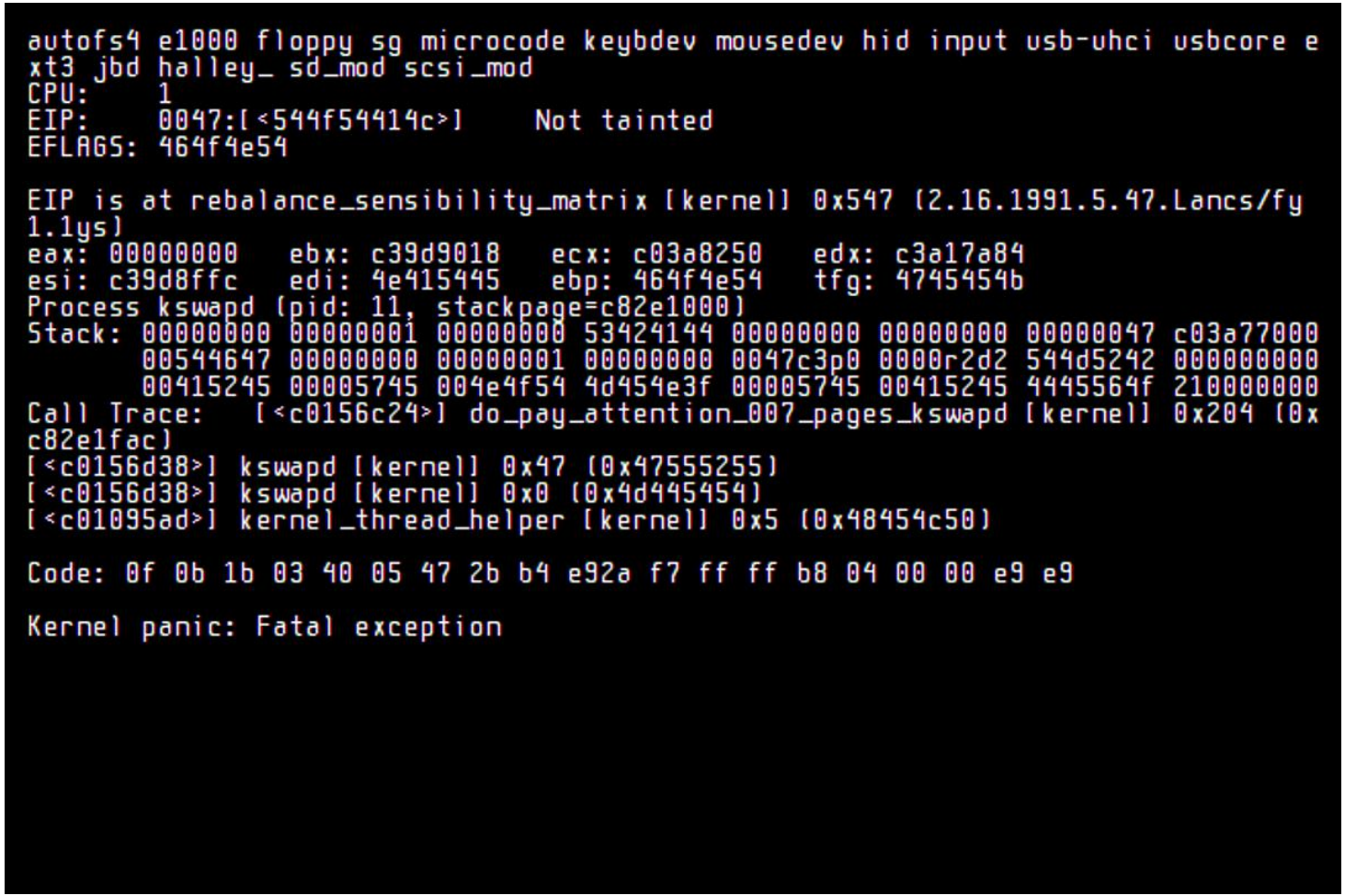


Причины использования языка ассемблер

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. Введение


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.
- Драйверы и системный код. Обращение к аппаратному обеспечению, регистрам и т.д. достаточно сложно организовать на языке высокого уровня.
- Вызов инструкций, которые недоступны в языках высокого уровня. Некоторые инструкции ассемблера не имеют эквивалентов в языках высокого уровня

Причины использования языка ассемблер

- Оптимизация кода по размеру. В настоящее время требования к размеру программы, налагаемые вследствие ограниченности размера памяти, не существенны. Однако, для работы с некоторыми ограниченными ресурсами (например, кэш-память сетевой платы) лучше использовать небольшой код.
- Оптимизация кода по скорости. Современные C++ компиляторы оптимизируют код очень хорошо в большинстве случаев. Но бывает, что и эти компиляторы не могут сделать программу такой быстрой, какой её можно сделать на ассемблере, правильно расположив команды.

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 = 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 = bits << 1;
      i++;
     goto @begin:
   }
@end:
   return (res – i);
 }
```

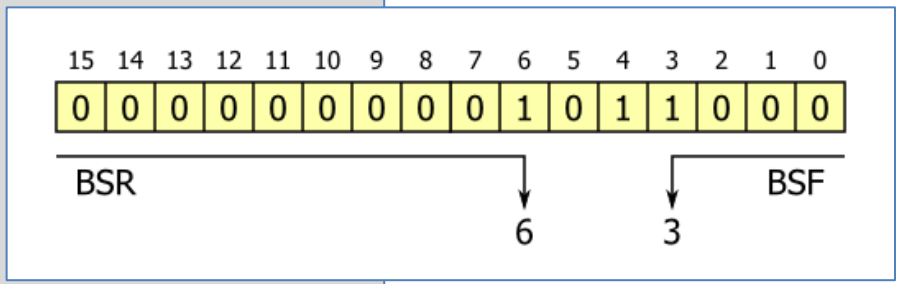
Пример (opal\plugins\audio\G.722.1\libg722_1\src\basop32.c)

```
/* 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;
}
```

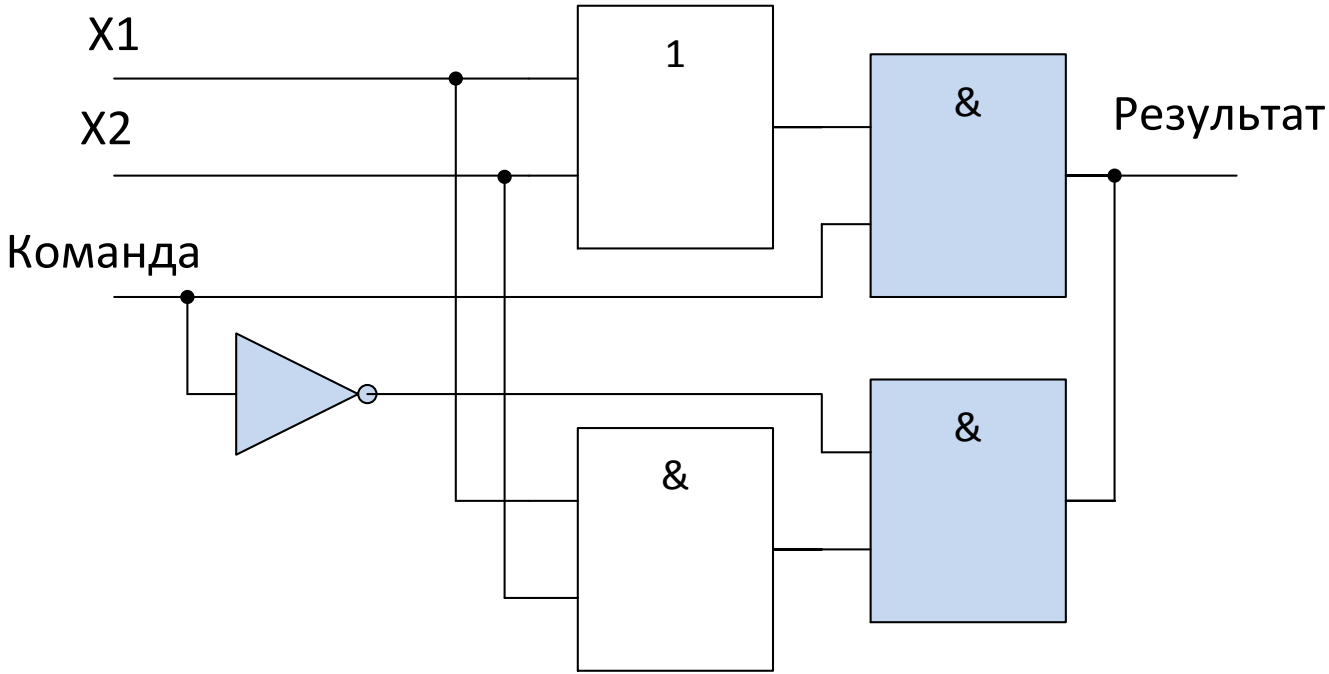
Пример

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;

#if defined(__i386__)  ||  defined(__x86_64__)
   __asm__ (" xorl %[res],%[res];\n"
                  " decl %[res];\n"
                  " bsrl %[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;
```



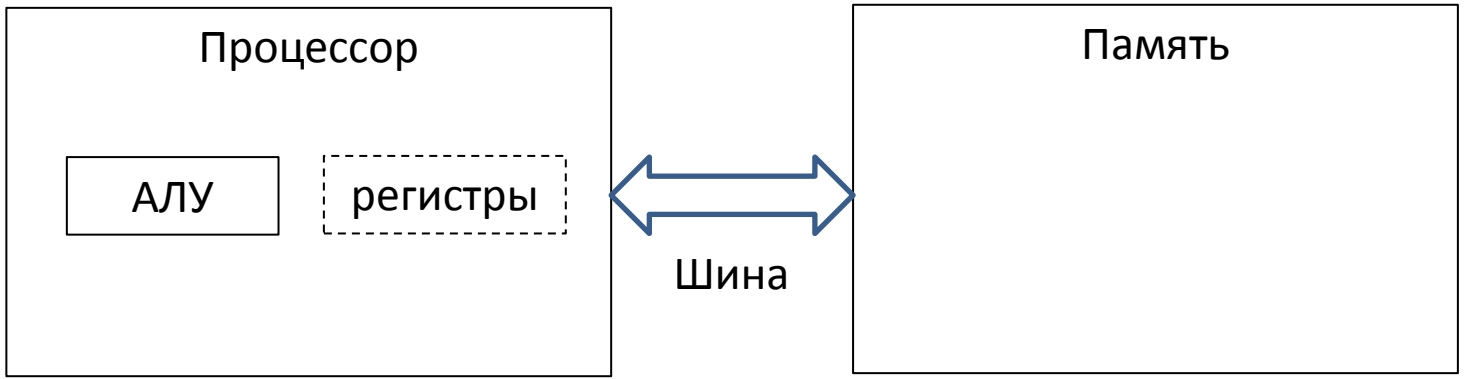
3. Элементы схемотехники. Вычислитель



3. Элементы схемотехники. Итог

1. Для вычислений нужен некий «вычислитель» (арифметико-логическое устройство)
2. Для хранения данных можно использовать регистры и ячейки на триггерах
3. Для согласованности выполнения нужно использовать тактирование
4. Логика работы вычислителя может быть жесткой, а может настраиваться (программироваться)

4. Программная архитектура процессора



4. Программная архитектура процессора

Память

- Что храним (данные, инструкции)

Фон-Неймановская архитектура

Гарвардская архитектура

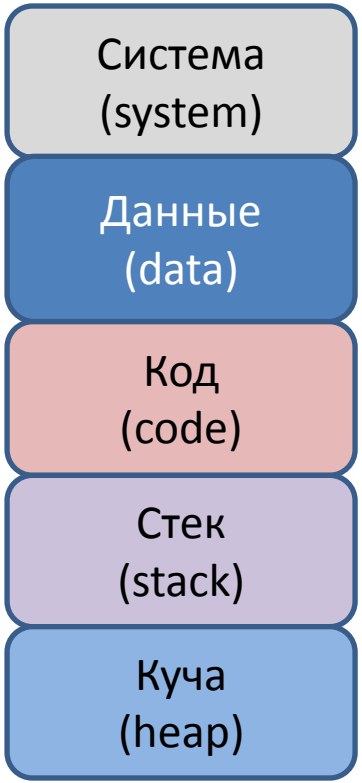
- Тип доступа (чтение, чтение-запись)

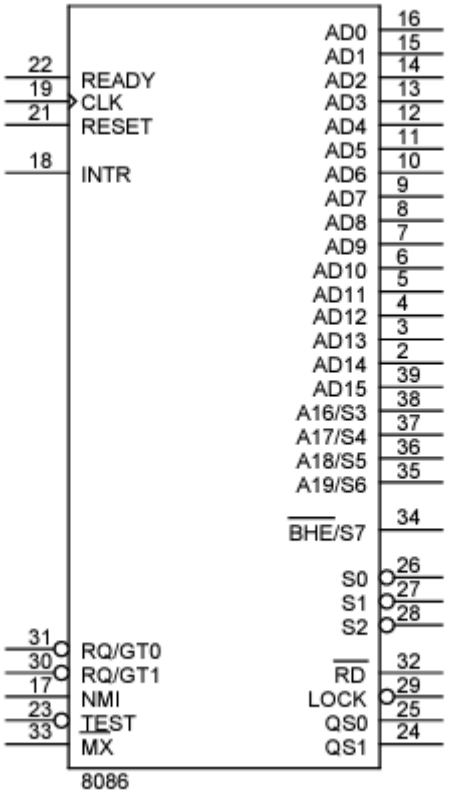
ROM, RAM, ПЗУ, ППЗУ

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

- Сегменты

- Где храним код?





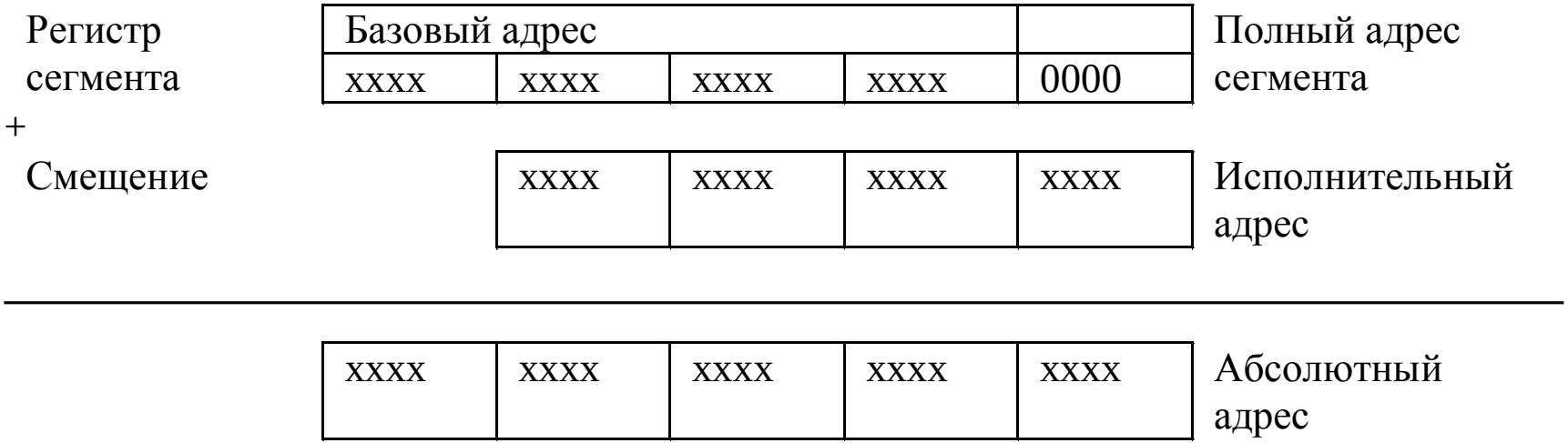
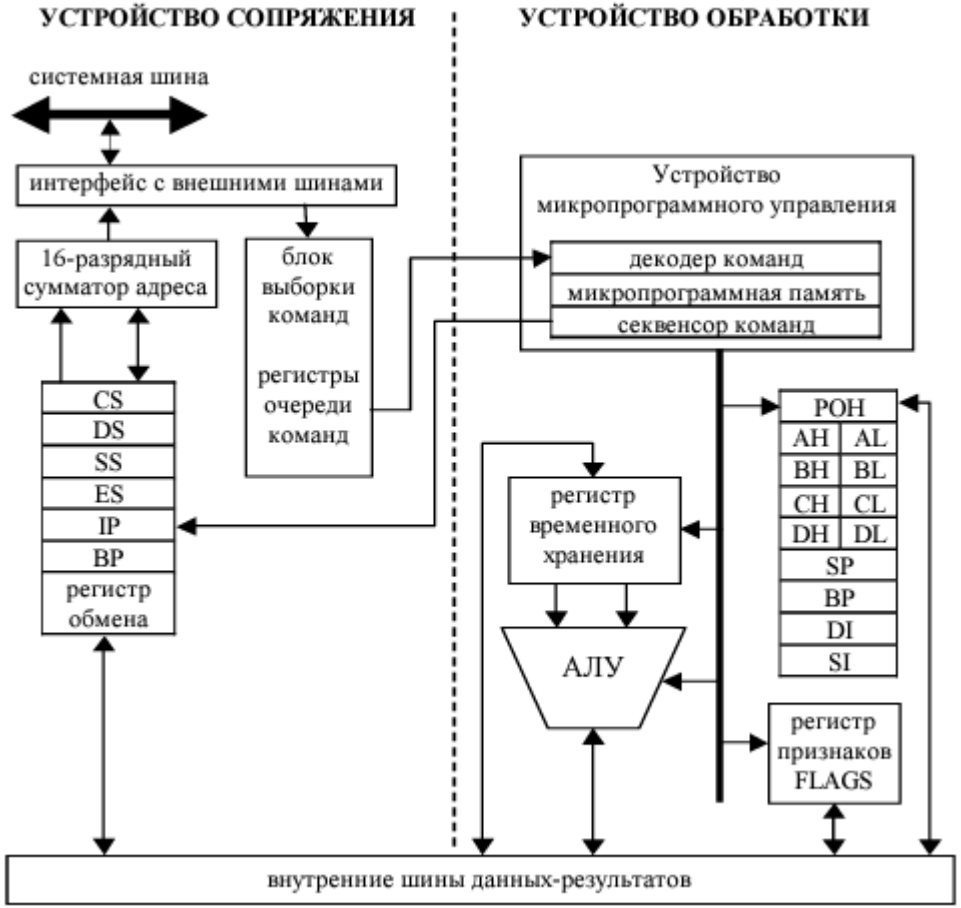
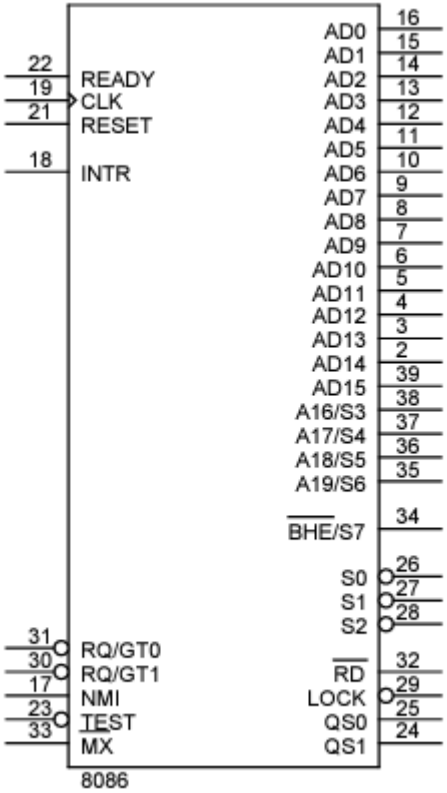


Рис. 1.2. Принцип получения абсолютного адреса.

! Одному физическому адресу соответствует 4096 «логических» адресов

3. Программная архитектура процессора





3. Программная архитектура процессора

```
 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
- Мнемокод
- Ассемблер

3. Программная архитектура процессора

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
```

Низкоуровневое программирование

Низкоуровневое программирование – это программирование, основанное на прямом использовании возможностей и особенностей конкретной вычислительной системы. Для того чтобы писать программы на этом уровне, необходимо знать архитектуру аппаратной части системы:

- структуру и функционирование системы в целом;
- организацию оперативной памяти;
- состав внешних устройств, их адреса и форматы регистров;
- организацию и функционирование процессора, состав и форматы его регистров, способы адресации, систему команд;
- систему прерываний и т. д.

При этом не следует забывать, что вычислительная система – это совокупность не только аппаратных, но и программных средств, и особенности имеющегося программного обеспечения (операционной системы) оказывают существенное влияние на разработку программ.

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:

4. Архитектура процессора.

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:

- **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.

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.

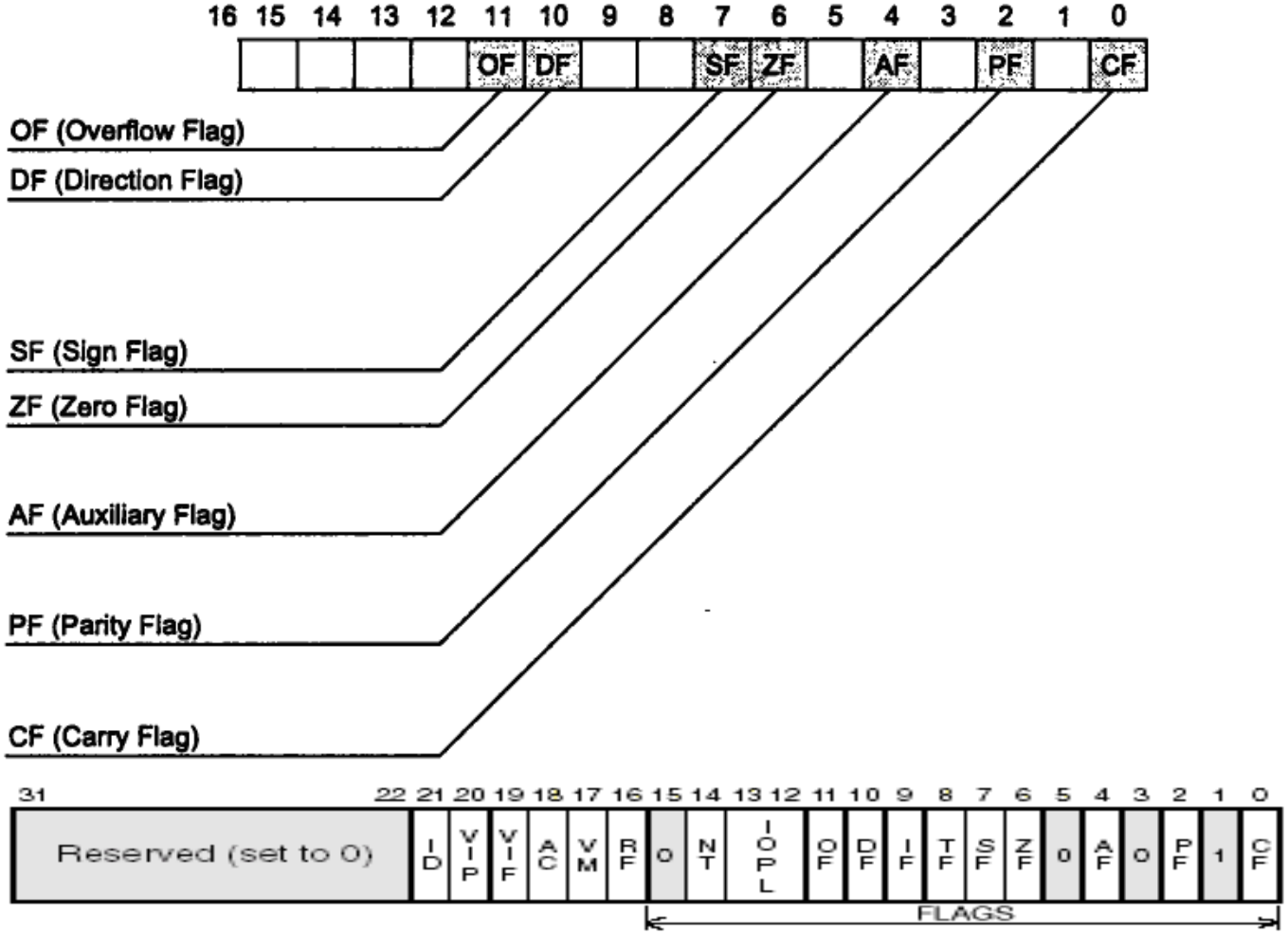
Регистр указателя стека. Указатель стека **SP** – это 16-битовый регистр, который определяет смещение текущей вершины стека. Указатель стека SP вместе с сегментным регистром стека SS используются микропроцессором для формирования физического адреса стека.

Стек всегда растет 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»).

- ***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.

4. Архитектура процессора.

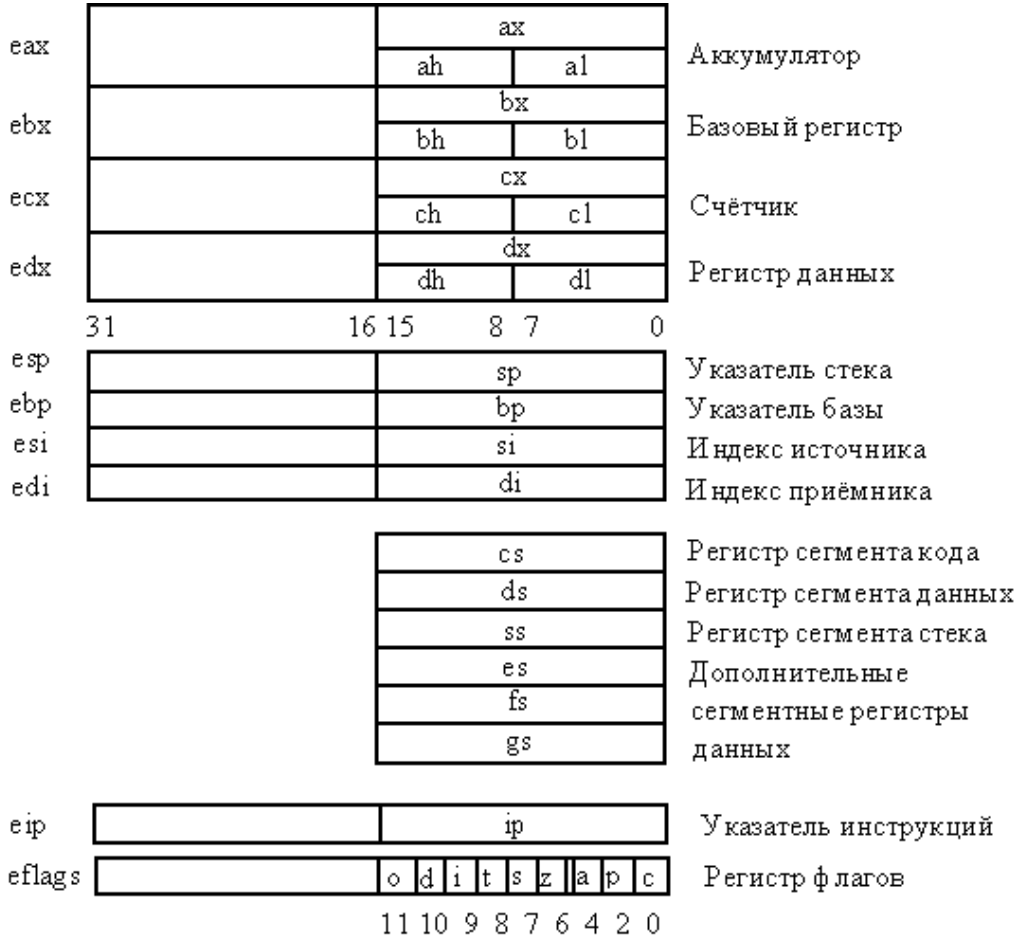


Указанные на рисунке флаги наиболее часто используются в прикладных программах и сигнализируют о следующих событиях:

- 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.

4. Архитектура процессора.



4. 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 64Kb 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

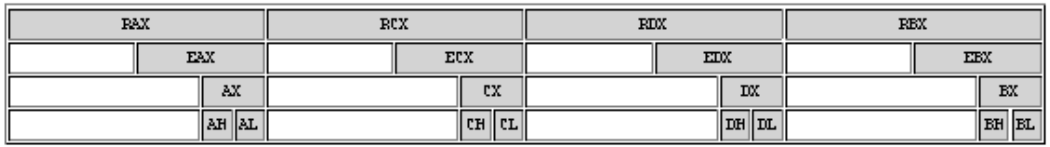
EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, R8D — R15D — 32-битные (extended AX)

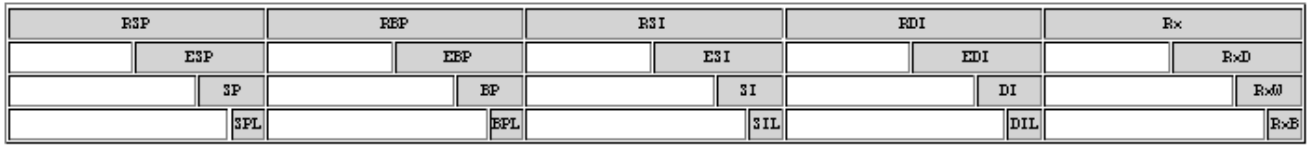
AX, CX, DX, BX, SP, BP, SI, DI, R8W — R15W — 16-битные

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





где x — 8..15.

Регистры RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, Rx, RxD, RxW, RxB, 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

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	db 0
		Var1 db 5
		Var2 db ‘a’
		Var3 db -1
		Var4 db 1.3
		
		MasQ DB 1, 2
			
			char var1 = 5;
			unsigned char var2 = ‘a’;
			char var3 = -1;
			
			char MasQ[2] = {1, 2};

Длина (бит)	Директива	Пример	Пример Си(*)

16	DW	dw 0
		Var1 dw 5
		Var2 dw ‘a’
		Var3 dw -1
		Var4 dw 1.3
			short var1 = 5;
			unsigned short var2 = ‘a’;
			short var3 = -1;

32	DD	dd 0
		Var1 dd 5
		Var2 dd ‘a’
		Var3 dd -1
		Var4 dd 1.3
			int var1 = 5;
			unsigned int var2 = ‘a’;
			int var3 = -1;
			float var4 = 1.3;

Длина (бит)	Директива	Пример	Пример Си (*)

64	DQ	dq 0
		Var1 dq 5
		Var2 dq ‘a’
		Var3 dq -1
		Var4 dq 1.3
			long var1 = 5;
			unsigned long var2 = ‘a’;
			long = -1;
			double var4 = 1.3;

80	DT	dt 0 Var1 dt 5 Var2 dt ‘a’ Var3 dt -1 Var4 dt 1.3	long double var4 = 1.3;

(*) Важно!

В nasm 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 ; 1.0x2^2 = 4.0
dq 0x1p+32 ; 1.0x2^32 = 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 3.14159265358979323846 ; 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];
	…	

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.

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**
MET:	MOV	AX, BX	;Пересылка

Инструкция определяется
- Размер в памяти
- Скорость вычисления

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

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 AH помещаем 4Ch, в AL – 00h.
   int 21h           ;Завершение программы
   ;-------------------------------------------------
   hello db 'Hello, world!$‘
CodeSg ends
end start
```

Спасибо