АРХИТЕКТУРА ЭВМ
конспект лекций

СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

Тема 1

Системы счисления.

Представление информации в ЭВМ

Основные понятия

Цифрами называется набор символов, с помощью которых записываются числа. Количество этих символов определяет **основание** системы счисления. Так, в десятичной системе имеется 10 цифр (0,1,2,3,4,5,6,7,8,9), а в двоичной – 2 цифры (0 и 1).

Система счисления называется **позиционной,** если в записи числа каждая цифра имеет свой вес.

Рассмотрим 4-х разрядное десятичное число – 7823. В нем: 7 тысяч; 8 сотен; 2 десятка и 3 единицы.

Величина числа определяется по формуле:

$$D = 7\cdot10^3+8\cdot10^2+2\cdot10^1+3\cdot10^0$$

или в общем виде для целого числа :

$$D = d_3\cdot10^3+d_2\cdot10^2+d_1\cdot10^1+d_0\cdot10^0$$

Каждая цифра d_i этого десятичного числа имеет вес, равный 10^i . Число 10 является основанием системы счисления.

В общем случае, в системе счисления с основанием b произвольное число, имеющее целую и дробную части записывается с помощью цифр d_i (их количество равно основанию системы) следующим образом:

$$d_{p-1}d_{p-2}\cdots d_1d_0.d_{-1}d_{-2}\cdots d_{-n},$$

где p – число цифр, расположенных слева , а n – число цифр, расположенных справа от точки, отделяющей целую часть числа от дробной.

Основные понятия

Значение числа представляет собой сумму произведений отдельных цифр на основание системы счисления в соответствующей степени:

$$D = \sum_{i=-n}^{p-1} d_i \cdot b^i$$

В позиционной системе счисления крайняя левая цифра называется **цифрой старшего разряда**, а крайняя правая – **цифрой младшего разряда**.

В вычислительной технике используется двоичная система счисления. Цифры 0 и 1 называют битами. Каждый бит, стоящий в i-той позиции числа имеет вес 2^i .Например,

$$10001_2=1\cdot16+0\cdot8+0\cdot4+0\cdot2+1\cdot1=17_{10}$$

В двоичной системе используются понятия **бит старшего разряда** (старший бит) и **бит младшего разряда** (младший бит). Группу из трех рядом стоящих двоичных разрядов называют **триадой**, из четырех разрядов – **тетрадой**, а восьми разрядов – **байтом.**

Используемые системы счисления

Для пользователей ЭВМ кроме систем счисления 2 и 10 удобны восьмеричная (основание 8) и шестнадцатеричная (основание 16) системы счисления. Ниже в таблице приведено соответствие между числами указанных систем счисления.

Десятичные	Двоичные	Восьмеричные	Шестнадцатеричные
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Перевод чисел между системами с основанием 2^n

Восьмеричная и шестнадцатеричная системы счисления удобны для представления двоичных чисел, т.к. их основания представляют собой степени числа 2.

Для преобразования двоичного числа в восьмеричное (шестнадцатеричное) нужно двоичное число, начиная от точки влево и вправо разбить на триады (тетрады) и каждую из них записать соответствующей восьмеричной (шестнадцатеричной) цифрой. При этом неполные группы слева и справа для удобства дополняются незначащими нулями.

$$11.1010011011_2 = 011\ .\ 101\ \ 001\ \ 101\ \ 100_2 = 3.5154_8$$
$$11.1010011011_2 = 0011\ .\ 1010\ \ 0110\ \ 1100_2 = 3.A6C_{16}$$

Для обратного преобразования $8 \rightarrow 2$ или $16 \rightarrow 2$ необходимо каждую восьмеричную или шестнадцатеричную цифру записать соответствующим эквивалентом из 3 или 4 бит.

Перевод числа из десятичной системы в двоичную

1. ***Для целой части числа***:

Последовательно делим ***целую часть*** на основание новой системы счисления, записывая получившиеся остатки. Цифры остатков объединяем в обратном порядке (первый остаток – младший разряд, последний остаток – старший разряд).

Частное от деления на 2	Остатки
300	0 (мл.бит)
150	0
75	1
37	1
18	0
9	1
4	0
2	0
1	1 (ст.бит)
0 признак окончания	

$$300_{10}=100101100_2$$

Перевод числа из десятичной системы в двоичную

2. Для дробной части числа:

Последовательно умножаем ***дробную часть*** числа на основание новой системы счисления, каждый раз отделяя целые части произведений. Перевод заканчивается при достижении требуемой точности.

Целые части произведений	
(ст.бит) 0.	5408 × 2
1.	0816 × 2
0.	1632 × 2
0.	3264 × 2
0.	6528 × 2
1.	3056 × 2

0.	6112 × 2
1.	2224 × 2
0.	4448 × 2
0.	8896 × 2
1.	7792 × 2
1.	5584 × 2
(мл.бит) 1.	1168

$$0{,}5408_{10} = 0.\ 1000\ \ 1010\ \ 0111_2$$

Перевод числа из двоичной системы в десятичную

Для перевода двоичного числа в десятичное число нужно воспользоваться формулой:

$$D = \sum_{i=-n}^{p-1} d_i \cdot b^i$$

Преобразование можно выполнить отдельно для целой части числа и дробной части, а затем результаты сложить.

Целая часть:

$100101100_2 = 1\cdot2^8+0\cdot2^7+0\cdot2^6+1\cdot2^5+0\cdot2^4+1\cdot2^3+1\cdot2^2+0\cdot2^1+0\cdot2^0 = 256+32+8+4 = 300_{10}$

Дробная часть:

$0.100010100111_2 = 1\cdot2^{-1}+0\cdot2^{-2}+0\cdot2^{-3}+0\cdot2^{-4}+1\cdot2^{-5}+0\cdot2^{-6}+1\cdot2^{-7}+0\cdot2^{-8}+0\cdot2^{-9}+1\cdot2^{-10}+1\cdot2^{-11}+1\cdot2^{-12} =$

$=0.5+0+0+0+0.031+0+0.0078125+0+0+0.0009765625+0.00048828125+0.000244140625 =$

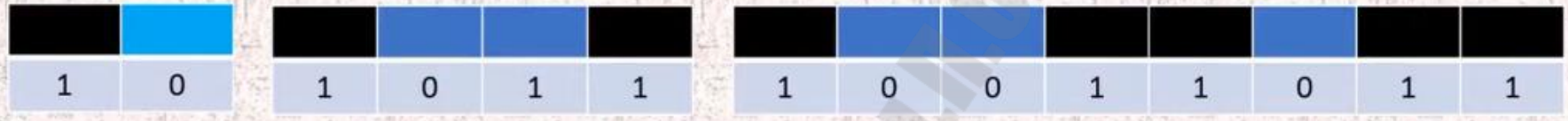
$=0.54077148$

Результат:

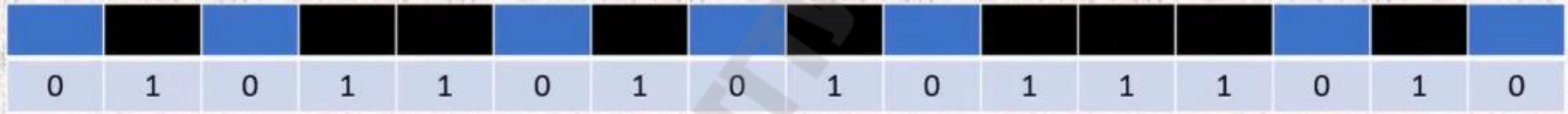
$$100101100.100010100111_2 = 300.54077148_{10}$$

Представление чисел в ЭВМ

Двоичная система счисления — позиционная **система счисления** с основанием **2**, то есть значение может быть либо 0, либо 1



2 бита 4 бита 8 бит (1 байт) DB(*)



16 бит (“машинное” слово, WORD) DW (*)

32 бит (двойное слово, DWORD) DD (*)

* типы данных в ассемблере

Отрицательные числа в прямом коде

Одним из способов представления двоичных чисел со знаком является **прямой код**. Для представления знака вводится дополнительный знаковый разряд: «0» это «+»; «1» это «–».

$$\mathbf{0}010\ 1011_2 = +\ 43_{10} \qquad \mathbf{1}010\ 1011_2 = -\ 43_{10}$$

Система представления чисел в прямом коде содержит одинаковое количество положительных и отрицательных чисел, причем ноль может быть представлен двумя способами, как +0 и –0 (оба значения совпадают).

Правила сложения двух чисел в прямом коде:

– если знаки чисел совпадают, то их величины складываются, а результату присваивается знак слагаемых;

– если знаки разные, то из большей величины вычитается меньшая, а результату присваивается знак большей величины.

Таким образом, необходимо производить проверку знаков слагаемых.

Отрицательные числа в дополнительном коде

В ЭВМ отрицательные числа обычно представляются в виде ***дополнений*** до основания системы счисления.

При двоичной системе это ***дополнение до двух*** такое, что ***сумма n-разрядного двоичного числа D и его дополнения равна 2^n.*** Поэтому для нахождения дополнения числа *D* нужно найти разность $2^n - D$.

Например, 8-битное двоичное число со знаком можно представить в виде семи (6…0) значащих разрядов и восьмого (7) знакового:

номера битов	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
	0	1	1	0	0	1	0	1
знак и веса битов	*+/-*	*64*	*32*	*16*	*8*	*4*	*2*	*1*

здесь $D = \mathbf{0}110\ 0101_2 = +101_{10}$.

Определим дополнение числа *D*. Для этого вычислим разность $2^8 - D$:

2^8	1	0	0	0	0	0	0	0	0
-									
D		0	1	1	0	0	1	0	1
		1	0	0	1	1	0	1	1

Непосредственно вычитать сложно, т.к. приходится делать заем из старшего разряда.

Отрицательные числа в дополнительном коде

Для упрощения вычислений преобразуем исходное выражение:

$$2^n - D = [(2^n - 1) - D] + 1.$$

Поскольку $2^8 = 1\ 0000\ 0000_2$, то число $2^n - 1 = 1111\ 1111_2$.

Из такого числа вычитать легче, т.к. ни в одном из разрядов, независимо от величины вычитаемого, заем делать не надо. Разность $(2^n - 1) - D$ представляет собой ***инвертированное*** значение числа *D*, т.е. ***обратный код*** числа *D*.

ПРАВИЛО*: для получения дополнения до двух некоторого числа D необходимо получить его обратный код, а затем к нему прибавить единицу младшего разряда.*

Достоинством представления отрицательных чисел в дополнительном коде является то, что ***при сложении чисел не нужно проверять их знак*** (как в случае использования прямого кода). Всегда выполняется двоичное сложение и, если результат не выходит за границы разрядной сетки представления числа, то результат всегда получается верным.

Вычитание двоичных чисел заменяется сложением уменьшаемого с дополнительным кодом вычитаемого.

Двоичные числа с плавающей точкой

Вещественное число = $(-1)^s \, 2^{e-127} \, (1.f)$

s: знак

e: показатель степени

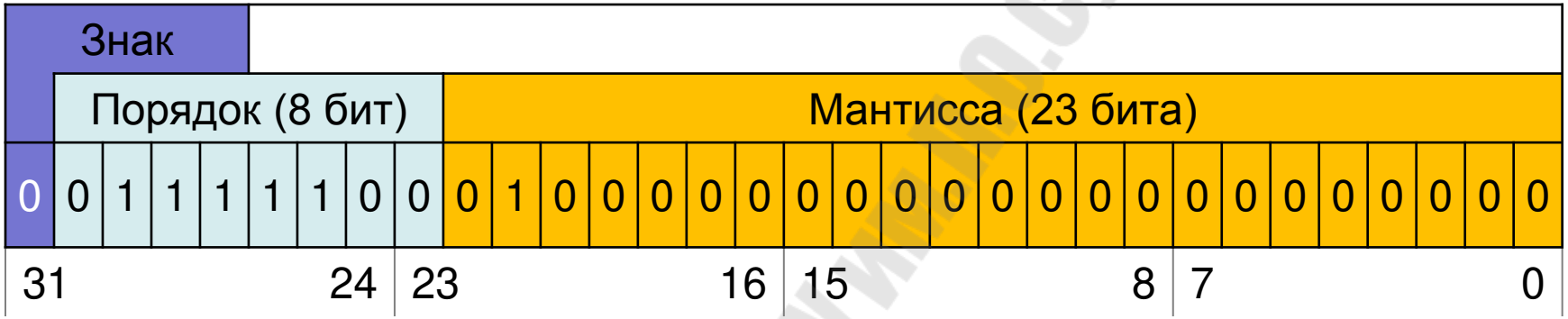
f: мантисса

Данные	Кол-во битов	Содержание
s: знак	1	0: «+»; 1: «-»
e: показатель степени	8	Показатель степени (e) может принимать значения от 0 до 255. Фактическое значение показателя рассчитывается путем вычитания 127 из e и лежит в диапазоне -127...128. При этом значениям «e=0» и «e=255» соответствуют особые состояния.
f: мантисса	23	Мантисса двоичного значения с плавающей запятой удовлетворяет неравенству 2,0 > 1, f ≥1,0.

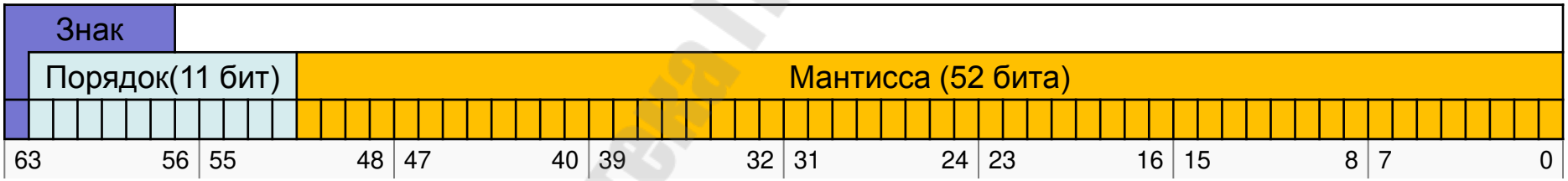
Двоичные числа с плавающей точкой

Формат представления чисел с плавающей запятой описан в стандарте EEE754:

– ***с одинарной точностью*** (single-precision) 32 бита;



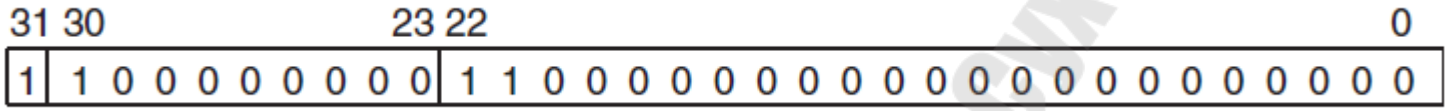
– ***с двойной точностью*** (double-precision) 64 бита;



– с одинарной расширенной точностью (single-extended precision) >=43 бит (редко используемый);
– с двойной расширенной точностью (double-extended precision) >= 79 бит (обычно используют 80 бит).

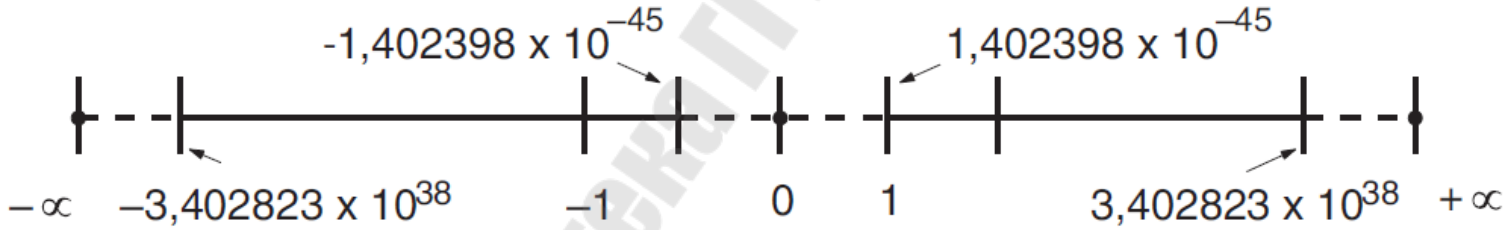
Двоичные числа с плавающей точкой

Пример.



Знак: -
Показатель: 128 - 127 = 1
Мантисса: $1 + (2^{-1} + 2^{-2}) = 1 + 0{,}75 = 1{,}75$
Значение: $-1{,}75 \times 2^{1} = -3{,}5$

В формате с одинарной точностью может быть представлен следующий ***диапазон чисел***:



Для особых состояний, которыми являются NaN, $\pm\infty$ и 0, используются следующие форматы записи:

NaN*: e = 255, f ≠ 0
+∞: e = 255, f = 0, s= 0
–∞: e = 255, f = 0, s= 1
0: e = 0

Порядок байт

Преобразуем число $723{,}45_{10}$ к формату с плавающей точкой IEEE754:

Нормализуем мантиссу (приведем к виду 1.xxx...x * 2^exp)

Шаг	Мантисса	* 2^двоичный порядок
0	723.45	* 2^0
1	361.725	* 2^1
2	180.8625	* 2^2
3	90.43125	* 2^3
4	45.215625	* 2^4
5	22.6078125	* 2^5
6	11.30390625	* 2^6
7	5.651953125	* 2^7
8	2.8259765625	* 2^8
9	1.41298828125	* 2^9

x = +1.41298828125 * 2^9

Переводим нормализованную мантиссу в двоичный вид (p = 23)

Шаг i	Двоичная мантисса	остаток * 2^-i
0	1.	+ 0.41298828125 * 2^-0
1	1.0	+ 0.8259765625 * 2^-1
2	1.01	+ 0.651953125 * 2^-2
3	1.011	+ 0.30390625 * 2^-3
4	1.0110	+ 0.6078125 * 2^-4
5	1.01101	+ 0.215625 * 2^-5
6	1.011010	+ 0.43125 * 2^-6
7	1.0110100	+ 0.8625 * 2^-7
8	1.01101001	+ 0.725 * 2^-8
9	1.011010011	+ 0.45 * 2^-9
10	1.0110100110	+ 0.9 * 2^-10
11	1.01101001101	+ 0.8 * 2^-11
12	1.011010011011	+ 0.6 * 2^-12
13	1.0110100110111	+ 0.2 * 2^-13
14	1.01101001101110	+ 0.4 * 2^-14
15	1.011010011011100	+ 0.8 * 2^-15
16	1.0110100110111001	+ 0.6 * 2^-16
17	1.01101001101110011	+ 0.2 * 2^-17
18	1.011010011011100110	+ 0.4 * 2^-18
19	1.0110100110111001100	+ 0.8 * 2^-19
20	1.01101001101110011001	+ 0.6 * 2^-20
21	1.011010011011100110011	+ 0.2 * 2^-21
22	1.0110100110111001100110	+ 0.4 * 2^-22
23	1.01101001101110011001100	+ 0.8 * 2^-23

Порядок байт

```c
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <inttypes.h>

int main()
{
    setlocale(0,"");

    float num;
    printf("float: ");
    scanf("%f", &num);

    uint8_t size = sizeof(float);
    uint8_t* ptr = (uint8_t*)&num;
    printf("hex: 0x");
    for(uint8_t i = 0; i < size; i++)
    {
        printf("%02X", *(ptr + i));
    }

    printf("\nГотово");
    return 0;
}
```

Знак	Смещенный порядок (e + 127)	Дробная часть мантиссы
0	10001000	01101001101110011001101

x = 0x4434dccd

```
float: 723,45
hex: 0xCDDC3444
Готово
Process returned 0 (0x0)   execution time : 24.022 s
Press any key to continue.
```

Порядок байт


```c
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

int main()
{
    uint16_t x = 0x0001;
    printf("%s-endian\n", *((uint8_t*) &x) ? "little" : "big");
    return 0;
}
```

```
little-endian

Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

Двоично-десятичные числа

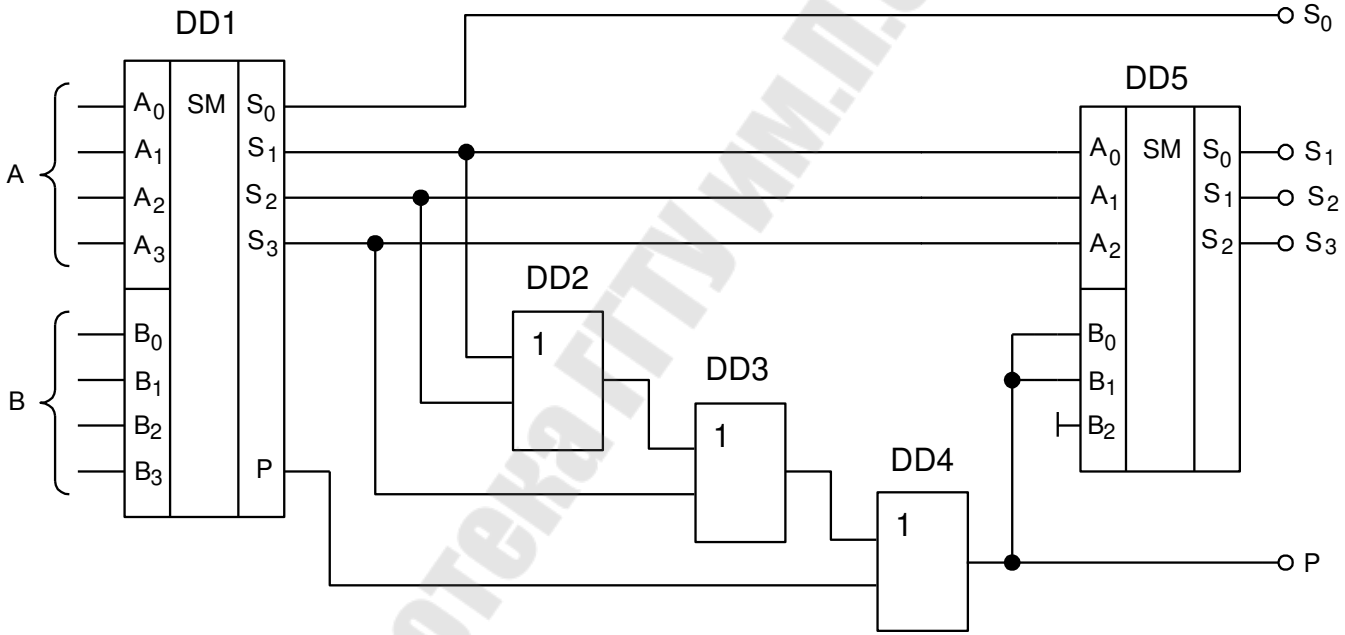
Особое место занимает двоично-десятичная система представления чисел. В ней десятичные цифры от 0_{10} до 9_{10} представляются 4-х разрядными двоичными комбинациями от 0000_2 до 1001_2; комбинации от 1010_2 до 1111_2 не используются.

Эта система нашла широкое применение во многих устройствах ввода-вывода десятичных данных. Многие ЭВМ имеют полный набор команд для операций над двоично-десятичными числами.

		Двоично-десятичное представление			
Первое сл.	1889_{10}	0001	1000	1000	1001
Второе сл.	6376_{10}	0110	0011	0111	0110
Сумма	8265_{10}	0111	1011	1111	1111
Коррекция		0000	0110	0110	0110
				1	0101
			1	0110	
		1	0010		
		1000			
Откорректир. результат		8	2	6	5

Двоично-десятичные числа

Аппаратная реализация ***двоично-десятичного сумматора***:
A и B – двоично-десятичные операнды;
S – двоично-десятичный результат;
P – флаг переноса.



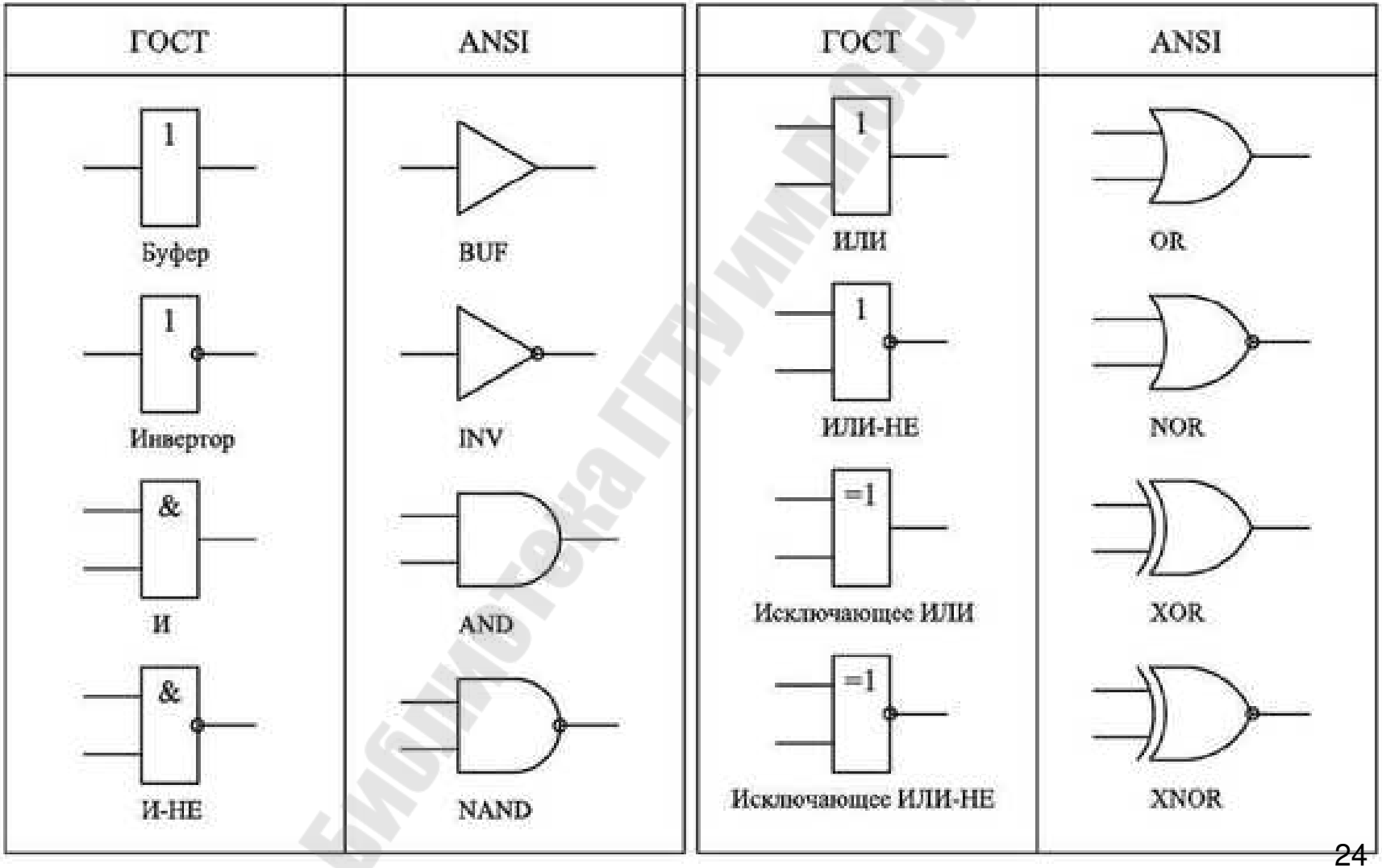
Сначала двоично-десятичные коды суммируются как двоичные. Если результатом суммирования является несуществующий двоично-десятичный код, его необходимо уменьшить на 10_{10}, и дополнительно сформировать сигнал переноса. Уменьшение кода на 10_{10} может выполняться его суммированием с дополнительным кодом числа 10_{10} (0110_2).

Тема 2

Логические и арифметические основы ЭВМ

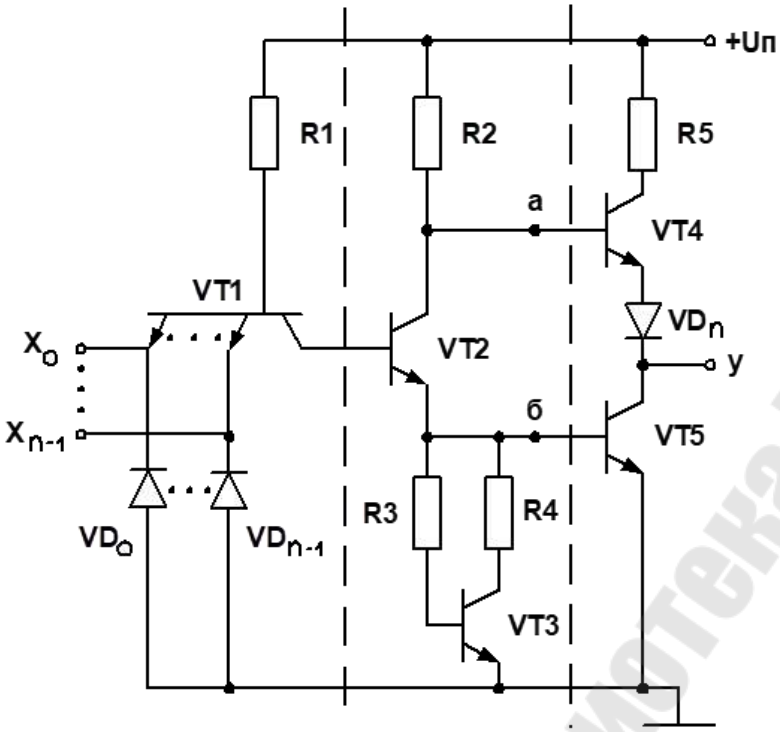
Логические основы ЭВМ

Условные графические обозначения логических элементов



Логические основы ЭВМ

Схема элемента И-НЕ



Таблицы истинности

И

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

ИЛИ

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

НЕ

A	X
0	1
1	0

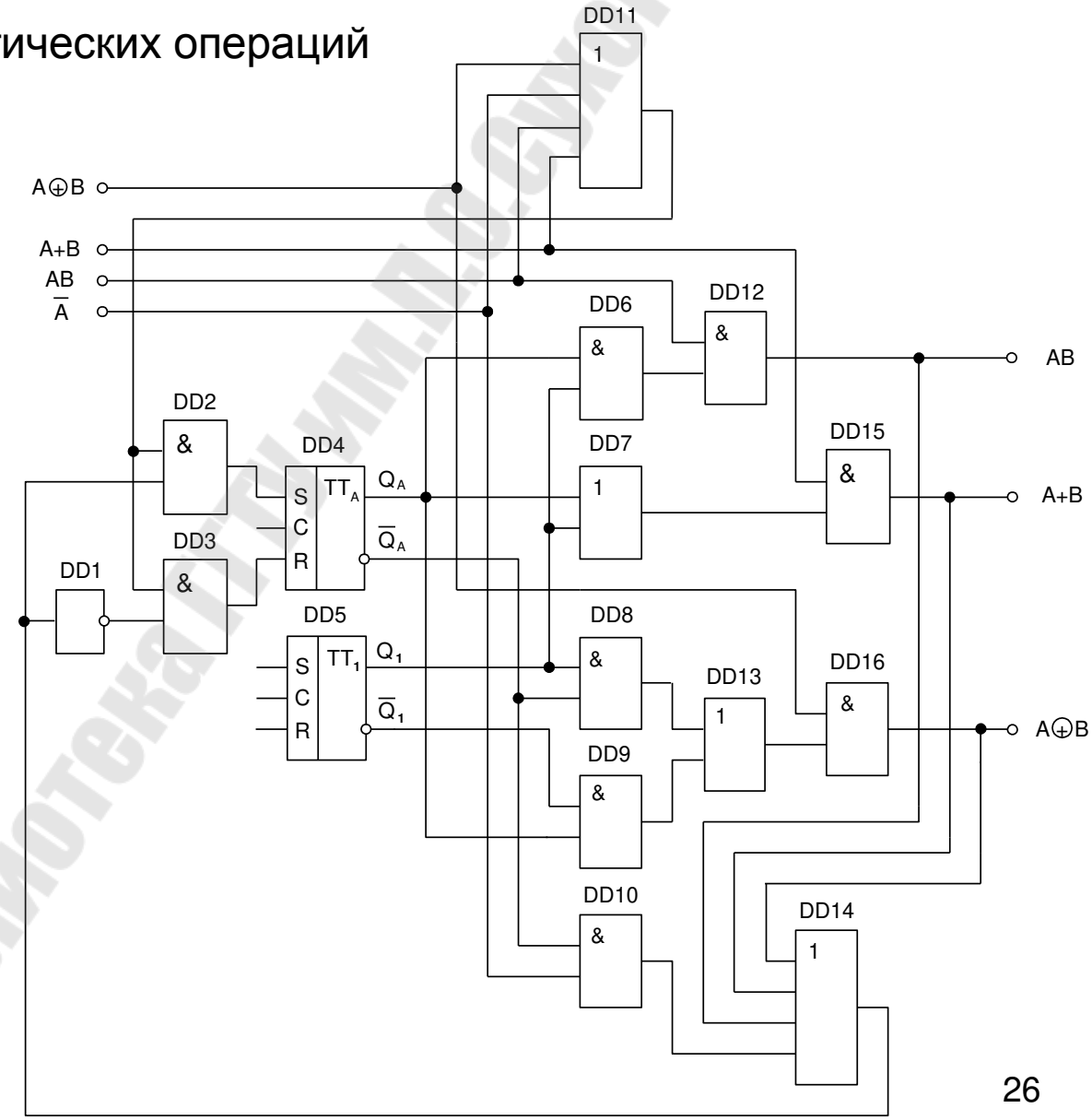
И - НЕ

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

ИЛИ - НЕ

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

Аппаратная реализация логических операций АЛУ



Логические команды Ассемблера

Команда AND выполняет поразрядно логическую операцию «И» над битами операндов *операнд_1* и *операнд_2*. Результат записывается на место *операнд_1*.

AND *операнд_1*, *операнд_2*

Варианты команды AND:

AND reg, reg	*AND mem, imm*	*AND reg, imm*
AND reg, mem	*AND mem, reg*	

где *reg* – регистр; *mem* – память; *imm* – непосредственное число

Пример сброса бит 7:4 (биты 3:0 не меняются):

```
mov al, 00111011b       ; AL = 0011 1011b
and al, 00001111b       ; AL = 0000 1011b
```

Команда AND всегда сбрасывает флаги переполнения (OF) и переноса (CF). Кроме того, она устанавливает значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата.

Логические команды Ассемблера

Команда OR выполняет поразрядно логическую операцию «ИЛИ» над битами операндов *операнд_1* и *операнд_2*. Результат записывается на место *операнд_1*.

OR *операнд_1*, *операнд_2*

Варианты команды OR:

OR reg, reg	*OR mem, imm*	*OR reg, imm*
OR reg, mem	*OR mem, reg*	

где *reg* – регистр; *mem* – память; *imm* – непосредственное число

Пример записи:

```
mov dl, 5          ; DL = 0000 0101b
or dl, 30h         ; DL = 0011 0101b
```

Команда AND всегда сбрасывает флаги переполнения (OF) и переноса (CF). Кроме того, она устанавливает значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата.

Логические команды Ассемблера

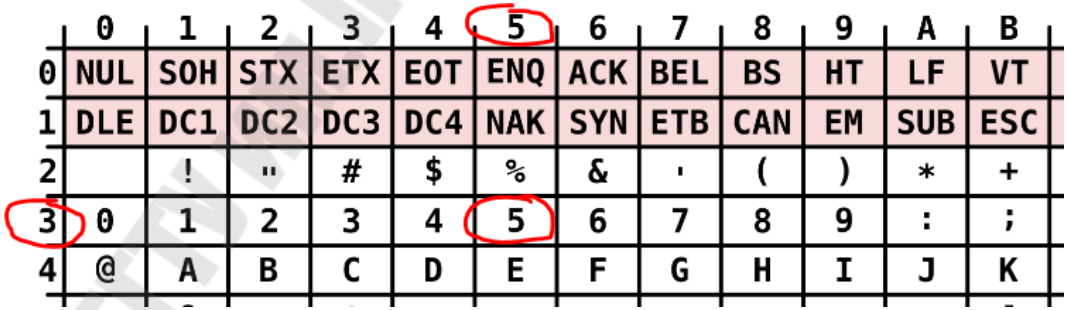
Применение команды OR:

- установка в единицу отдельных битов двоичного числа (например, флагов состояния процессора) по заданной маске;
- преобразование двоичного числа, в диапазоне от 0 до 9 в ASCII-код этого числа (путём добавления к коду числа значения 30h);

код символа ‘5’ = 35h

mov dl, 5 ; DL = 0000 0101b

or dl, 30h ; DL = 0011 0101b



	0	1	2	3	4	5	6	7	8	9	A	B
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC
2		!	"	#	$	%	&	'	(	)	*	+
3	0	1	2	3	4	5	6	7	8	9	:	;
4	@	A	B	C	D	E	F	G	H	I	J	K

- с помощью команды OR можно определить, какое значение находится в регистре (отрицательное, положительное или нуль)

or al, al

Флаг нуля (ZF)	Флаг знака (SF)	Значение числа
0	0	Больше нуля
1	0	Равно нулю
0	1	Меньше нуля

Логические команды Ассемблера

Команда XOR выполняет поразрядно логическую операцию «Исключающее ИЛИ» над битами операндов *операнд_1* и *операнд_2*. Результат записывается на место *операнд_1*.

XOR *операнд_1*, *операнд_2*

Варианты команды XOR:

XOR reg, reg	*XOR mem, imm*	*XOR reg, imm*
XOR reg, mem	*XOR mem, reg*	

где *reg* – регистр; *mem* – память; *imm* – непосредственное число

Пример записи:

```
mov dl, 5          ; DL = 0000 0101b
xor dl, dl         ; DL = 0000 0000b
```

Команда XOR всегда сбрасывает флаги переполнения (OF) и переноса (CF). Кроме того, она устанавливает значения флагов знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата.

Логические команды Ассемблера

Флаг четности (PF) позволяет узнать, какое количество единичных битов (четное или нечетное) содержится в младшем байте результата выполнения логической или арифметической команды.

Для этого сначала нужно выполнить команду XOR с нулевым значением, а затем проверить флаг четности:

```
mov al, 10110101b     ; число содержит нечетное (5) количество единиц
xor al, 0             ; поэтому флаг четности (PF) не устанавливается

mov al, 11001100b     ; число содержит четное (4) количество единиц
xor al, 0             ; поэтому флаг четности (PF) устанавливается
```

Для контроля четности 16-разрядных операндов, нужно выполнить команду XOR между старшим и младшим байтами этого числа:

```
mov ax, 64C1h         ;  AX = 0110 0100 1100 0001b
xor ah, al            ; флаг четности (PF) устанавливается
```

Логические команды Ассемблера

Команда **NOT** позволяет выполнить инверсию всех битов операнда, в результате чего получается ***обратный код числа***:

NOT *операнд_1*

Варианты команды NOT:

NOT reg
NOT mem

Например, ***обратный код*** числа F0h равен 0Fh:

```
mov al, 11110000b
not al                  ; AL = 00001111b
```

дополнительный код числа 0000 0005h равен FFFF FFFBh:

```
mov eax, 5              ; EAX = 5
not eax                 ; EAX = 1111 1111 1111 1111 … 11010b
add eax, 1              ; EAX = 1111 1111 1111 1111 … 11011b = -5
```

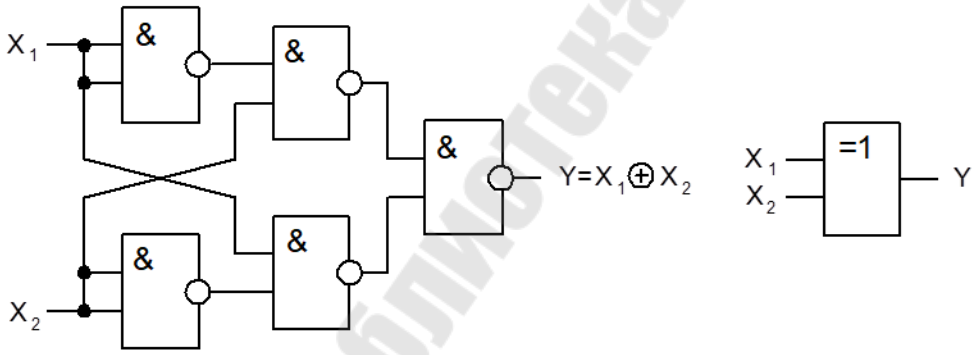
Команда NOT **не изменяет флаги** процессора.

Арифметическое сложение целых двоичных чисел

Результат сложения любых двух цифр находится на пересечении соответствующих строки и столбца. Если это пересечение находится в той части таблицы, которая расположена на тёмном фоне, то сложение данных цифр порождает перенос 1 в ближайший старший разряд.

+		Слагаемое	
		0	1
Слагаемое	0	0	1
	1	1	0

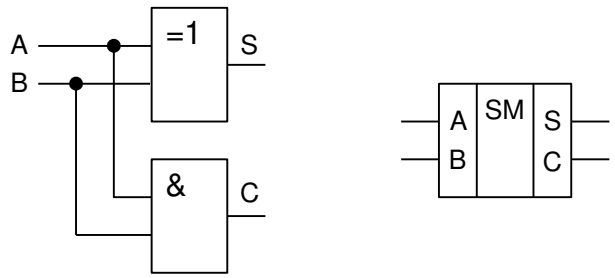
Аппаратная реализация схемы «Исключающее ИЛИ»



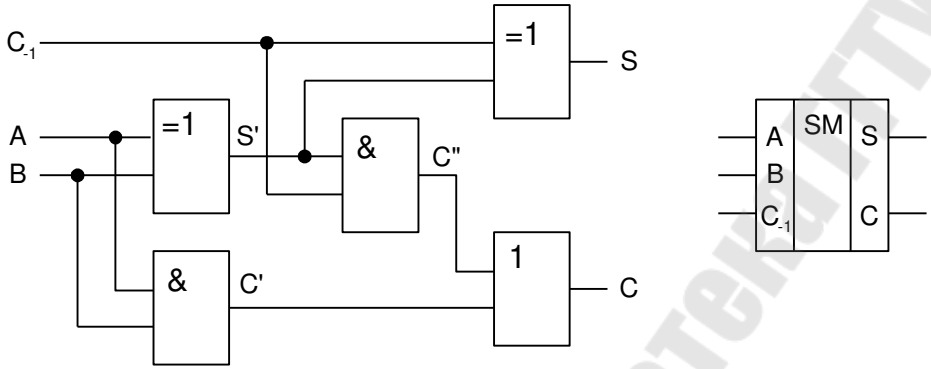
A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Арифметическое сложение целых двоичных чисел

Одноразрядный полусумматор



Одноразрядный сумматор

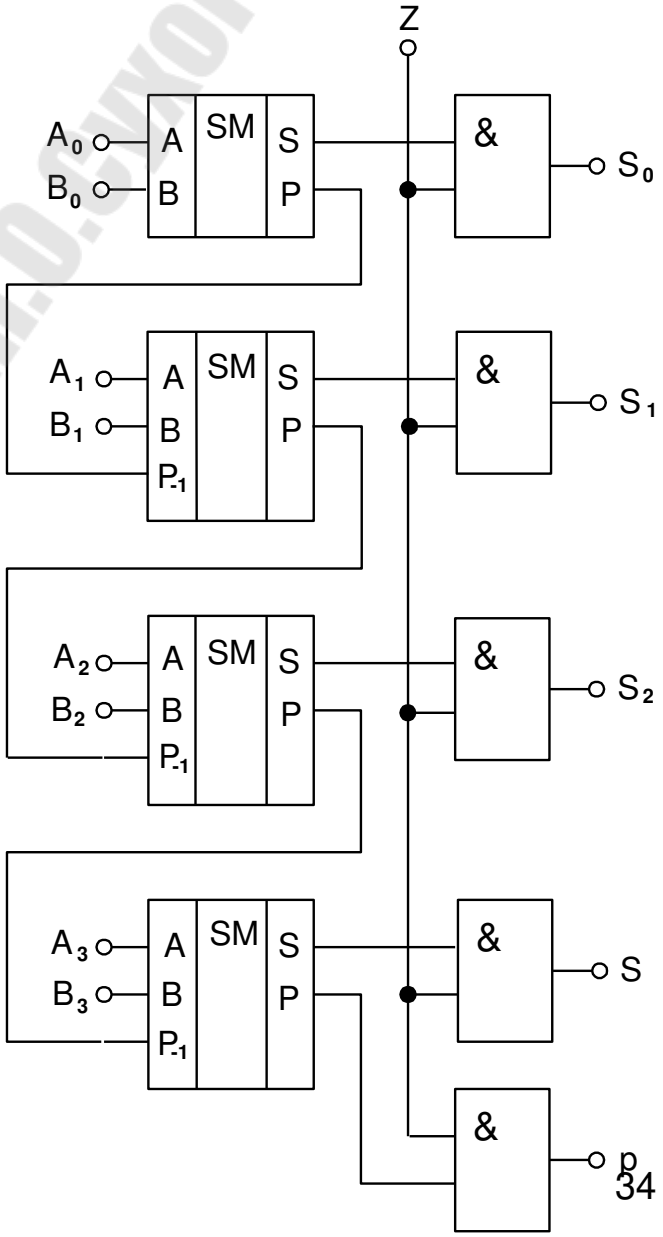


Аппаратная реализация ***двоичного 4-разрядного параллельного сумматора***:

A и B – двоичные операнды;

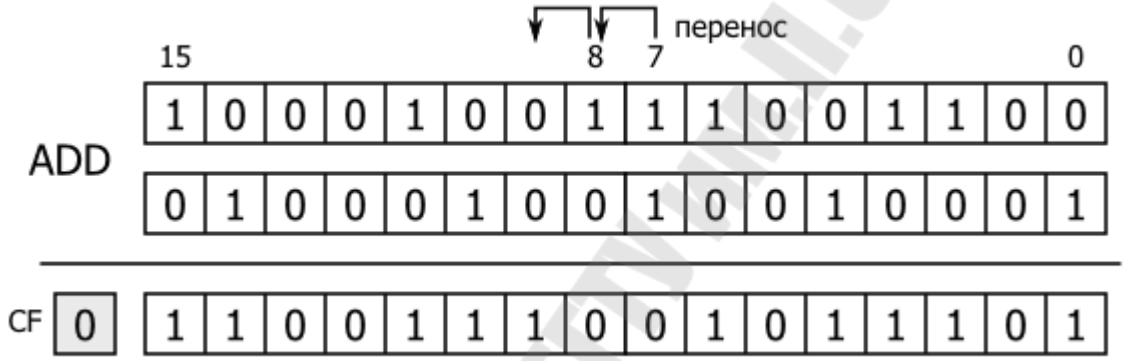
S – двоичный результат;

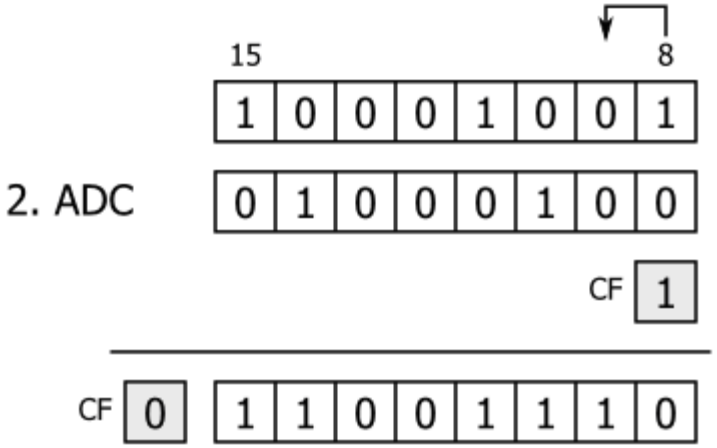
P – флаг переноса.



Арифметическое сложение целых двоичных чисел

В системе команд процессоров x86 имеются как команды обычного арифметического сложения ADD, так и специальные команды сложения с учётом флага переноса (CF) ADC.





Арифметические команды Ассемблера

Команда **ADD** выполняет арифметическое сложение двух операндов *операнд_1* и *операнд_2*. Операнды должны иметь одинаковый размер. Результат записывается на место *операнд_1*.

ADD *операнд_1*, *операнд_2*

операнд_1 = операнд_1 + операнд_2

Варианты команды ADD:

ADD reg, reg	*ADD reg, mem*	*ADD mem, imm*
ADD reg, imm	*ADD mem, reg*	

Пример выполнения:

```
a dd 45d          ; задание типа и значения переменной A
b dd -32d         ; задание типа и значения переменной B
e dd ?            ; задание типа переменной E
mov   eax, a      ; копирование A в регистр EAX
add    eax, b     ; сложение регистра EAX с переменной B
mov   e, eax      ; копирование суммы из EAX в E
```

Команда ADD изменяет флаги переполнения (OF), переноса (CF), знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата.

Арифметические команды Ассемблера

```
.data                       ; инициализированные данные
a       db 0FFh
b       db 8h
.data?                      ; неинициализированные данные
d       db ?

.code                       ; секция программы
start:                      ; метка начала программы
        xor eax, eax        ; EAX = 0
        xor ebx, ebx        ; EBX = 0
        mov al, a           ; записываем A в AL
        mov bl, b           ; записываем B в BL
        add al, bl          ; складываем AL в BL
        mov d, al           ; сохраняем AL в D
```

0-й шаг:

```
Registers (FPU)
EAX 0019FFCC
ECX 00401000 default_4.<ModuleEntryPoint>
EDX 00401000 default_4.<ModuleEntryPoint>
EBX 00308000
ESP 0019FF74
EBP 0019FF80
ESI 00401000 default_4.<ModuleEntryPoint>
EDI 00401000 default_4.<ModuleEntryPoint>

EIP 00401000 default_4.<ModuleEntryPoint>

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 30B000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

```
00401000  $  33C0          XOR EAX,EAX
00401002  ?  33DB          XOR EBX,EBX
00401004  ?  A0 00304000   MOV AL,BYTE PTR DS:[403000]
00401009  ?  8A1D 01304000 MOV BL,BYTE PTR DS:[403001]
0040100F  ?  02C3          ADD AL,BL
00401011  ?  A2 04304000   MOV BYTE PTR DS:[403004],AL
00401016  ?  6A 00         PUSH 0
00401018  ?  E8 01000000   CALL <JMP.&kernel32.ExitProcess>   Jump to KERNEL32.ExitProcess
0040101D  ?  CC            INT3
0040101E  ?  FF25 00204000 JMP DWORD PTR DS:[<&kernel32.ExitProces
00401024     00            DB 00
```

```
Address   Hex dump                                        ASCII (ANSI - кириллица)
00403000  FF 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00  я
00403010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Арифметические команды Ассемблера

```
00401000  $  33C0          XOR EAX,EAX
00401002  ?  33DB          XOR EBX,EBX
00401004  ?  A0 00304000   MOV AL,BYTE PTR DS:[403000]
00401009  ?  8A1D 0130400( MOV BL,BYTE PTR DS:[403001]
0040100F  ?  02C3          ADD AL,BL
00401011  ?  A2 04304000   MOV BYTE PTR DS:[403004],AL
00401016  ?  6A 00         PUSH 0
00401018  ?  E8 01000000   CALL <JMP.&kernel32.ExitProcess>      Jump to KERNEL32.E
0040101D  ?  CC            INT3
0040101E  ?  FF25 00204000 JMP DWORD PTR DS:[<&kernel32.ExitProces
00401024     00            DB 00
```

```
Registers (FPU)
EAX 00000000
ECX 00401000 default_4.<ModuleEntryPoint>
EDX 00401000 default_4.<ModuleEntryPoint>
EBX 003DC000
ESP 0019FF74
EBP 0019FF80
ESI 00401000 default_4.<ModuleEntryPoint>
EDI 00401000 default_4.<ModuleEntryPoint>

EIP 00401002 default_4.00401002

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 3DF000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

1-й шаг:

```
Registers (FPU)
EAX 00000000
ECX 00401000 default_4.<ModuleEntryPoint>
EDX 00401000 default_4.<ModuleEntryPoint>
EBX 00000000
ESP 0019FF74
EBP 0019FF80
ESI 00401000 default_4.<ModuleEntryPoint>
EDI 00401000 default_4.<ModuleEntryPoint>

EIP 00401004 default_4.00401004

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 3DF000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCE

EFL 00000246 (NO,NB,E,BE,NS,PE,GE
```

2-й шаг:

```
00401000  $  33C0          XOR EAX,EAX
00401002  ?  33DB          XOR EBX,EBX
00401004  ?  A0 00304000   MOV AL,BYTE PTR DS:[403000]
00401009  ?  8A1D 0130400( MOV BL,BYTE PTR DS:[403001]
0040100F  ?  02C3          ADD AL,BL
00401011  ?  A2 04304000   MOV BYTE PTR DS:[403004],AL
00401016  ?  6A 00         PUSH 0
00401018  ?  E8 01000000   CALL <JMP.&kernel32.ExitProcess>      Jump to KERNEL32.ExitProcess
0040101D  ?  CC            INT3
0040101E  ?  FF25 00204000 JMP DWORD PTR DS:[<&kernel32.ExitProces
00401024     00            DB 00
```

Арифметические команды Ассемблера

```
00401000  $  33C0          XOR EAX,EAX
00401002  ?  33DB          XOR EBX,EBX
00401004  ?  A0 00304000   MOV AL,BYTE PTR DS:[403000]
00401009  ?  8A1D 0130400( MOV BL,BYTE PTR DS:[403001]
0040100F  ?  02C3          ADD AL,BL
00401011  ?  A2 04304000   MOV BYTE PTR DS:[403004],AL
00401016  ?  6A 00         PUSH 0
00401018  ?  E8 01000000   CALL <JMP.&kernel32.ExitProcess>      Jump to KERNEL32.E
0040101D  ?  CC            INT3
0040101E  ?  FF25 0020400( JMP DWORD PTR DS:[<&kernel32.ExitProces
00401024     00            DB 00
```

```
Registers (FPU)
EAX 000000FF
ECX 00401000 default_4.<ModuleEntryPoint>
EDX 00401000 default_4.<ModuleEntryPoint>
EBX 00000000
ESP 0019FF74
EBP 0019FF80
ESI 00401000 default_4.<ModuleEntryPoint>
EDI 00401000 default_4.<ModuleEntryPoint>

EIP 00401009 default_4.00401009

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 3DF000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
```

3-й шаг:

```
Registers (FPU)
EAX 000000FF
ECX 00401000 default_4.<ModuleEntryPoint>
EDX 00401000 default_4.<ModuleEntryPoint>
EBX 00000008
ESP 0019FF74
EBP 0019FF80
ESI 00401000 default_4.<ModuleEntryPoint>
EDI 00401000 default_4.<ModuleEntryPoint>

EIP 0040100F default_4.0040100F

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 3DF000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE
```

4-й шаг:

```
00401000  $  33C0          XOR EAX,EAX
00401002  ?  33DB          XOR EBX,EBX
00401004  ?  A0 00304000   MOV AL,BYTE PTR DS:[403000]
00401009  ?  8A1D 0130400( MOV BL,BYTE PTR DS:[403001]
0040100F  ?  02C3          ADD AL,BL
00401011  ?  A2 04304000   MOV BYTE PTR DS:[403004],AL
00401016  ?  6A 00         PUSH 0
00401018  ?  E8 01000000   CALL <JMP.&kernel32.ExitProcess>      Jump to KERNEL32.ExitProcess
0040101D  ?  CC            INT3
0040101E  ?  FF25 0020400( JMP DWORD PTR DS:[<&kernel32.ExitProces
00401024     00            DB 00
```

Арифметические команды Ассемблера

```
00401000  $  33C0          XOR EAX,EAX
00401002  ?  33DB          XOR EBX,EBX
00401004  ?  A0 00304000   MOV AL,BYTE PTR DS:[403000]
00401009  ?  8A1D 0130400( MOV BL,BYTE PTR DS:[403001]
0040100F  ?  02C3          ADD AL,BL
00401011  ?  A2 04304000   MOV BYTE PTR DS:[403004],AL
00401016  ?  6A 00         PUSH 0
00401018  ?  E8 01000000   CALL <JMP.&kernel32.ExitProcess>        Jump to KERNEL32.Exi
0040101D  ?  CC            INT3
0040101E  ?  FF25 0020400( JMP DWORD PTR DS:[<&kernel32.ExitProces
00401024     00            DB 00
```

```
Registers (FPU)
EAX 00000007
ECX 00401000 default_4.<ModuleEntryPoint>
EDX 00401000 default_4.<ModuleEntryPoint>
EBX 00000008
ESP 0019FF74
EBP 0019FF80
ESI 00401000 default_4.<ModuleEntryPoint>
EDI 00401000 default_4.<ModuleEntryPoint>

EIP 00401011 default_4.00401011

C 1  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 3DF000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000213 (NO,B,NE,BE,NS,PO,GE,G)
```

5-й шаг:

6-й шаг:

```
Registers (FPU)
EAX 00000007
ECX 00401000 default_4.<ModuleEntryPoint>
EDX 00401000 default_4.<ModuleEntryPoint>
EBX 00000008
ESP 0019FF74
EBP 0019FF80
ESI 00401000 default_4.<ModuleEntryPoint>
EDI 00401000 default_4.<ModuleEntry

EIP 00401016 default_4.00401016

C 1  ES 002B 32bit 0(FFFFFFFF)
P 0  CS 0023 32bit 0(FFFFFFFF)
A 1  SS 002B 32bit 0(FFFFFFFF)
Z 0  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 3DF000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr 00000000 ERROR_SUCCESS

EFL 00000213 (NO,B,NE,BE,NS,PO,GE,G)
```

```
00401000  $  33C0          XOR EAX,EAX
00401002  ?  33DB          XOR EBX,EBX
00401004  ?  A0 00304000   MOV AL,BYTE PTR DS:[403000]
00401009  ?  8A1D 0130400( MOV BL,BYTE PTR DS:[403001]
0040100F  ?  02C3          ADD AL,BL
00401011  ?  A2 04304000   MOV BYTE PTR DS:[403004],AL
00401016  ?  6A 00         PUSH 0
00401018  ?  E8 01000000   CALL <JMP.&kernel32.ExitProcess>        Jump to KERNEL32.ExitProcess
0040101D  ?  CC            INT3
0040101E  ?  FF25 0020400( JMP DWORD PTR DS:[<&kernel32.ExitProces
00401024
```

```
Address   Hex dump                                        ASCII (ANSI - кириллица)
00403000  FF 08 00 00 07 00 00 00 00 00 00 00 00 00 00 00  a
00403010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Арифметические команды Ассемблера

Команда **ADC** (Add with Carry) производит целочисленное сложение двух знаковых или беззнаковых операндов и флага переноса. Первый операнд может быть переменной в регистре или в памяти (*r8, r16, r32, r/m8, r/m16, r/m32*). Второй операнд – непосредственным значением (*imm8, imm16, imm32*), переменной в регистре или в памяти. При этом оба операнда одновременно не могут быть переменными в памяти.

ADC *операнд_1*, *операнд_2*
операнд_1 = операнд_1 + операнд_2 + флаг CF

Команда **ADC** обычно используется в многобайтных или многословных (*multi-word*) операциях сложения:

```
xor edx, edx                ; EDX = 0
mov  eax, 0FFFF FFFFh       ; EAX = 4 294 967 295(10)
add  eax, 0FFFF FFFFh       ; EAX = EAX + 4 294 967 295(10)
adc  edx, 0                 ; EDX = EDX + CF  (учитываем перенос)
                            ; EDX:EAX = 0000 0001h:FFFF FFFEh
```

Команда ADC изменяет флаги переполнения (OF), переноса (CF), знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата.

Вычитание целых двоичных чисел

Из таблицы следует, что если уменьшаемое меньше вычитаемого, имеет место заём (borrow).

--		Вычитаемое	
		0	1
Уменьшаемое	0	0	1
	1	1	0

	1	0	0	1	1	1	1	0
-	0	1	1	0	0	1	0	1
	0	0	1	1	1	0	0	1

Отрицательные числа в прямом коде

Одним из способов представления двоичных чисел со знаком является **прямой код**. Для представления знака вводится дополнительный знаковый разряд: «0» это «+»; «1» это «–».

$$\mathbf{0}010\ 1011_2 = +\ 43_{10} \qquad \mathbf{1}010\ 1011_2 = -\ 43_{10}$$

Система представления чисел в прямом коде содержит одинаковое количество положительных и отрицательных чисел, причем ноль может быть представлен двумя способами, как +0 и –0 (оба значения совпадают).

Правила сложения двух чисел в прямом коде:

– если знаки чисел совпадают, то их величины складываются, а результату присваивается знак слагаемых;

– если знаки разные, то из большей величины вычитается меньшая, а результату присваивается знак большей величины.

Таким образом, необходимо производить проверку знаков слагаемых.

Отрицательные числа в дополнительном коде

В ЭВМ отрицательные числа обычно представляются в виде ***дополнений*** до основания системы счисления.

При двоичной системе это ***дополнение до двух*** такое, что ***сумма n-разрядного двоичного числа D и его дополнения равна 2^n.*** Поэтому для нахождения дополнения числа *D* нужно найти разность $2^n - D$.

Например, 8-битное двоичное число со знаком можно представить в виде семи (6…0) значащих разрядов и восьмого (7) знакового:

номера битов	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
	0	1	1	0	0	1	0	1
знак и веса битов	*+/-*	*64*	*32*	*16*	*8*	*4*	*2*	*1*

здесь $D = \mathbf{0}110\ 0101_2 = +101_{10}$.

Определим дополнение числа *D*. Для этого вычислим разность $2^8 - D$:

2^8	1	0	0	0	0	0	0	0	0
-									
D		0	1	1	0	0	1	0	1
		1	0	0	1	1	0	1	1

Непосредственно вычитать сложно, т.к. приходится делать заем из старшего разряда.

Отрицательные числа в дополнительном коде

Для упрощения вычислений преобразуем исходное выражение:

$$2^n - D = [(2^n - 1) - D] + 1.$$

Поскольку $2^8 = 1\ 0000\ 0000_2$, то число $2^n - 1 = 1111\ 1111_2$.

Из такого числа вычитать легче, т.к. ни в одном из разрядов, независимо от величины вычитаемого, заем делать не надо. Разность $(2^n - 1) - D$ представляет собой ***инвертированное*** значение числа *D*, т.е. ***обратный код*** числа *D*.

ПРАВИЛО*: для получения дополнения до двух некоторого числа D необходимо получить его обратный код, а затем к нему прибавить единицу младшего разряда.*

Достоинством представления отрицательных чисел в дополнительном коде является то, что ***при сложении чисел не нужно проверять их знак*** (как в случае использования прямого кода). Всегда выполняется двоичное сложение и, если результат не выходит за границы разрядной сетки представления числа, то результат всегда получается верным.

Вычитание двоичных чисел заменяется сложением уменьшаемого с дополнительным кодом вычитаемого.

Отрицательные числа в дополнительном коде

8-разрядное двоичное слово	Десятичный эквивалент	
	двоичного числа со знаком	двоичного числа без знаком
0000 0000	+0	0
0000 0001	+1	1
0000 0010	+2	2
0000 0011	+3	3
…	…	…
0111 1100	+124	124
0111 1101	+125	125
0111 1110	+126	126
0111 1111	+127	127
1000 0000	-128	128
1000 0001	-127	129
1000 0010	-126	130
1000 0011	-125	131
…	…	…
1111 1100	-4	252
1111 1101	-3	253
1111 1110	-2	254
1111 1111	-1	255

Арифметика чисел в дополнительном коде

Пример 1. Вычислить разность чисел 58_{10} и 23_{10}.

а) определим дополнительный код числа 23_{10}:

число 23_{10} в двоичной форме:	$0001\ 0111_2$;
обратный код числа 23_{10}:	$1110\ 1000_2$;
добавляемая к обратному коду 1_2:	$\underline{0000\ 0001}_2$;
число 23_{10} в дополнительном коде:	$1110\ 1001_2$.

б) вычислим разность (складываем уменьшаемое с дополнительным кодом вычитаемого):

$0011\ 1010_2$ - число 58_{10};

$\underline{1110\ 1001}_2$ - дополнительный код числа 23_{10};

$1\ 0010\ 0011_2$ - разность 35_{10} (единица переноса в старший разряд отбрасывается в случае положительного результата).

Арифметика чисел в дополнительном коде

Пример 2. Вычислить разность чисел 26_{10} и 34_{10}.

а) определим дополнительный код числа 34_{10}:

число 34_{10} в двоичной форме:	$0010\ 0010_2$;
обратный код числа 34_{10}:	$1101\ 1101_2$;
добавляемая к обратному коду 1_2:	$\underline{0000\ 0001}_2$;
число 34_{10} в дополнительном коде:	$1101\ 1110_2$.

б) вычислим разность (складываем уменьшаемое с дополнительным кодом вычитаемого):

$0001\ 1010_2$ - число 26_{10};
$\underline{1101\ 1110}_2$ - дополнительный код числа 34_{10};
$1111\ 1000_2$ - разность в дополнительном коде (поскольку в старшем разряде единица).

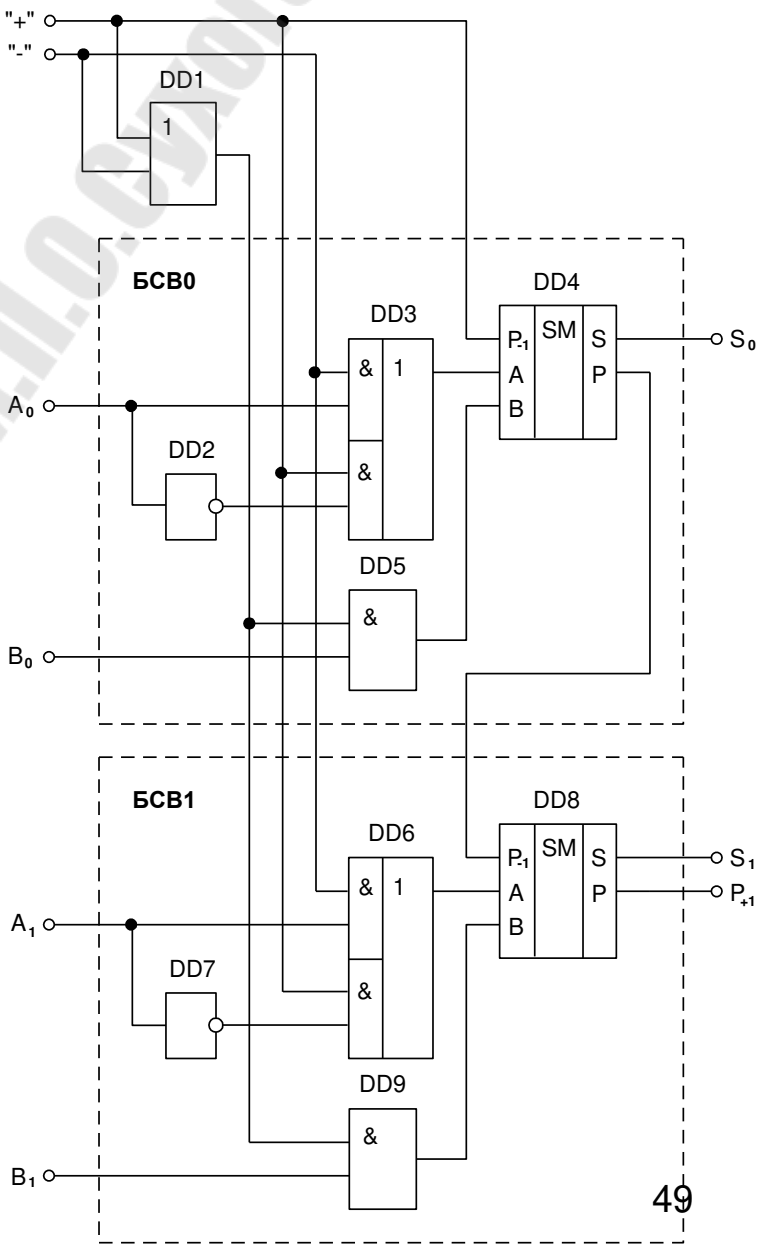
в) определим абсолютное значение (прямой код разности):

$1111\ 1000_2$ - разность в дополнительном коде;
$0000\ 0111_2$ - инверсия (обратный код) дополнительного кода;
$\underline{0000\ 0001}_2$ - добавляемая к обратному коду 1_2;
$0000\ 1000_2$ - абсолютное значение разности 8_{10}.

Сложение/вычитание целых двоичных чисел

Аппаратная реализация операций арифметического сложения/вычитания:
A и B – двоичные операнды;
S – двоичный результат;
P – флаг переноса;
+/– – вход выбора операции.

Схема позволяет получить на выходе сигнал либо суммы, либо разности двух двоичных кодов. Вид выполняемой операции определяется значением управляющего сигнала.



Арифметические команды Ассемблера

Команда **SUB** (subtract) выполняет арифметическое вычитание двух операндов *операнд_1* и *операнд_2*. Операнды должны иметь одинаковый размер. Результат записывается на место *операнд_1*.

SUB *операнд_1*, *операнд_2*

операнд_1 = операнд_1 - операнд_2

Варианты команды SUB:

SUB reg, reg	*SUB reg, mem*	*SUB mem, imm*
SUB reg, imm	*SUB mem, reg*	

Пример выполнения:

```
mov  eax, operand1     ; первое слагаемое помещаем в EAX
sub  eax, operand2     ; производим вычитание двух операндов в
                       ; дополнительном коде
jns  m1                ; переход, если результат не отрицательный
not  eax               ; инверсия EAX
inc  eax               ; увеличиваем EAX на 1
m1:                    ; EAX - результат вычитания в прямом коде
```

Команда ADD изменяет флаги переполнения (OF), переноса (CF), знака (SF), нуля (ZF) и четности (PF) в соответствии со значением результата.

Арифметическое умножение целых двоичных чисел

В результате умножения двоичных чисел перенос не возникает никогда.

x		Множимое	
		0	*1*
Множитель	*0*	0	0
	1	0	1

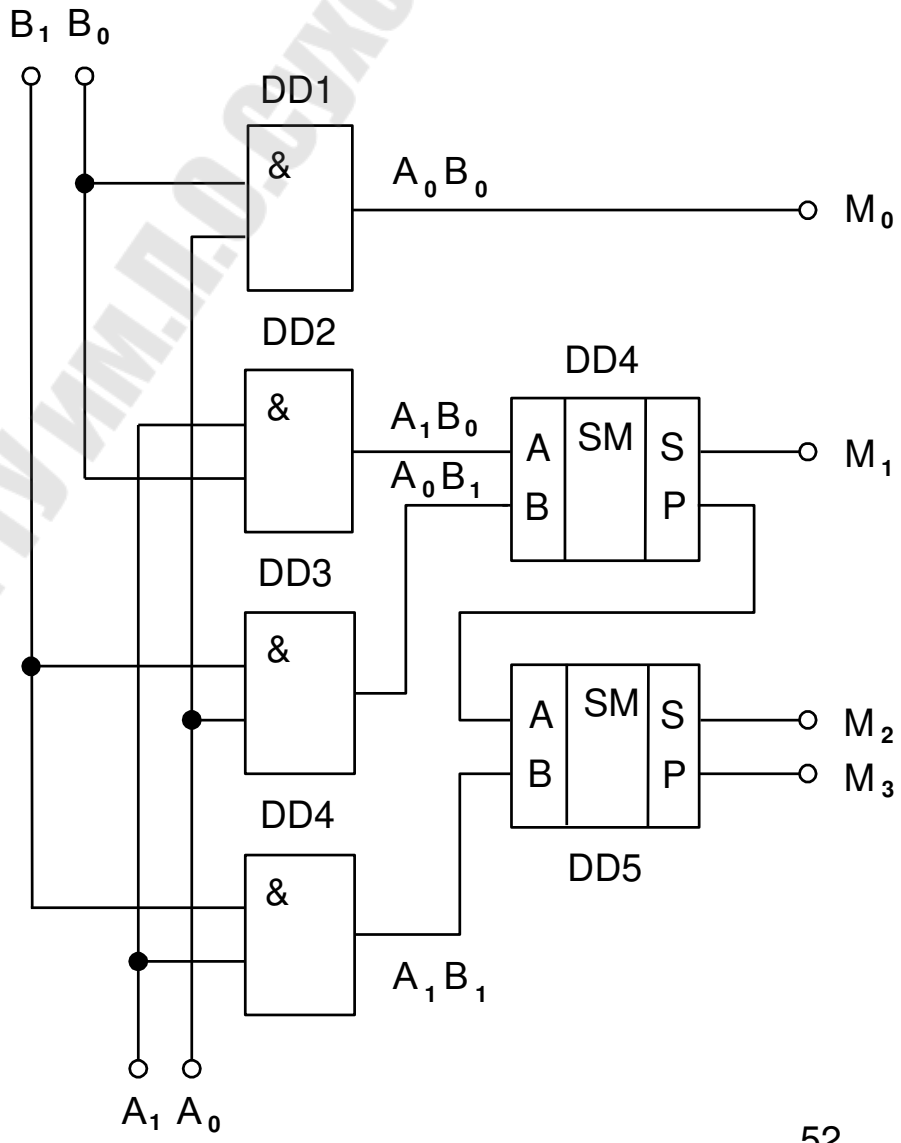
Воспользуемся таблицей двоичного умножения для вычисления произведения 8-разрядных двоичных эквивалентов десятичных чисел 17_{10} и 12_{10}:

0001 0001 - множимое 17_{10};
0000 1100 - множитель 12_{10};
0000 0000 - первое частичное произведение;
0 0000 000 - второе частичное произведение;
00 0100 01 - третье частичное произведение;
000 1000 1 - четвертое частичное произведение;
0000 0000 - пятое частичное произведение;
0 0000 000 - шестое частичное произведение;
00 0000 00 - седьмое частичное произведение;
000 0000 0 - восьмое частичное произведение;
0000 0000 1100 1100 - результат (204_{10}).

Арифметическое умножение целых двоичных чисел

Аппаратная реализация ***арифметического умножения*** двоичных целых чисел без знака:
A и B – 2-разрядные сомножители;
M – 4-разрядное произведение.

```
                    A1A0
                    B1B0
              A1B0  A0B0
A1B1          A0B1
A1B1  A1B0+A0B1 A0B0
```



Арифметическое умножение целых двоичных чисел

Алгоритм двоичного умножения, именуемый умножением путем сдвига и сложения:

1) ***формирование частичного произведения***. Если значение текущего разряда множителя равно 0, то и результат равен 0, если значение этого разряда равно 1, то результат - является копией множимого.
2) ***формирование сдвига***. Каждое очередное частичное произведение, начиная со второго, имеет сдвиг на один разряд (позицию) влево по отношению к множимому.
3) ***формирование суммы***. Каждый раз, когда значение разряда множителя равно 1, к результату необходимо прибавить множимое, расположенное в позиции, определенной правилом сдвига.
4) ***формирование результирующего произведения***. Искомое произведение есть результат выполнения всех операций сдвига и сложения.

Арифметическое умножение целых двоичных чисел

В процессорах x86 для умножения чисел без знака предназначена команда **MUL**. У этой команды только один операнд — второй множитель, который должен находиться в регистре или в памяти. Местоположение первого множителя и результата задаётся неявно и зависит от размера операнда:

Размер операнда	1-й множитель	Результат
1 байт	AL	AX
2 байта	AX	DX:AX
4 байта	EAX	EDX:EAX

Запись «DX: AX» означает, что старшее слово результата будет находиться в DX, а младшее — в AX.

Пример:

```
mul bl    ; AX = AL * BL
mul ax    ; DX:AX = AX * AX
```

Если старшая часть результата равна нулю, то флаги CF и OF будут иметь нулевое значение. В этом случае старшую часть результата можно отбросить. Это свойство можно использовать в программе, если результат должен быть такого же размера, как множители.

Арифметическое умножение целых двоичных чисел

Команда **IMUL** умножает целые числа со знаком и может использовать один, два или три операнда:

- один операнд, аналогично MUL;
- два операнда

IMUL операнд_1, операнд_2
*операнд_1 = операнд_1 * операнд_2*

варианты: *IMUL reg, reg* *IMUL reg, mem* *IMUL reg, imm;*

```
imul edx, ecx        ; EDX = EDX * ECX
imul ebx, A          ; EBX = EBX * A
imul ecx, 6          ; ECX = ECX * 6
```

- три операнда

IMUL операнд_1, операнд_2, операнд_3
*операнд_1 = операнд_2 * операнд_3*

варианты: *IMUL reg, reg, imm* *IMUL reg, mem, imm* *IMUL reg, imm, imm*

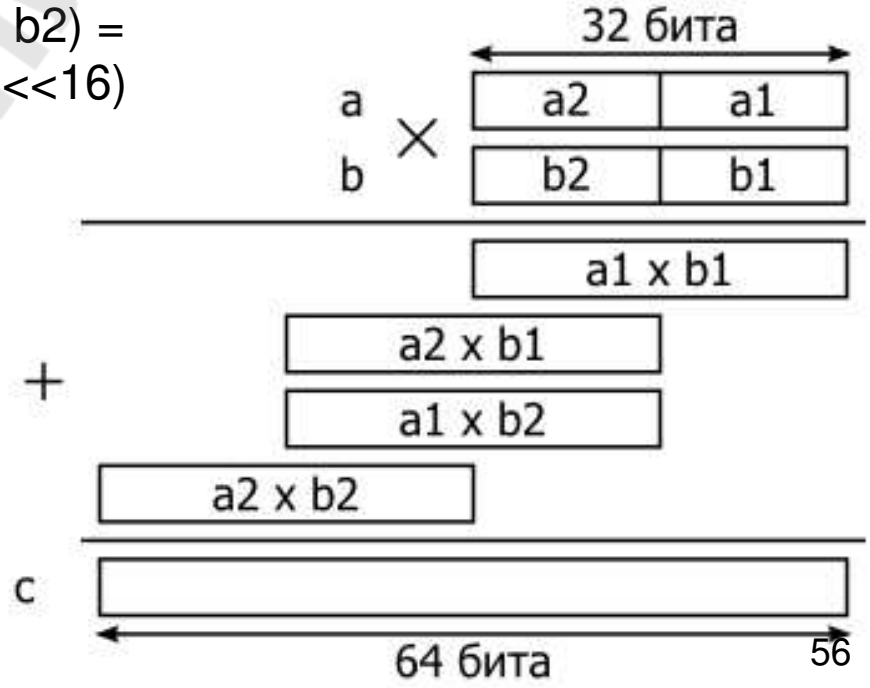
```
imul ebx, A, 9       ; EBX = A * 9
imul ecx, edx, 11    ; ECX = EDX * 11
```

Арифметическое умножение целых двоичных чисел

Команда MUL в 16-битном режиме позволяет умножать максимум 16-битные значения, поэтому для умножения 32-битных чисел необходим специальный алгоритм.

Умножать большие числа приходится по частям, а затем складывать эти промежуточные результаты. Допустим, нам надо умножить два 32-битных числа a и b. В результате должно получиться 64-битное число c. Обозначим как a1 — младшее слово a, a2 — старшее слово a, b1 — младшее слово b, b2 — старшее слово b.

c = a x b = ((a2<<16) + a1) x ((b2<<16) + b2) =
= (a1 x b1) + ((a2 x b1)<<16) + ((a1 x b2)<<16)
+ ((a2 x b2)<<32)



Деление целых двоичных чисел

В процессорах x86 для деления чисел без знака предназначена команда **DIV**. У этой команды только один операнд — делитель, который должен находиться в регистре или в памяти. Местоположение делимого и результата задаётся неявно и зависит от размера операнда:

Размер операнда	Делимое	Частное	Остаток
1 байт	AX	AL	AH
2 байта	DX:AX	AX	DX
4 байта	EDX:EAX	EAX	EDX

Запись «DX: AX» означает, что старшее слово результата будет находиться в DX, а младшее — в AX.

Пример:

```
div bl     ; AL = AX / BL
div bx     ; AX = DX:AX / BX
```

Команда **IDIV** используется для деления чисел со знаком, синтаксис ее такой же, как у команды DIV.

Деление целых двоичных чисел

Пример. Вычислить радиус по диаметру, значение которого сохранено в 16-битной переменной *diameter*, результат записать в переменную *radius*, а остаток проигнорировать.

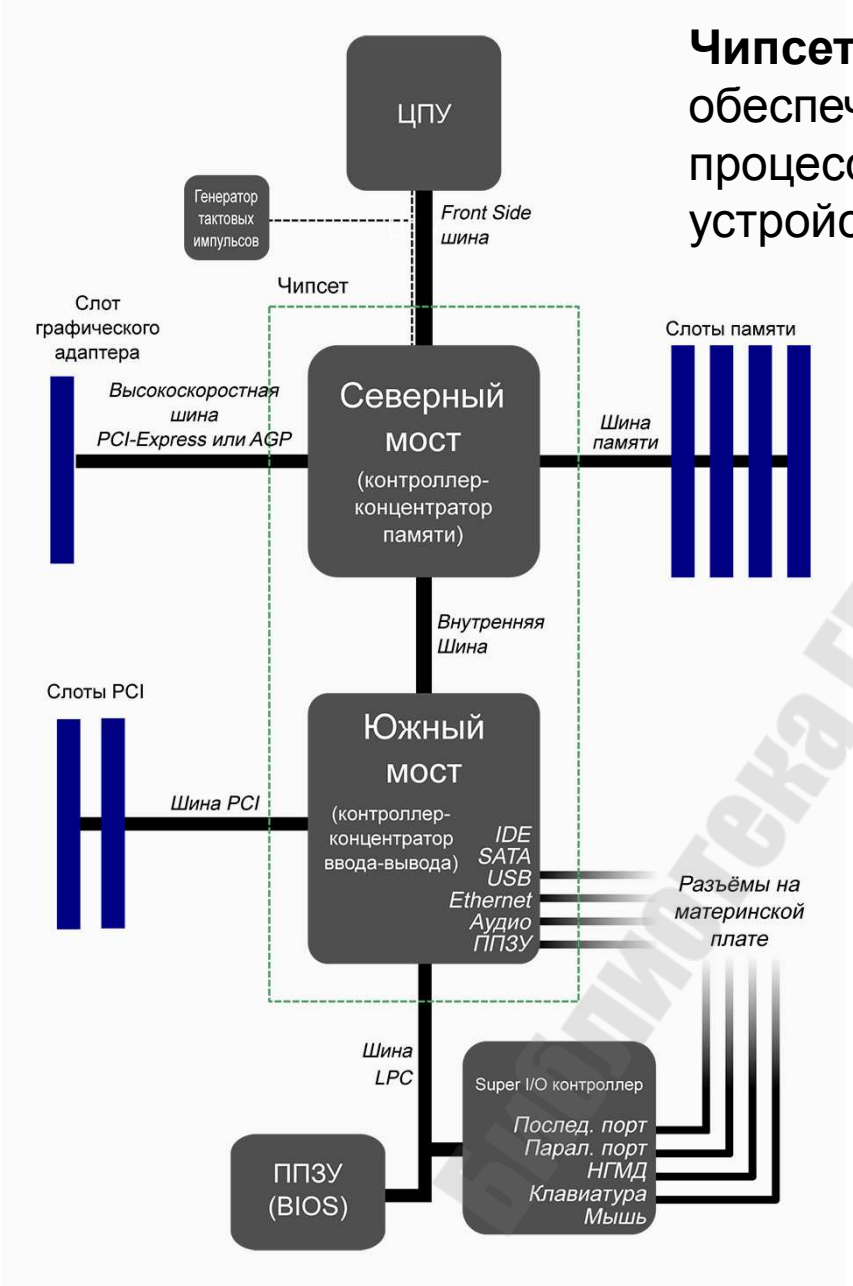
```
.data
diameter        dw  100

.data?
radius      db  ?

.code
start:
mov ax, diameter            ; AX = diameter
mov bl, 2                   ; загружаем делитель 2
div bl                      ; делим
mov radius, al              ; сохраняем результат
…
end start
```

Тема 3
Архитектура центрального процессора

Архитектура центрального процессора



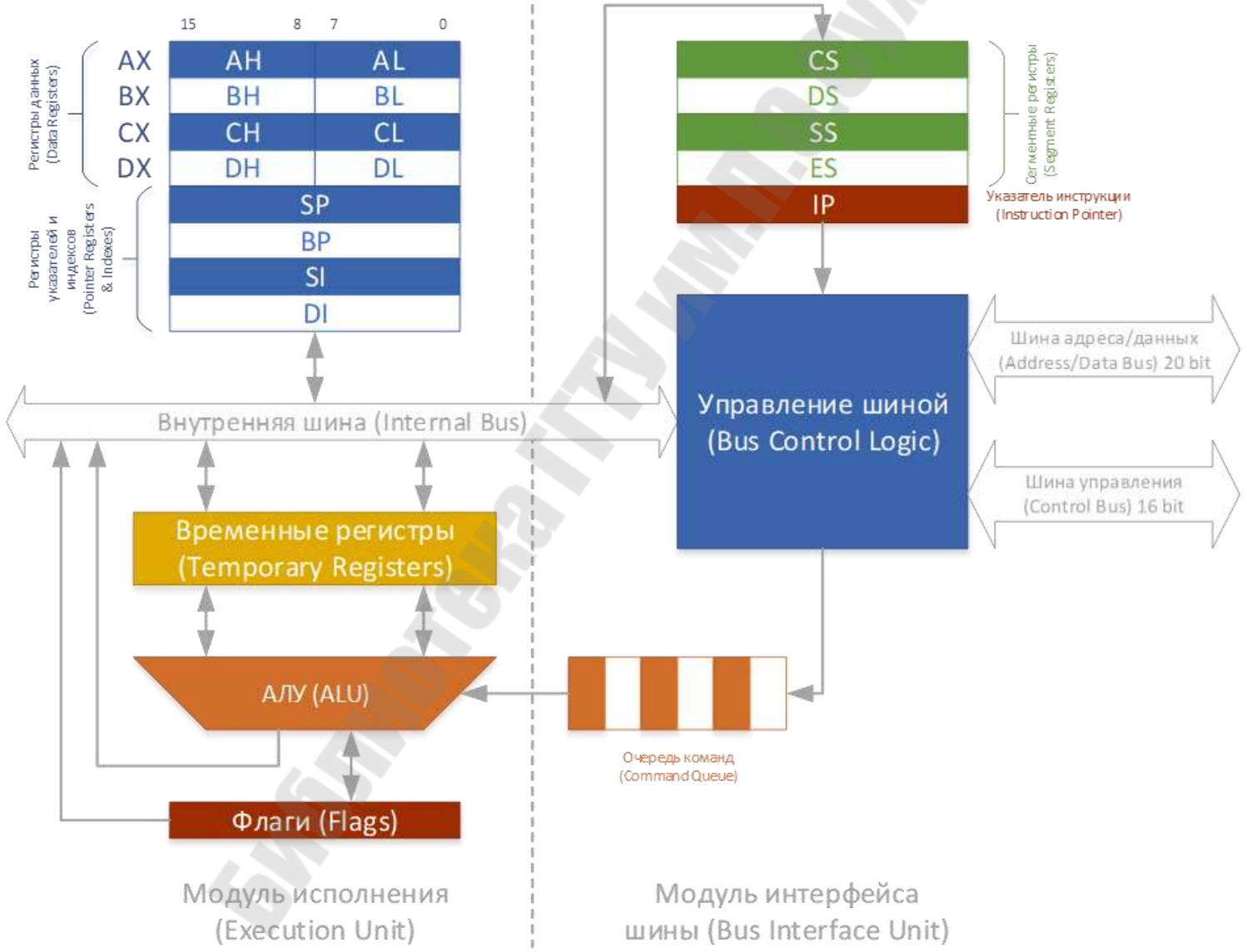
Чипсет (*Chipset*) - это набор микросхем, которые обеспечивают связь и взаимодействие между процессором, памятью, периферийными устройствами и другими компонентами системы.

Северный мост (*Northbridge*) определяет частоту системной шины *Front Side Bus* (*FSB*), осуществляет обмен между центральным процессором и:
- оперативной памятью (ОЗУ);
- видеоадаптером;
- южным мостом.

Южный мост *(Southbridge)* управляет:
- периферийными устройствами: HDD, SSD, DVD, USB, аудиоадаптеры, внешние интерфейсы и другие;
- вводом-выводом (I/O): клавиатура, мышь, Ethernet;
- управление энергопотреблением;
- BIOS.

Архитектура центрального процессора

Архитектура процессора i8086

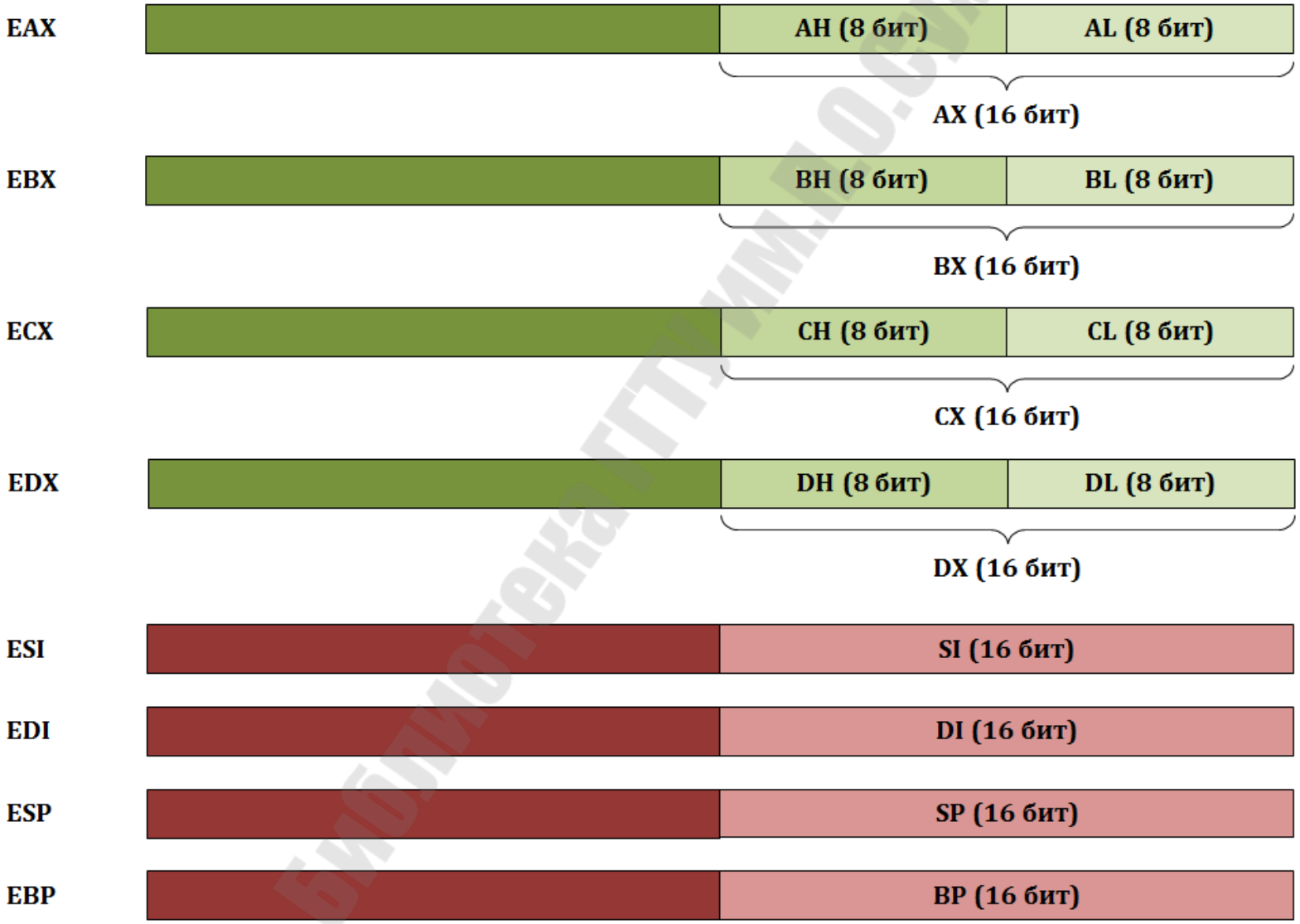


Архитектура центрального процессора

Иерархия запоминающих устройств

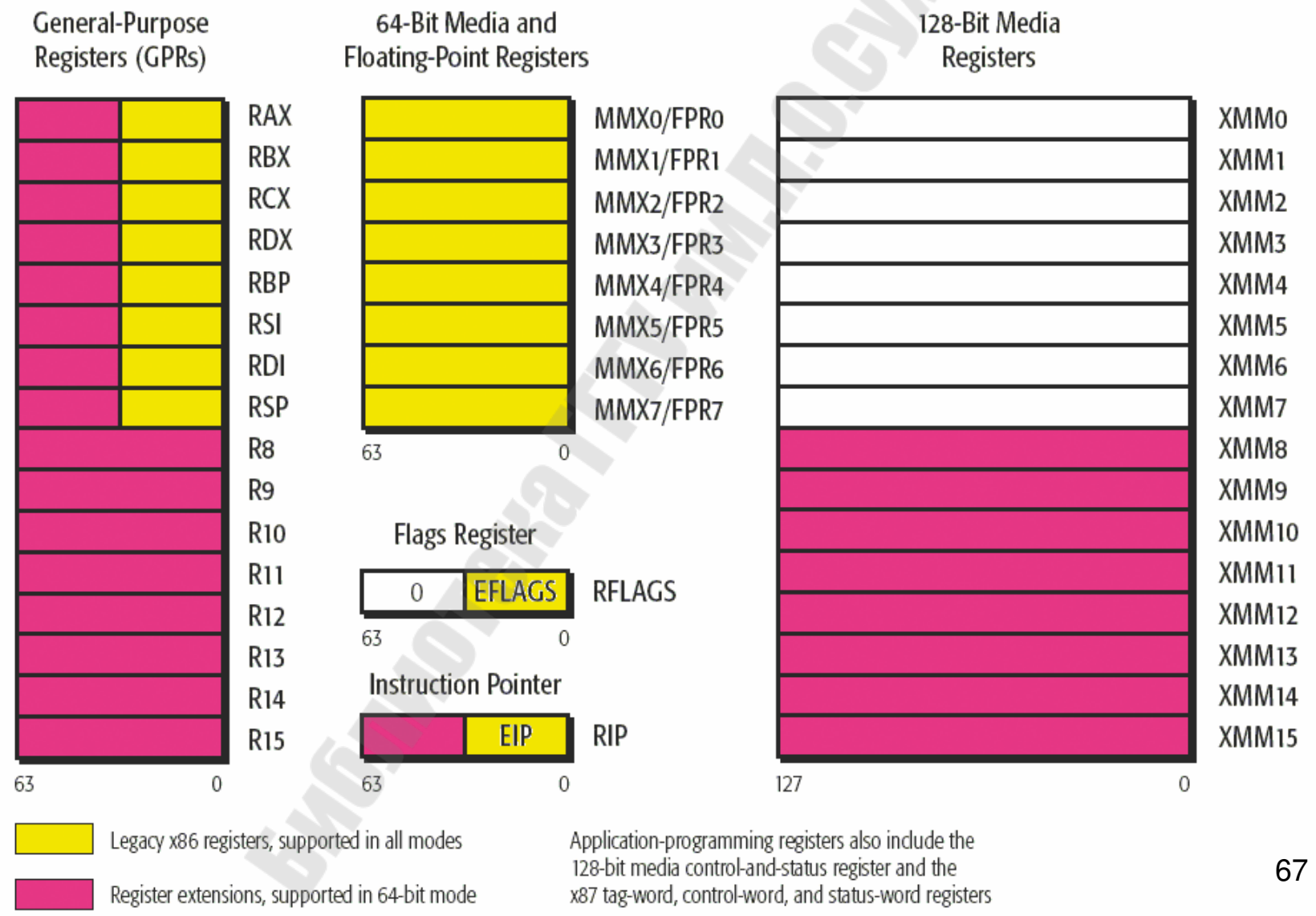
Архитектура центрального процессора

Регистры микропроцессора i386 архитектуры x86-32



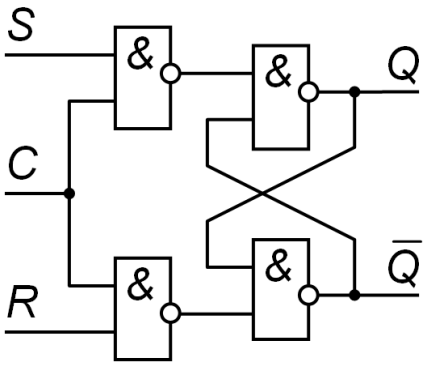
Архитектура центрального процессора

Регистры процессора архитектуры x86-64 (AMD64, Intel64)

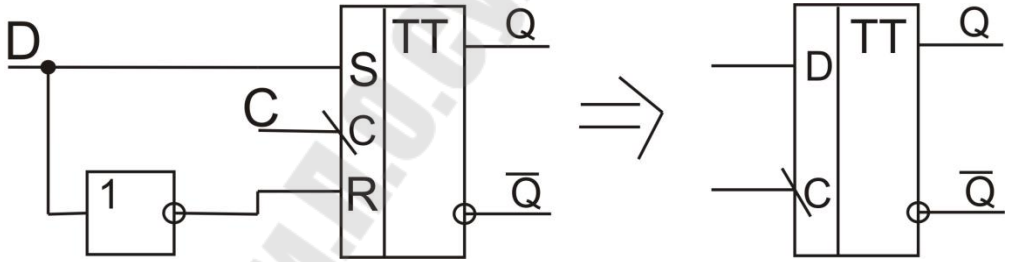


Архитектура центрального процессора

Синхронный RS-триггер



D-триггер на базе RS-триггера



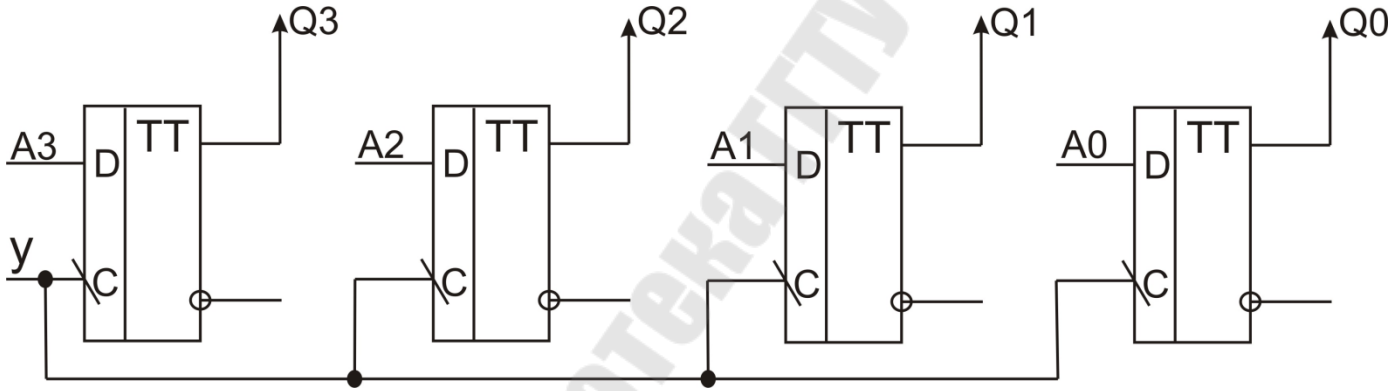
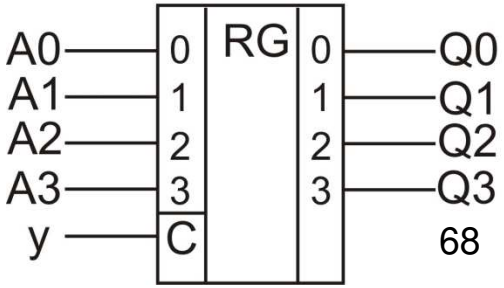


Схема 4-разрядного параллельного регистра на D-триггерах



УГО 4-разрядного параллельного регистра

Архитектура центрального процессора

Регистры x86 – это специальные ячейки памяти, расположенные непосредственно в процессоре. Работа с регистрами выполняется намного быстрее, чем с ячейками оперативной памяти, поэтому регистры активно используются как в программах на языке ассемблера, так и компиляторами языков высокого уровня.

РОН (General Purpose Registers):

- RAX/EAX/AX/AH/AL (*accumulator register*) – аккумулятор – хранение результатов арифметических операций, возвращаемых значений функций и других общих операций;
- RBX/EBX/BX/BH/BL (*base register*) – регистр базы – используется в качестве базового указателя для адресации данных, массивов и структур;
- RCX/ECX/CX/CH/CL (*counter register*) – счётчик – используется в качестве счетчика в циклах и других операциях счета;
- RDX/EDX/DX/DH/DL (*data register*) – регистр данных – для операций ввода-вывода и других операций, где требуется временное хранение данных;
- RSI/ESI/SI (*source index register*) – индекс источника;
- RDI/EDI/DI (*destination index register*) – индекс приёмника;
- RSP/ESP/SP (*stack pointer register*) – регистр указателя стека;
- RBPEBP/BP (*base pointer register*) – регистр указателя базы кадра стека.

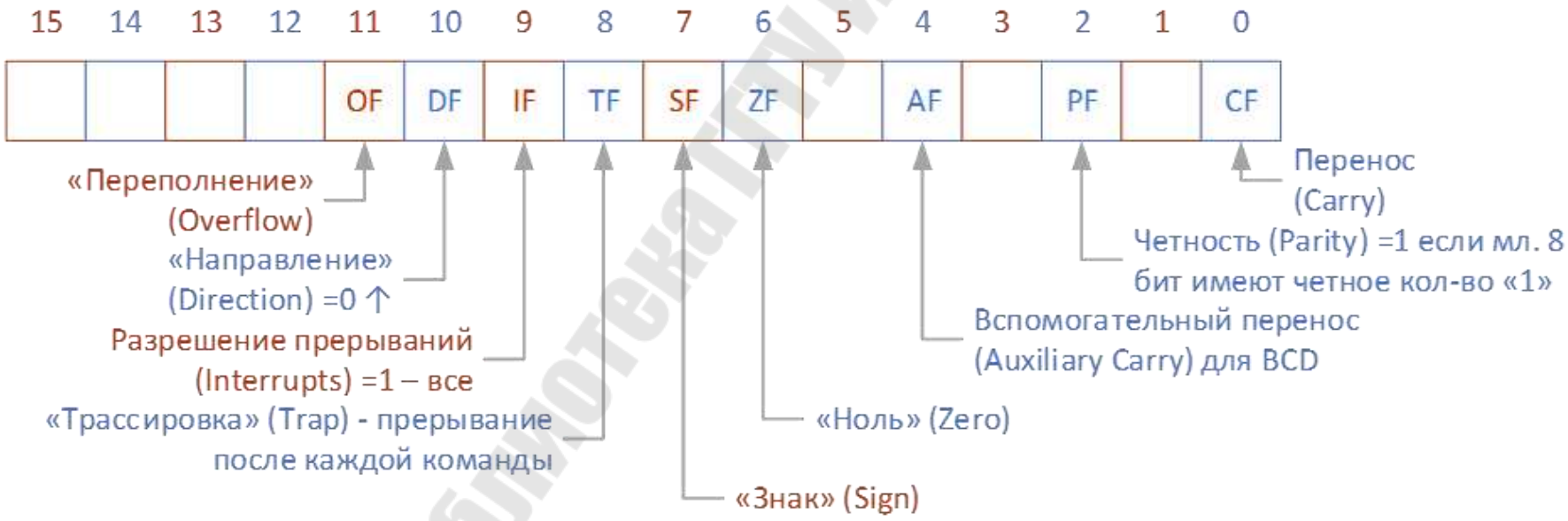
Регистры архитектуры x86

- MMX (*Multimedia Extensions*) - это регистры для поддержки инструкций SSE (Streaming SIMD Extensions), предназначенных для обработки мультимедийных данных. Каждый из 8 *64*-битных регистров MMX может содержать 64-битный целочисленный операнд или операнд с плавающей точкой;
- XMM (*eXtended Multimedia Registers*) - это 128-битные регистры, используемые в архитектуре *x86* для выполнения операций SIMD (Single Instruction, Multiple Data), например инструкций SSE. Эти регистры могут хранить данные в формате с плавающей точкой одинарной точности (32 бита), двойной точности (64 бита) или целых чисел (8, 16, 32 или 64 бита).

Архитектура центрального процессора

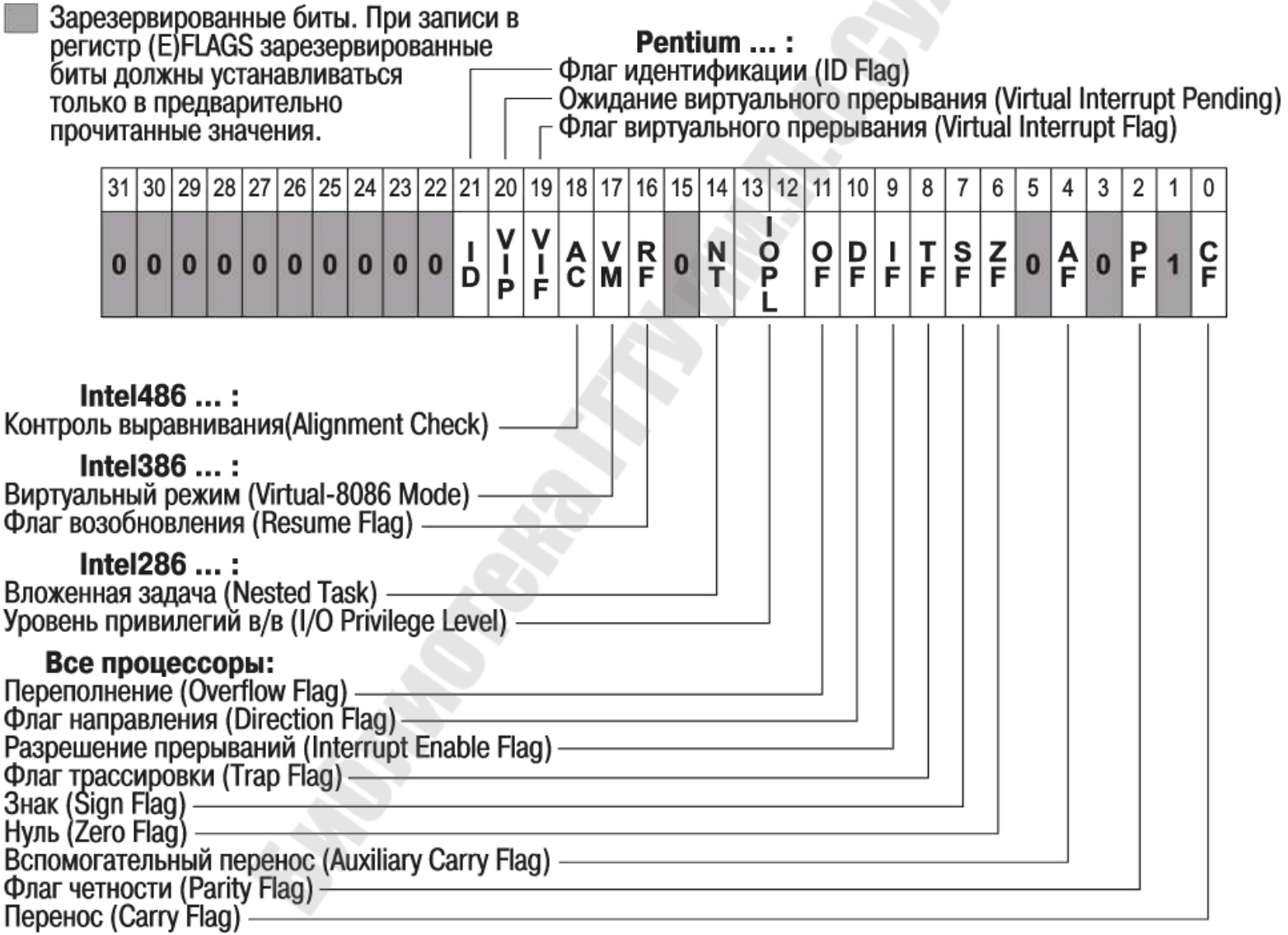
Регистр флагов (*Flags Register*): содержит флаги или биты состояния процессора. Используется для отслеживания результатов предыдущих операций и для принятия решений ветвления и выполнения условных инструкций. Флаги могут указывать на результаты операций сравнения, переполнения, знака и других условий.

Регистр флагов (FLAGS) процессора i8086



Архитектура центрального процессора

Регистр флагов (EFLAGS) 32-разрядного процессора

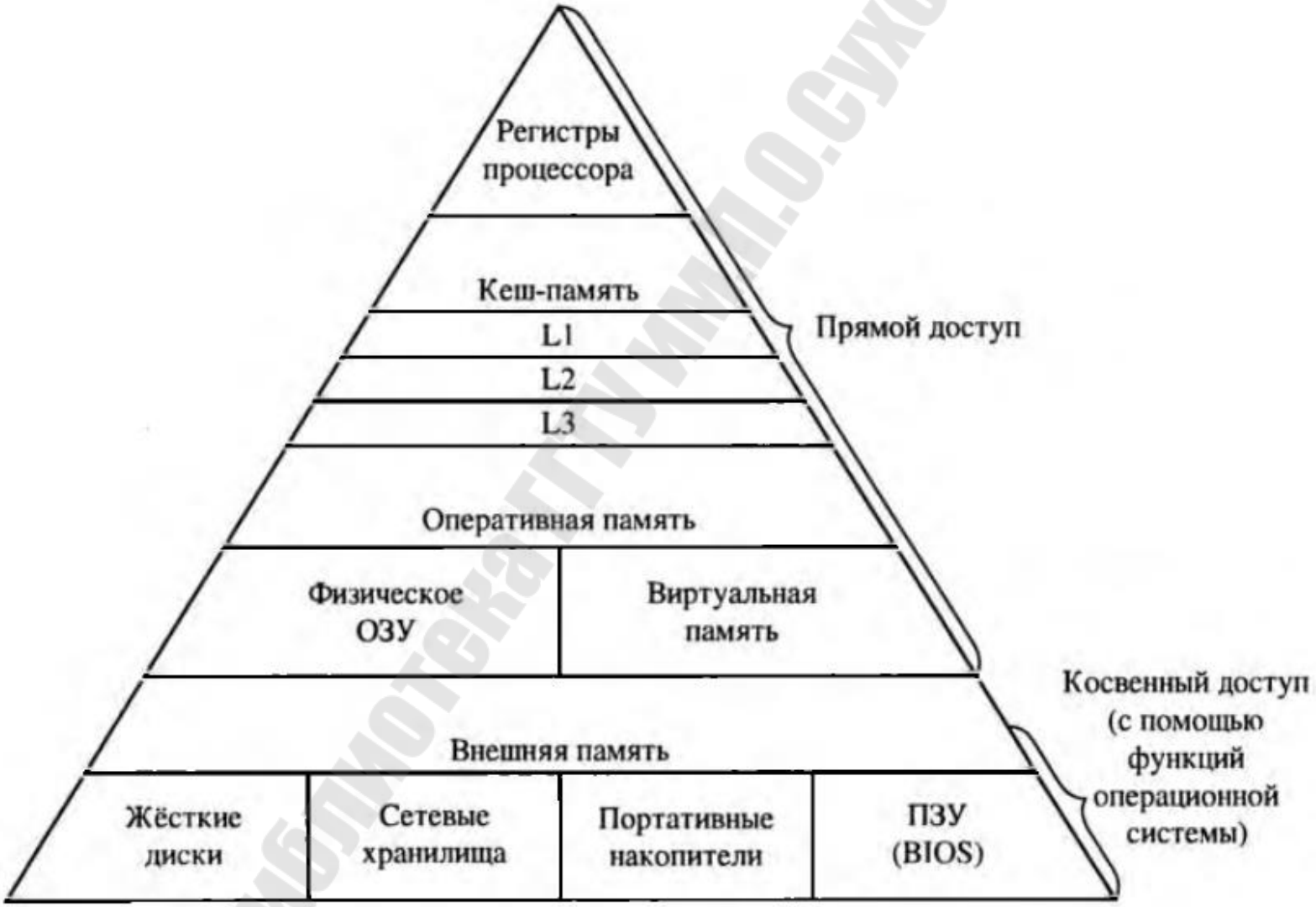


Архитектура центрального процессора

- **флаг переноса CF** (Carry Flag) устанавливается при переносе из старшего значащего бита (заёме в старший значащий бит) и *показывает наличие переполнения в беззнаковой целочисленной арифметике*. Также используется в длинной арифметике;
- **флаг переполнения OF** (Overflow Flag) сигнализирует о потере старшего бита результата в связи с переполнением разрядной сетки *при работе со знаковыми числами*;
- **флаг вспомогательного переноса AF** (Auxiliary Carry Flag) устанавливается при переносе из бита 3-го результата (заёме в 3-й бит) результата. Этот флаг ориентирован на использование в двоично-десятичной (binary coded decimal, BCD) арифметике;
- **флаг чётности PF** (Parity Flag) фиксирует *наличие четного числа единиц в младшем байте* результата операции, может быть использован, например, для контроля правильности передачи данных;
- **флаг нуля ZF** (Zero Flag) устанавливается, если результат равен нулю;
- **флаг знака SF** (Sign Flag) равен значению старшего значащего бита результата, который является знаковым битом в знаковой арифметике;

Тема 4
Организация памяти ЭВМ

Организация памяти ЭВМ



Иерархия памяти ЭВМ

Организация памяти ЭВМ

Уровни кэш-памяти

L1: встроен в каждое ядро процессора.

Разделяется на кэш для данных (*L1d*) и кэш для инструкций (*L1i*).

Обладает наименьшей задержкой и наибольшей скоростью, но имеет ограниченный объем.

L2: встроен в каждое ядро, но может быть реализован как отдельный кэш для каждого ядра или как общий кэш для нескольких ядер.

Имеет бОльшую емкость по сравнению с *L1*, но чуть большую задержку, чем *L1*.

L3: находится на кристалле процессора, но не привязан к конкретному ядру.

Имеет еще большую емкость, но более высокую задержку по сравнению с *L1* и *L2*.

Используется для кэширования данных, доступных нескольким ядрам процессора.

Организация памяти ЭВМ

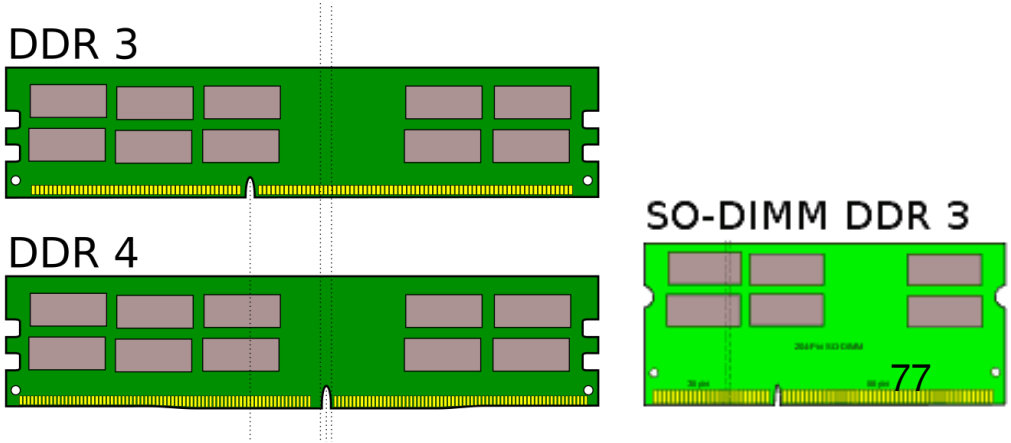
SDRAM (*synchronous dynamic random access memory*) – синхронное динамическое оперативное запоминающее устройство.

SDR (*single data rate*) – одинарная скоростью передачи данных – одна передача данных за период тактового сигнала.

DDR (*double data rate*) – удвоенная скорость передачи данных – две передачи данных за период тактового сигнала (по переднему и заднему фронту тактового импульса).

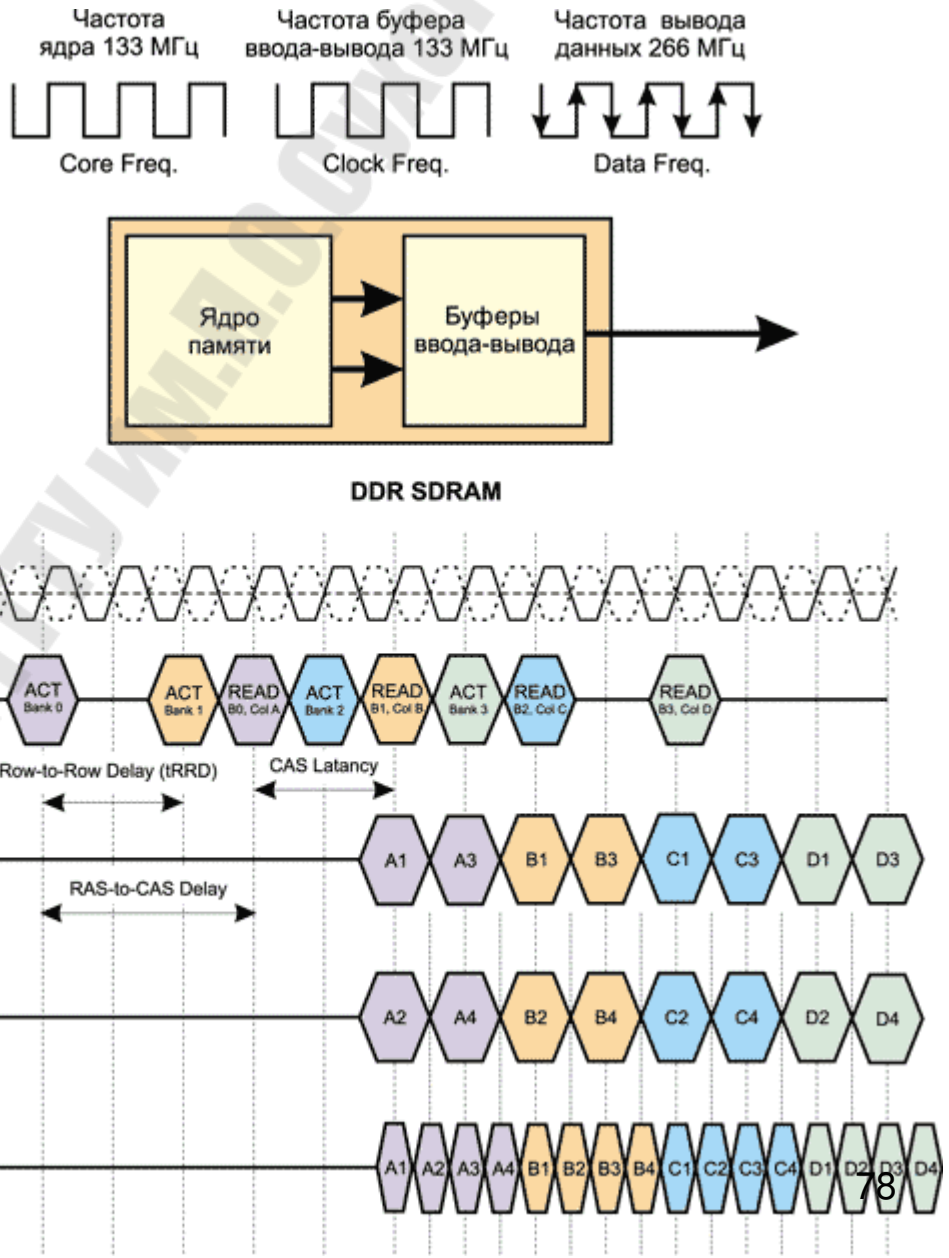
DIMM (*dual inline memory module)* - модуль памяти с двухрядным расположением выводов.
DIMM DDR5 имеет 288 контактов, ширина шины данных 64 бита; напряжение питания 1,1 В.

SODIMM (*small outline DIMM*) – компактная память для ноутбуков



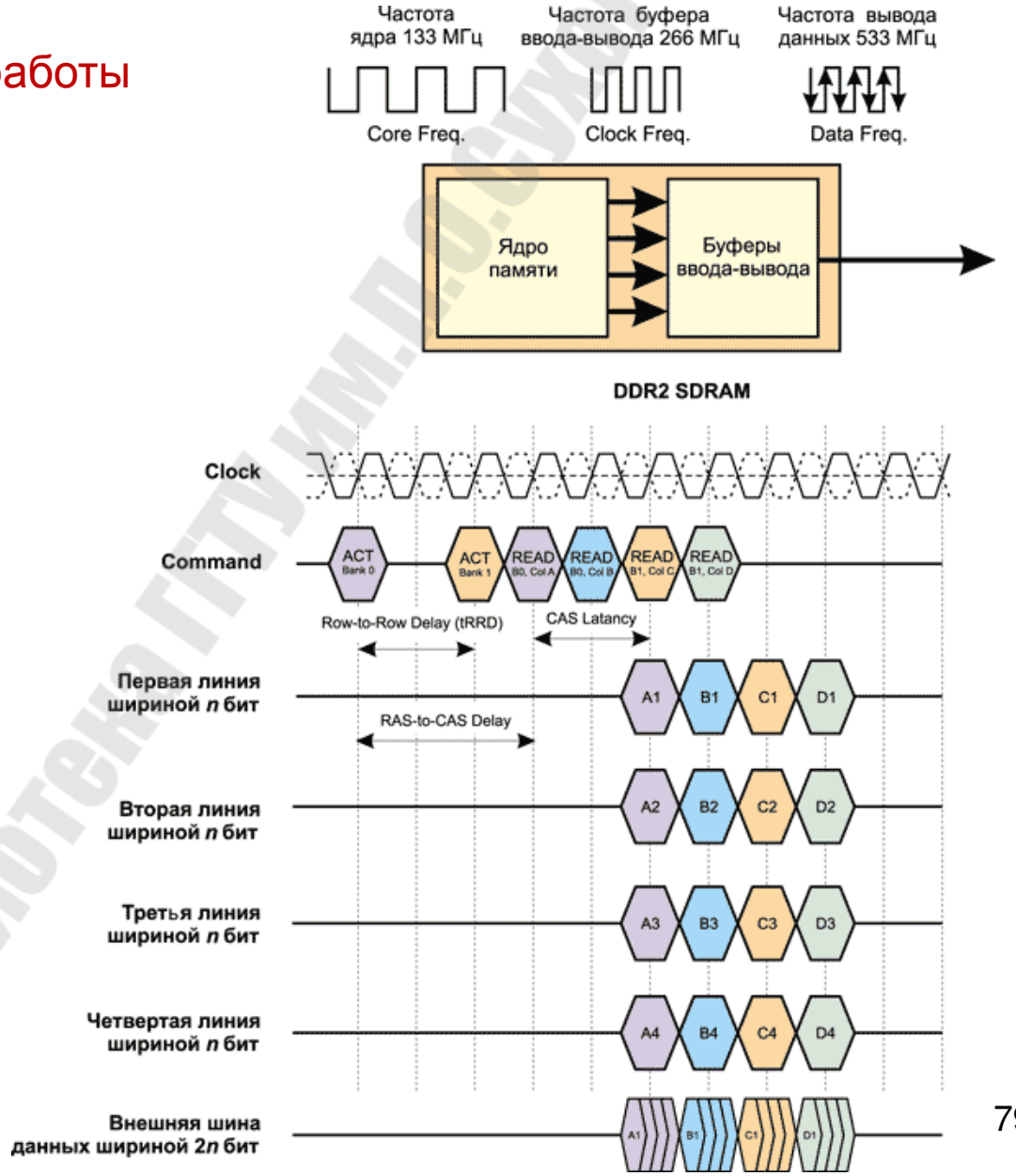
Организация памяти ЭВМ

Упрощенные диаграммы работы *DDR SDRAM*



Организация памяти ЭВМ

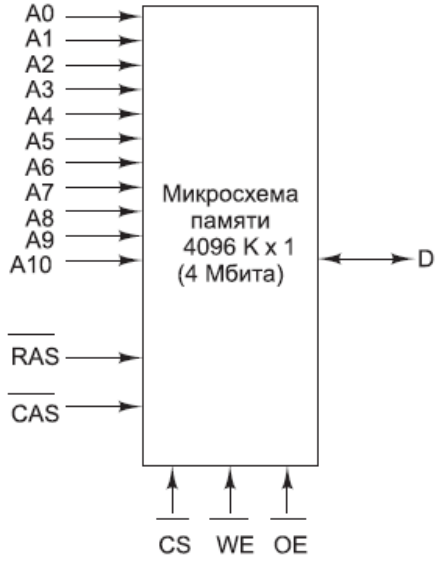
Упрощенные диаграммы работы *DDR2 SDRAM*

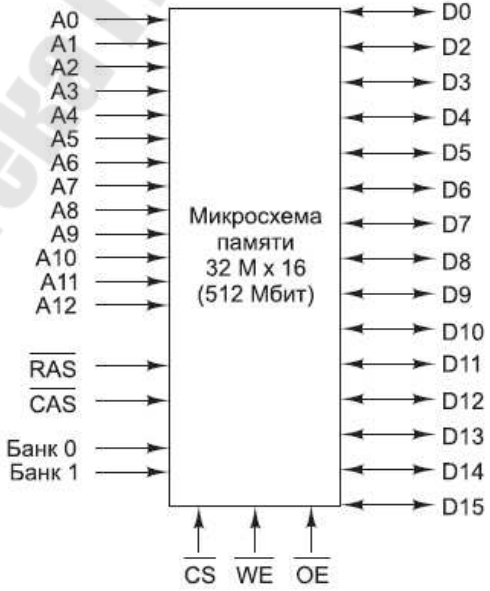


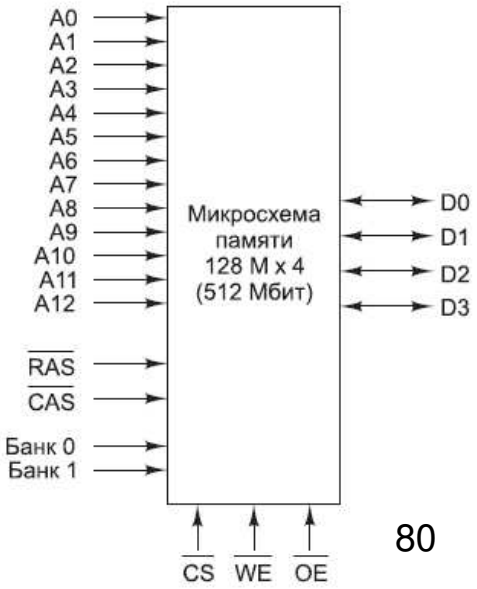
Организация памяти ЭВМ

Варианты организации микросхем памяти

Под **организацией ЗУ** (N x L) понимают число N кодовых слов, хранимых в ЗУ, с указанием их разрядности L. **Ёмкость ЗУ**: M = N x L. Например, ЗУ с организацией 32 x 8 и 256 x 1 имеют одинаковый объём памяти, равный 256 бит.







Организация памяти ЭВМ

Функциональная схема статической памяти

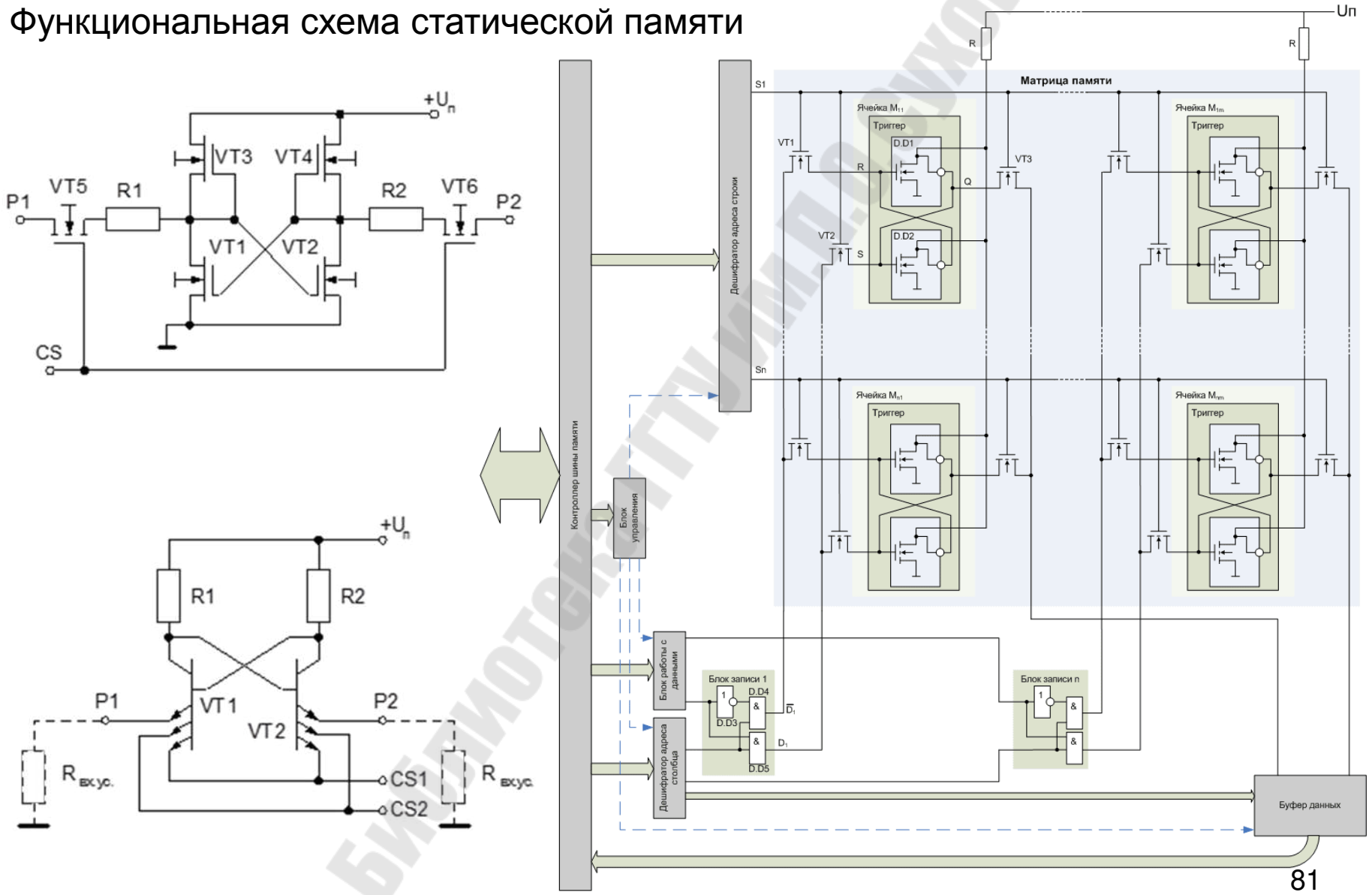
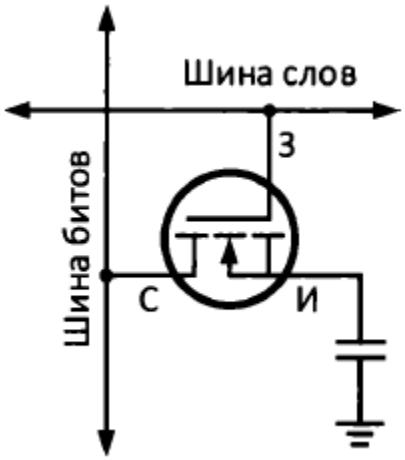
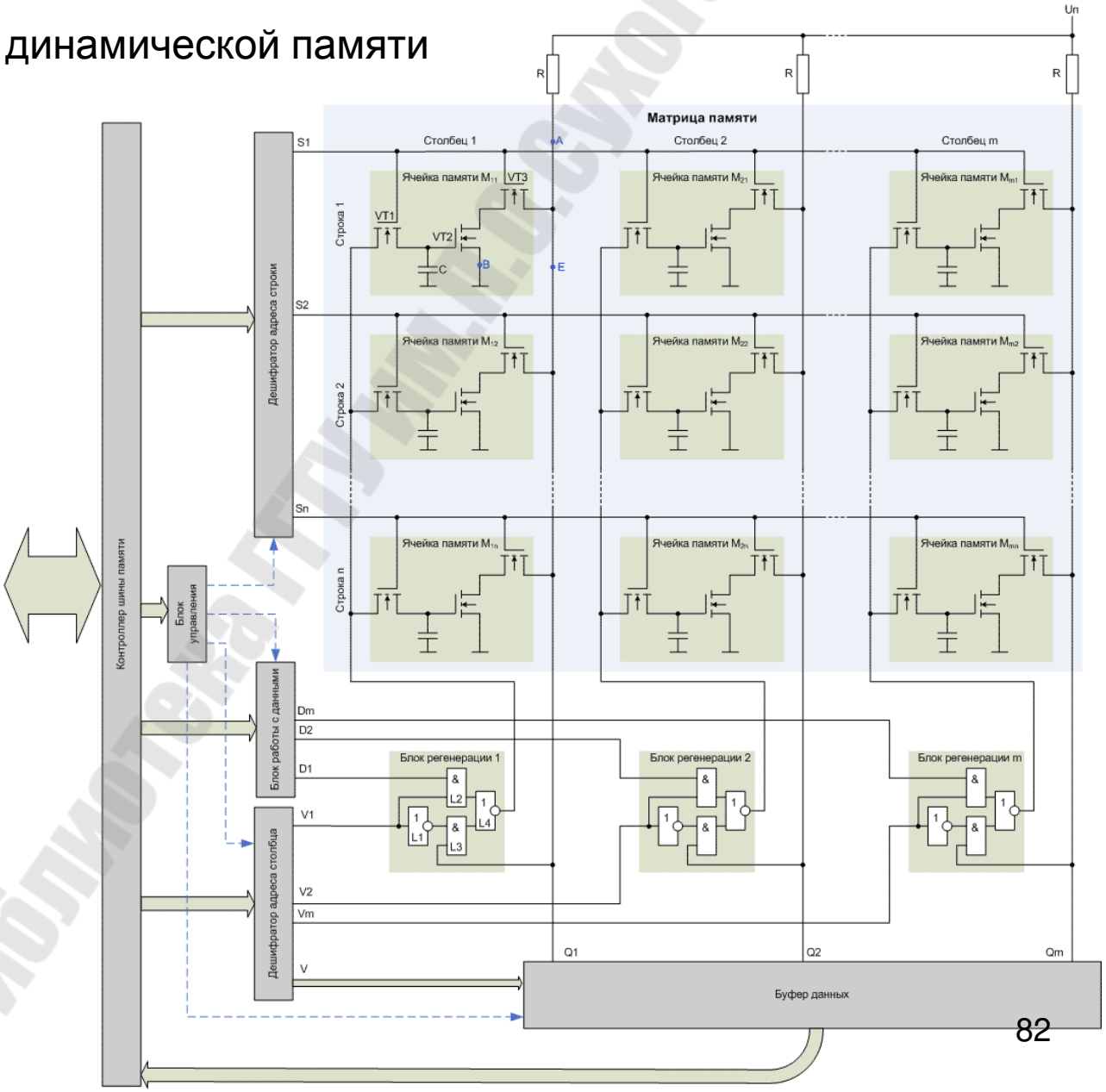


Рисунок 5. Упрощенная структурная схема статической оперативной памяти (SRAM)

Организация памяти ЭВМ

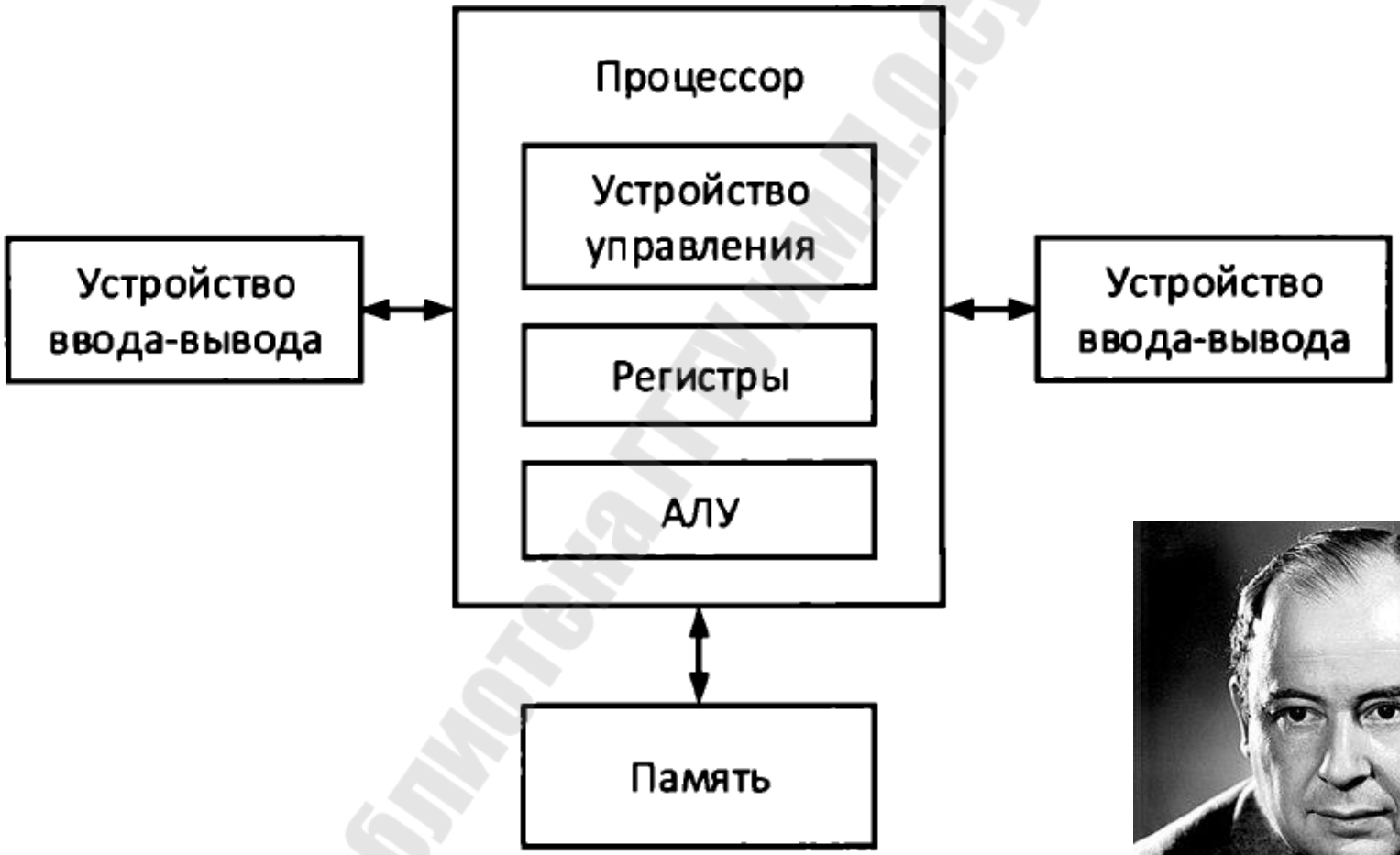
Функциональная схема динамической памяти





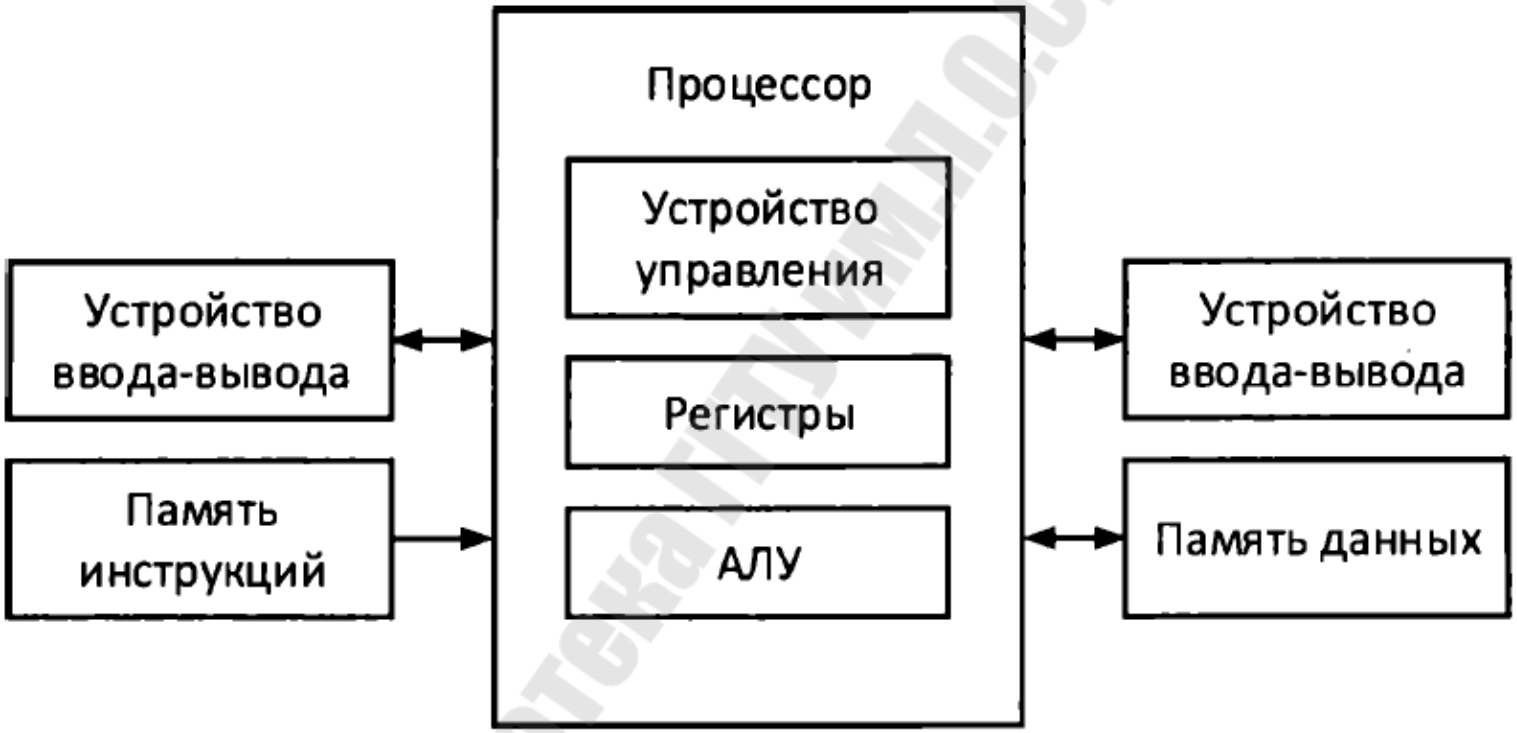
Организация памяти ЭВМ

Модель памяти фон Неймана (1945)



Организация памяти ЭВМ

Гарвардская модель памяти (1944) даёт одновременный доступ к инструкциям и данным



Организация памяти ЭВМ

Процессоры x86 имеют следующую разрядность шин данных (ШД) и адреса (ША):

Intel 8086 и 8088:
ШД: 16 бит.
ША: 20 бит. Внутренняя ША – 16 бит, внешняя ША – 20 бит.
Адресуются **до 1 МБ** памяти.

Intel 80286:
ШД: 16 бит.
ША: 24 бита.
Адресуются **до 16 МБ** памяти.

Intel 80386 и последующие модели:
ШД: 32 бита.
ША: 32 бита.
Адресуются **до 4 ГБ** памяти.

Intel x86-64 (*AMD64/Intel 64*):
ШД: 64 бита.
ША: 48 бит.
Адресуются **до 256 Тб** (терабайт) памяти.

Организация памяти ЭВМ

Основные режимы памяти в x86 процессорах:

- *реальный режим* (Real Mode): режим работы памяти в оригинальных 16-битных процессорах i8086/i8088. В архитектуре x86-32 это режим совместимости с16-битными процессорами x86. В реальном режиме процессор имеет прямой доступ ко всем 1 Мб адресуемой памяти и использует сегментированную адресацию;

- *защищенный режим* (Protected Mode): память разделена на сегменты, и процессор обеспечивает защиту памяти и привилегии доступа. Защищенный режим позволяет адресовать до 4 ГБ памяти;

- *режим виртуального 8086* (Virtual 8086 Mode): это режим, в котором можно запускать несколько 16-битных программ, работающих в реальном режиме, внутри 32-битной операционной системы.

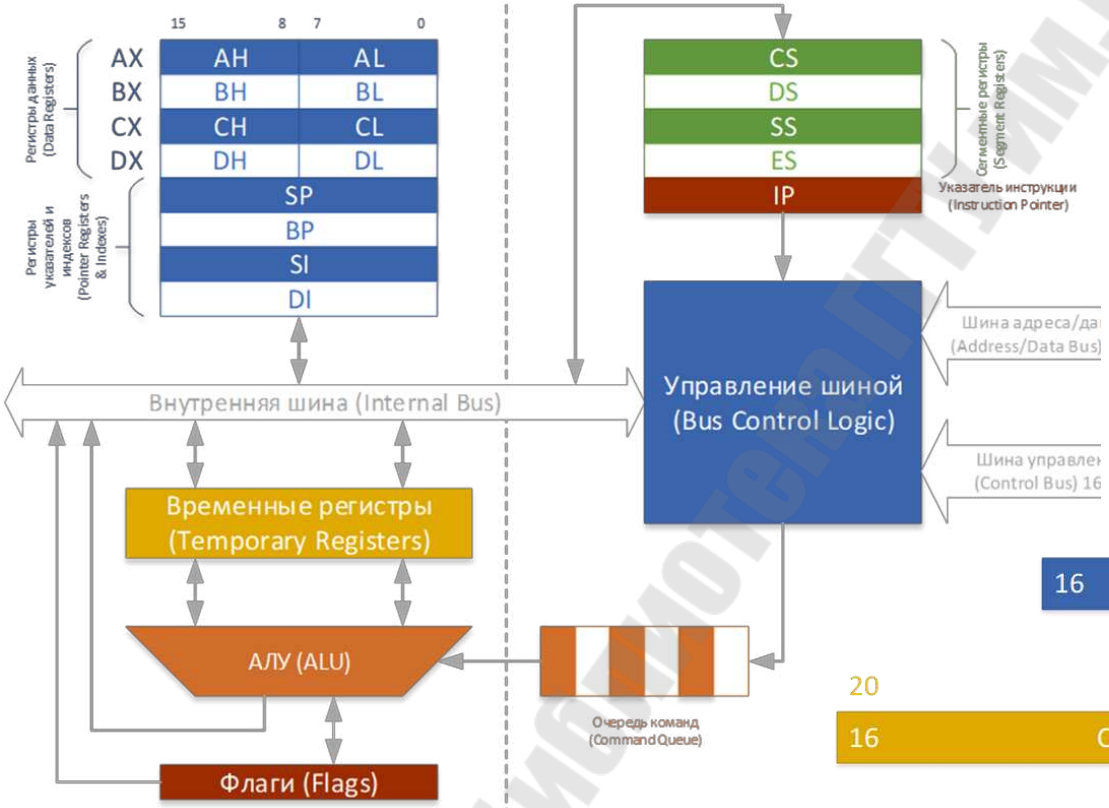
- *режим системного управления* (System Management Mode): специальный режим, используемый для реализации функций энергосбережения, управления системой и обработки системных событий;

- *64-битный режим* (Long Mode), был представлен в более новых процессорах x86-64 (например, AMD64 и Intel 64). В 64-битном режиме процессор может адресовать гораздо больше памяти и предоставляет дополнительные возможности для работы с большими объемами данных.

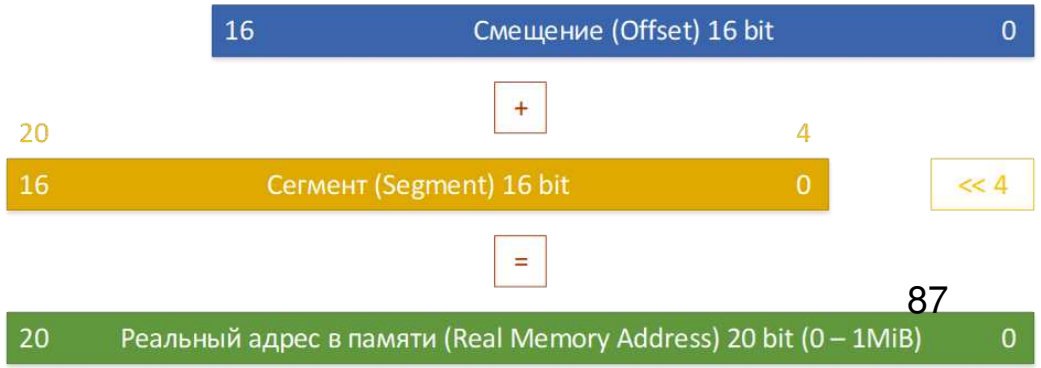
Организация памяти ЭВМ

Модель памяти в реальном режиме (R-режим 32-битного процессора)
Шина адреса имеет 20 бит и адресует 2^{20} = 1 048 576 = 1 Мб памяти.

Базовый адрес = Сегментный регистр << 4
Физический адрес = Базовый адрес + Смещение



CS (Code Segment) – сегмент кода, содержащий инструкции процессора;
DS (Data Segment) – сегмент данных;
SS (Stack Segment) – сегмент стека;
ES (Extra Segment) – дополнительный сегмент



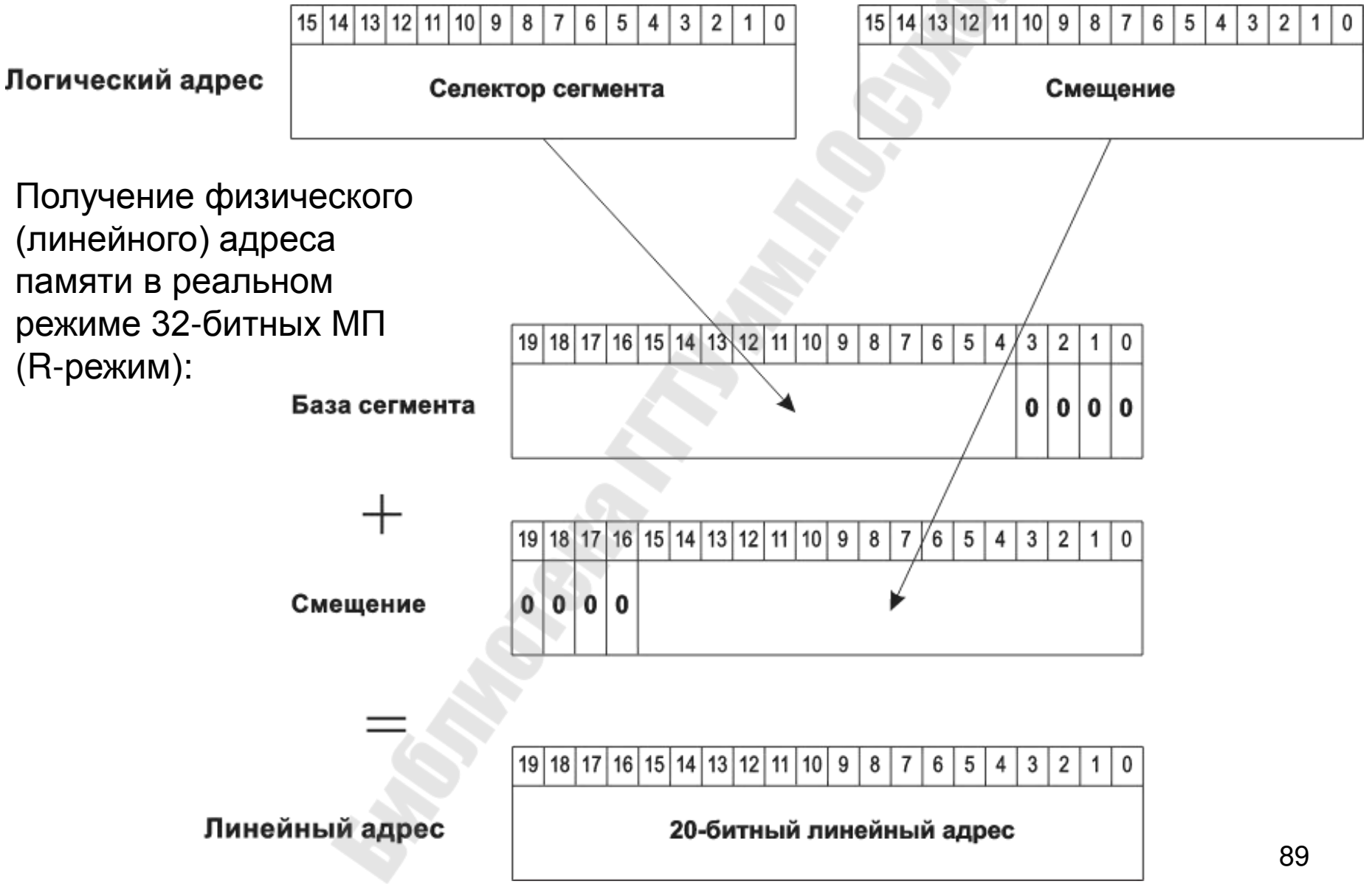
Организация памяти ЭВМ

Модель памяти в реальном режиме (память i8086/i80286):

адрес стека: SS:SP
адрес данных: DS:[offset]
адрес команд: CS:IP

Ближний переход – переход к команде в пределах данного сегмента (CS не меняем).
Дальний переход – переход к команде в другом сегменте (задаем CS и IP) – выполняется дольше.

Организация памяти ЭВМ



Пример 1

Пусть есть следующее значение регистра сегмента и смещения:

CS (Code Segment): 0x1000
IP (Instruction Pointer): 0x0010

Для вычисления физического адреса следует объединить сегментное значение и смещение следующим образом:

Физический адрес = (Сегментное значение << 4) + Смещение

В данном примере:

Физический адрес = (0x1000 << 4) + 0x0010 = 0x10010.

Пример 2

Предположим, что индекс – FFFFh и смещение – FFFFh, тогда

FFFF0h – индекс << 4
FFFFh – смещение
10FFEFh – 1 114 095$_{10}$ переполнение разрядности ША
|_ линия A20

При переполнении адреса в реальном режиме процессор может вести себя неопределенным образом. Возможны сбои системы, непредсказуемые результаты или зависание.

В реальном режиме процессора x86 линия A20 контролирует 21-й бит физического адреса, который определяет, какая область памяти будет доступна процессору. Этот бит отвечает за переход через границу 1 МБ памяти.

Организация памяти ЭВМ

Недостатки **реального режима**:

- ограничение объема адресуемой памяти: реальный режим ограничен адресацией 1 МБ памяти;

- *ограниченная безопасность*: реальный режим не предоставляет механизмов защиты памяти между различными программами или процессами. Любая программа может произвольно обращаться к памяти других программ или операционной системы, что может привести к сбоям системы или уязвимостям безопасности.

Организация памяти ЭВМ

Основные характеристики **защищенного режима** x86:

- защита памяти: в защищенном режиме используется механизм сегментации и страничного преобразования, позволяющий операционной системе контролировать доступ к памяти и обеспечивать изоляцию между различными процессами и задачами. Это помогает предотвратить ошибки программ и злонамеренные действия, такие как перезапись памяти или выполнение вредоносного кода.

- защита привилегий: защищенный режим позволяет определить уровни привилегий (кольца) для программного обеспечения. Обычно используется четыре уровня привилегий (с 0 до 3), где уровень 0 предоставляет наивысшие привилегии операционной системе, а уровни 1-3 предоставляют привилегии прикладным программам. Это позволяет операционной системе контролировать доступ к привилегированным инструкциям и ресурсам процессора.

Организация памяти ЭВМ

Системные регистры МП i486:

Системные регистры *GDTR*, *LDTR* и *IDTR* введены начиная с i80286 и предназначены для хранения базовых адресов таблиц дескрипторов при работе в защищенном режиме.

Организация памяти ЭВМ

Переключение между реальным и защищенным режимами:

Регистр CR0:

PE (*Protection Enable*) предназначен для переключения режима реальной адресации и защищенного режима;

Установить бит *PE* можно загрузкой слова состояния машины *MSW* с помощью команды *LMSW* или всего регистра *CR0* командой *MOV*.

Сбросить бит *PE* можно только загрузкой *CR0* командой *MOV*.

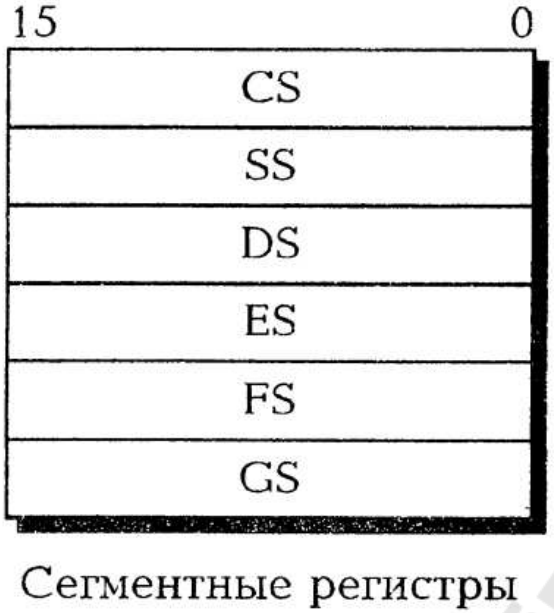
```
; Переход в защищенный режим
        mov eax, cr0
        or al, 1
        mov cr0, eax
```

Организация памяти ЭВМ

Адресация памяти в защищенном режиме 32-битных МП (Р-режим):

Селектор – это 16-битная величина, которая однозначно указывает на дескриптор в одной из таблиц дескрипторов.

Селекторы размещаются в сегментных регистрах *CS, DS, SS, ES, FS, GS*



Сегментные регистры

Структура селектора:

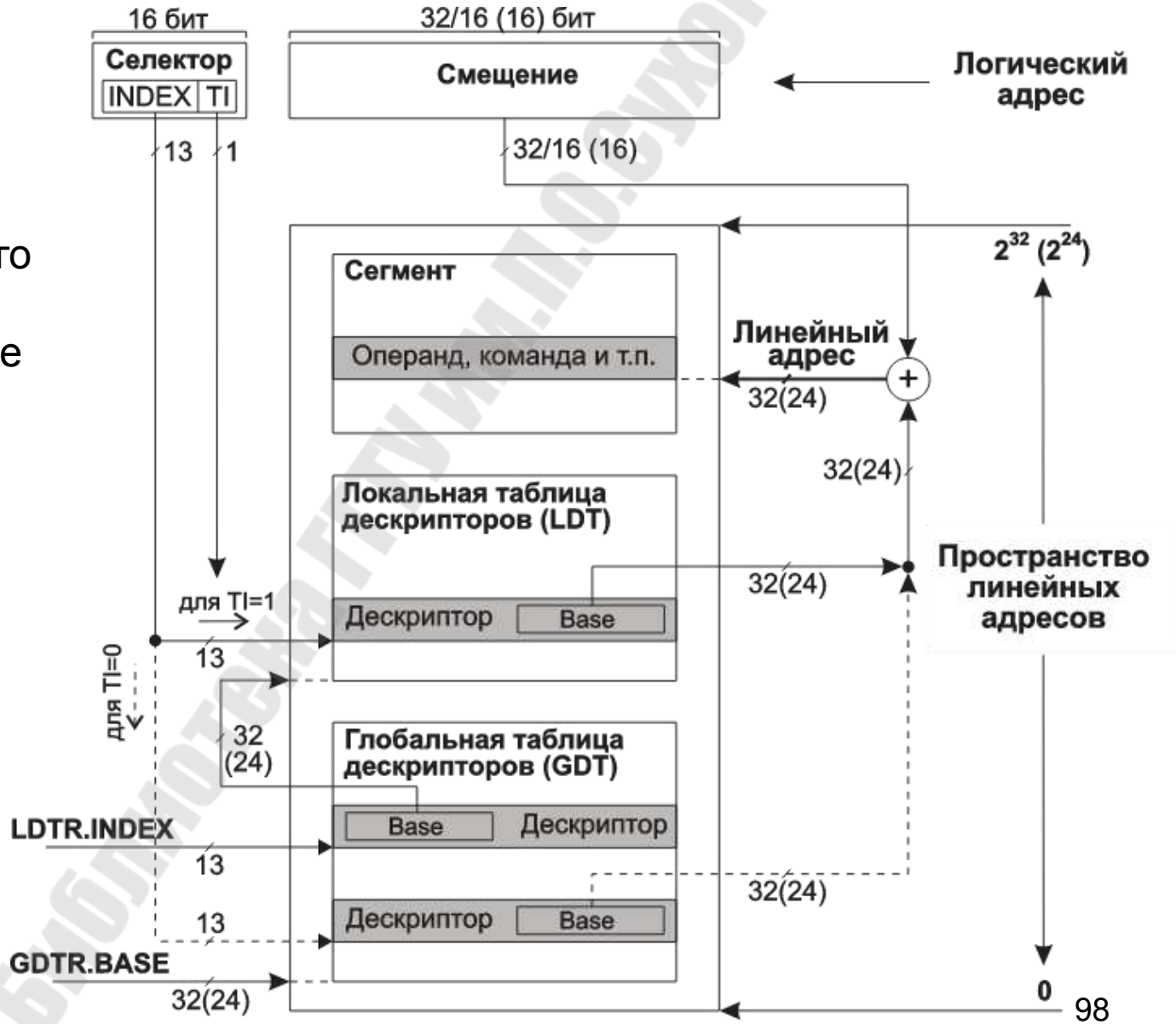
Организация памяти ЭВМ

Адресация памяти в защищенном режиме 32-битных МП (Р-режим):

Уровни привилегий и кольца защиты:

Память

Получение линейного адреса памяти в защищенном режиме 32-битных МП (Р-режим):



* В скобках даны значения для Intel286.

Организация памяти ЭВМ

Адресация памяти в защищенном режиме 32-битных МП (P-режим):

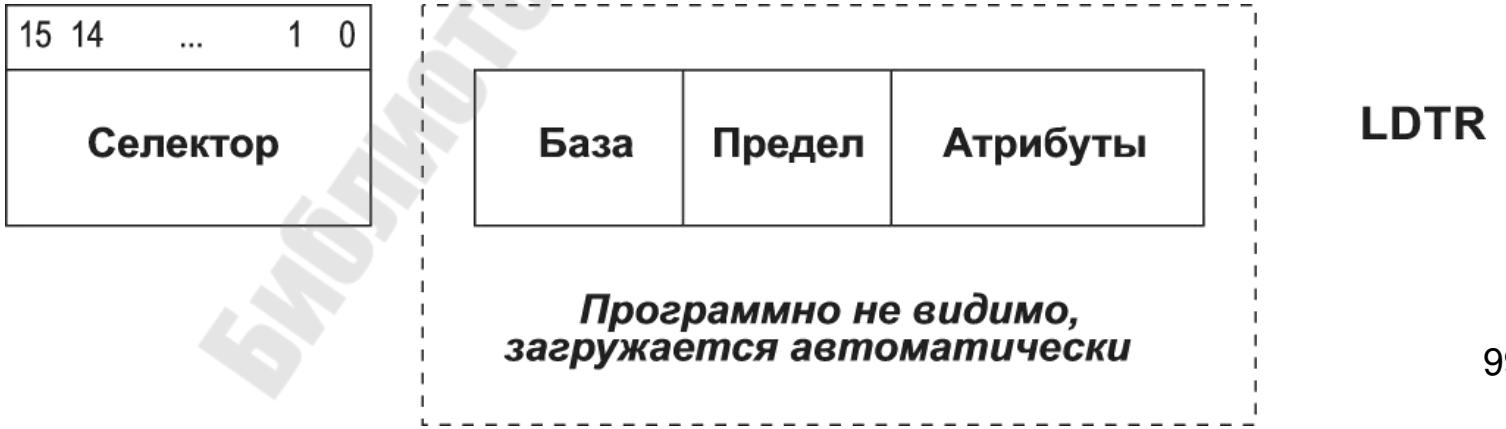
Index (биты 15..3) - указывает на нужный дескриптор в таблице дескрипторов.

адрес дескриптора = (*Index* << 3) + *GDTR.base* (*LDTR.base*)

Регистр глобальной таблицы дескрипторов *GDTR:*

* 24-битный линейный базовый адрес для Intel286

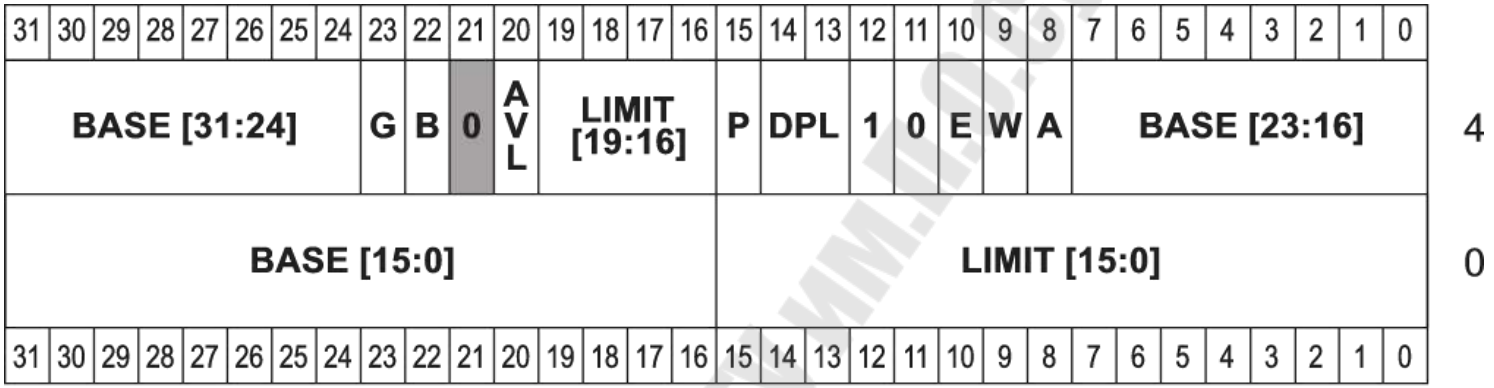
Регистр локальной таблицы дескрипторов *LDTR*:



Организация памяти ЭВМ

Адресация памяти в защищенном режиме 32-битных МП (Р-режим):

Формат дескриптора сегмента данных:



LIMIT - предел (Intel286 – 16 бит, Intel386 … – 20 бит):
- при *G* = 0, размер сегмента в байтах (от 1 байта до 1 Мб);
- при G = 1, размер сегмента в 4-Кб параграфах (от 4 Кб до 4 Гб).

BASE - база (Intel286 – 24 бита, Intel386 … – 32 бита): задает положение сегмента в 4 Гб (16Мб для Intel286) адресном пространстве. Рекомендуется выравнивание по 16 байтным границам.

DPL - уровень привилегий дескриптора.

E - расширяемость: E = 1 – сегмент расширяется вниз (стек);
E = 0 – сегмент расширяется вверх.

Адресация памяти в защищенном режиме 32-битных МП (Р-режим):

Формат дескриптора сегмента кода:

Бит 21 старшего двойного слова зарезервирован и всегда должен устанавливаться в 0.

Память

Для вычисления *эффективного адреса* в защищенном режиме выполняются следующие шаги:

- получение *сегментного дескриптора*: инструкция содержит информацию о сегменте памяти, в котором находится операнд. Сегментный дескриптор содержит базовый адрес сегмента и другие свойства, такие как размер и уровень привилегий доступа;
- вычисление *линейного адреса*: *базовый адрес* сегмента, указанный в сегментном дескрипторе, комбинируется с *смещением операнда*, указанным в инструкции, для получения линейного адреса. Линейный адрес представляет собой адрес в виртуальной памяти и не зависит от физической организации памяти.
- проверка *привилегий доступа*: при доступе к операнду проверяются привилегии текущего уровня привилегий и уровень привилегий сегмента. Если текущий уровень привилегий позволяет доступ к сегменту, и сегмент допустим (присутствует и активен), то можно продолжить операцию;
- *преобразование линейного адреса*: линейный адрес может быть преобразован в физический адрес через механизмы управления памятью, такие как страницы и трансляционные таблицы, которые обеспечивают отображение виртуальной памяти на физическую.

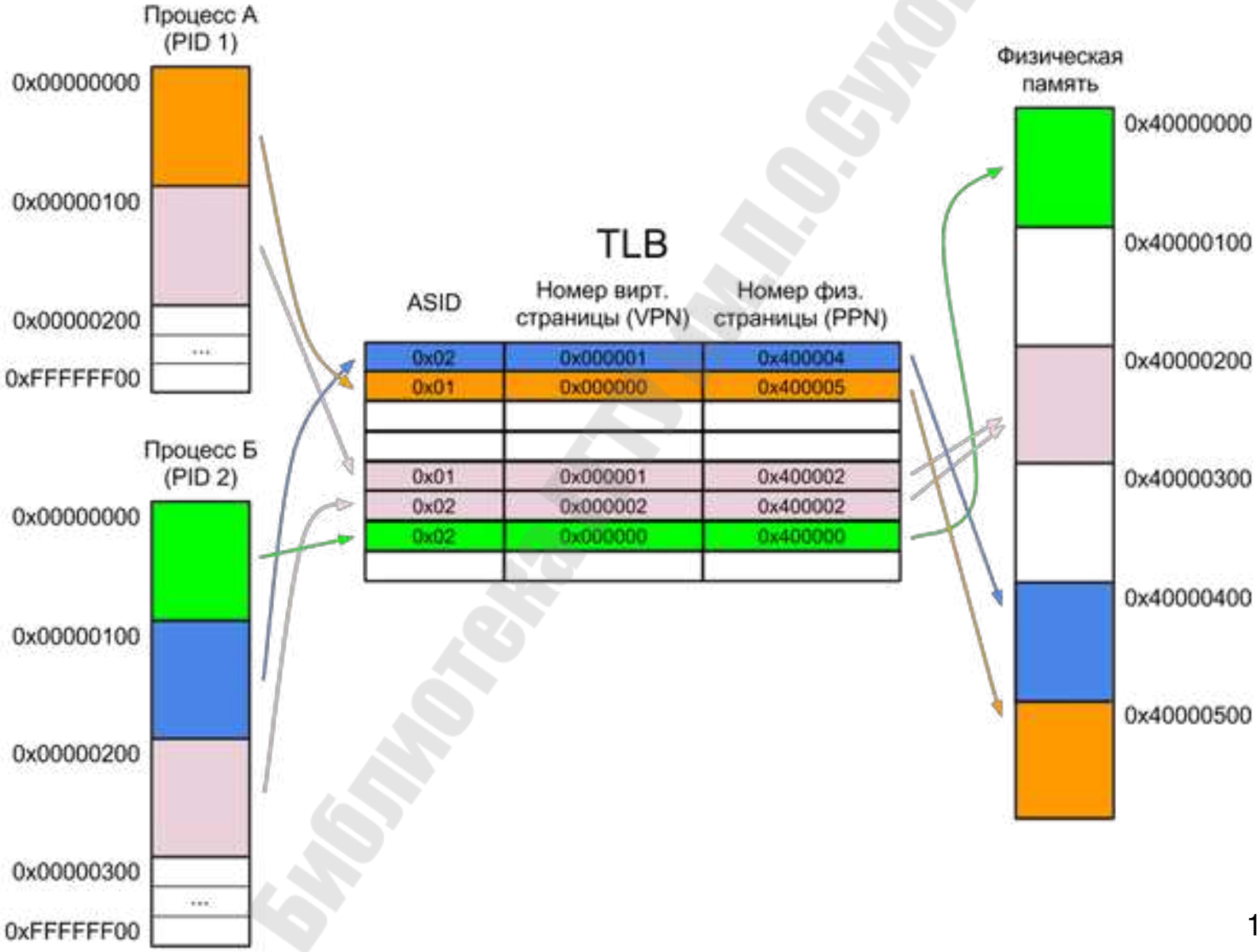
Организация памяти ЭВМ

Виртуальная память — виртуализация памяти позволяет каждой программе работать в собственном виртуальном адресном пространстве, независимо от физической памяти. Это позволяет операционной системе эффективно управлять памятью и защищать адресное пространство процессов друг от друга.

Виртуальная память обеспечивает безопасность и изоляцию процессов:

- каждому процессу предоставляется собственное адресное пространство;
- виртуальная память делится на страницы, по 4 кбайт. Когда процесс требует доступ к памяти, страница может быть загружена из диска в оперативную память;
- для преобразования виртуальных адресов в физические используются таблицы страниц;
- если оперативная память заполнена, система может перемещать неактивные страницы на диск в файл подкачки;
- виртуальная память помогает предотвратить доступ одного процесса к памяти другого.

Организация памяти ЭВМ



Организация памяти ЭВМ

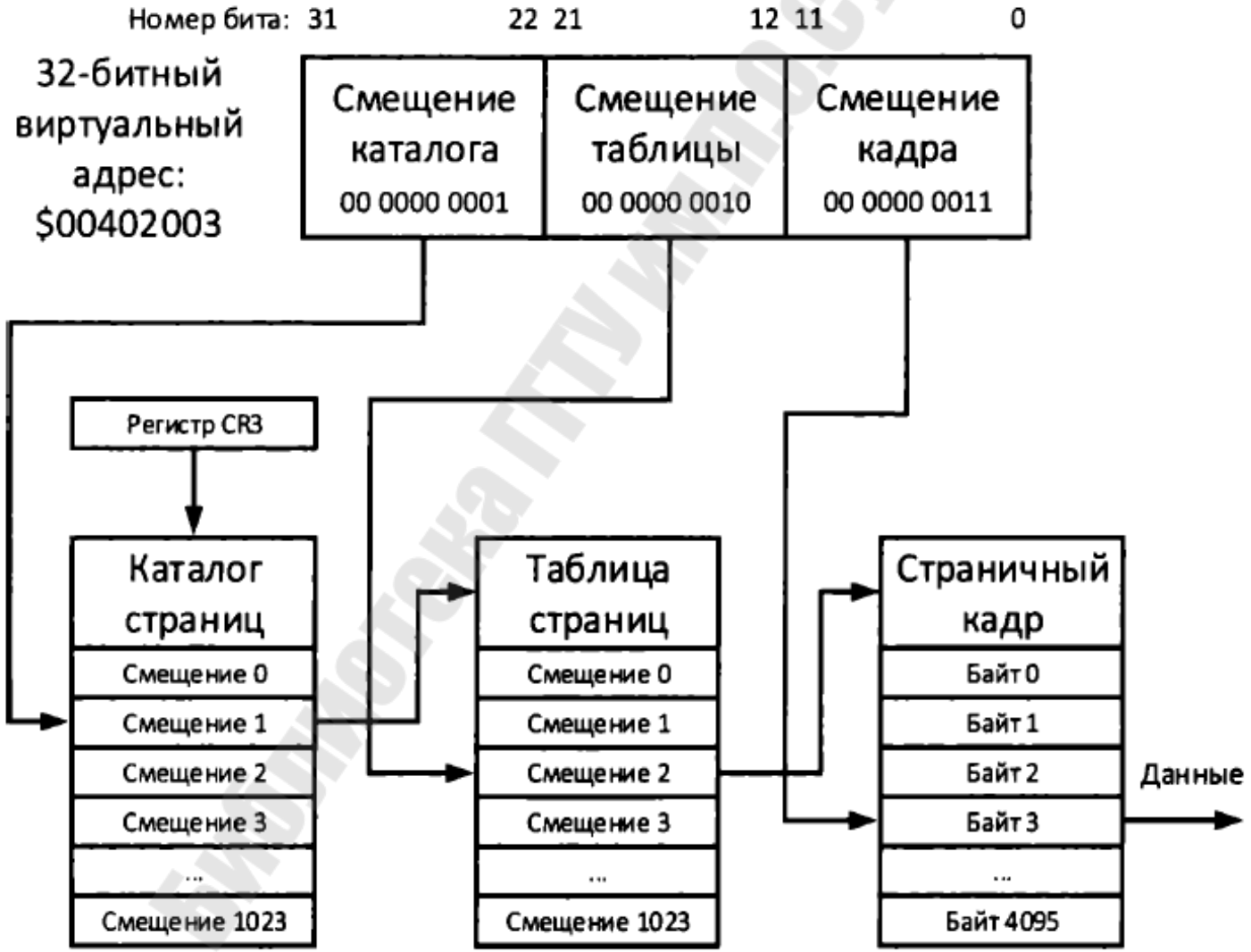
TLB (*translation look-aside buffer*) – это кэш внутри процессора, содержащий фиксированный набор записей (от 8 до 4096), каждая из которых содержит соответствие адреса виртуальной памяти адресу физической памяти.

ASID (*address space identifier*) – это 8 бит, в которые записан *PID* процесса, чтобы процесс Б не читал данные процесса А, когда пытается обратиться к тому же виртуальному адресу 0x00000000, к которому ранее обращался А.

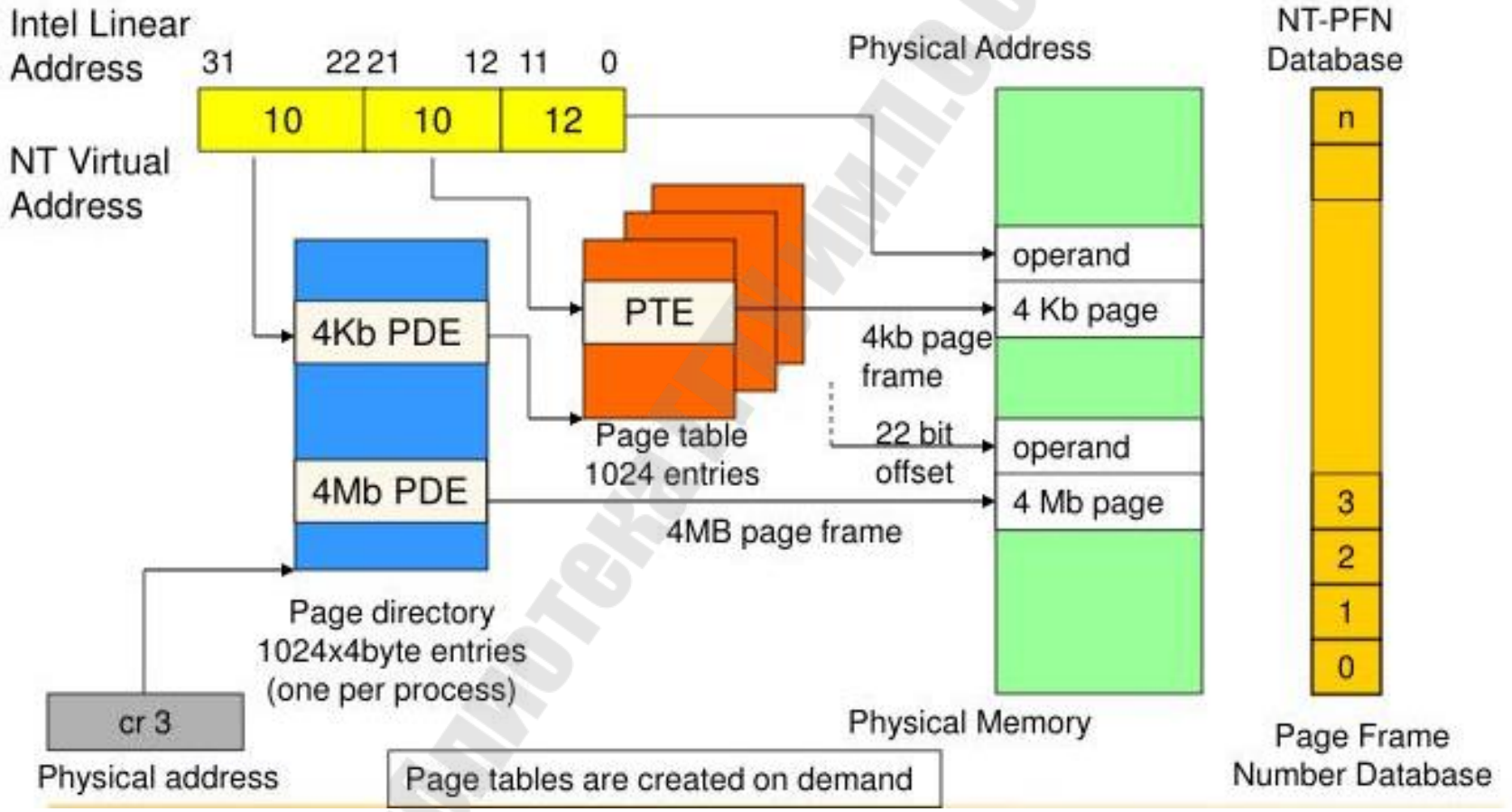
Если процессор поддерживает виртуализацию, то помимо *ASID* у него может быть еще и **VSID** (*virtual address Space IDentifier*), который содержит номер запущенной на процессоре виртуальной машины.

Организация памяти ЭВМ

Процессор использует механизм управления памятью (***MMU - Memory Management Unit***), чтобы переводить виртуальные адреса в физические.



Преобразование линейного адреса в физический в P-режиме:



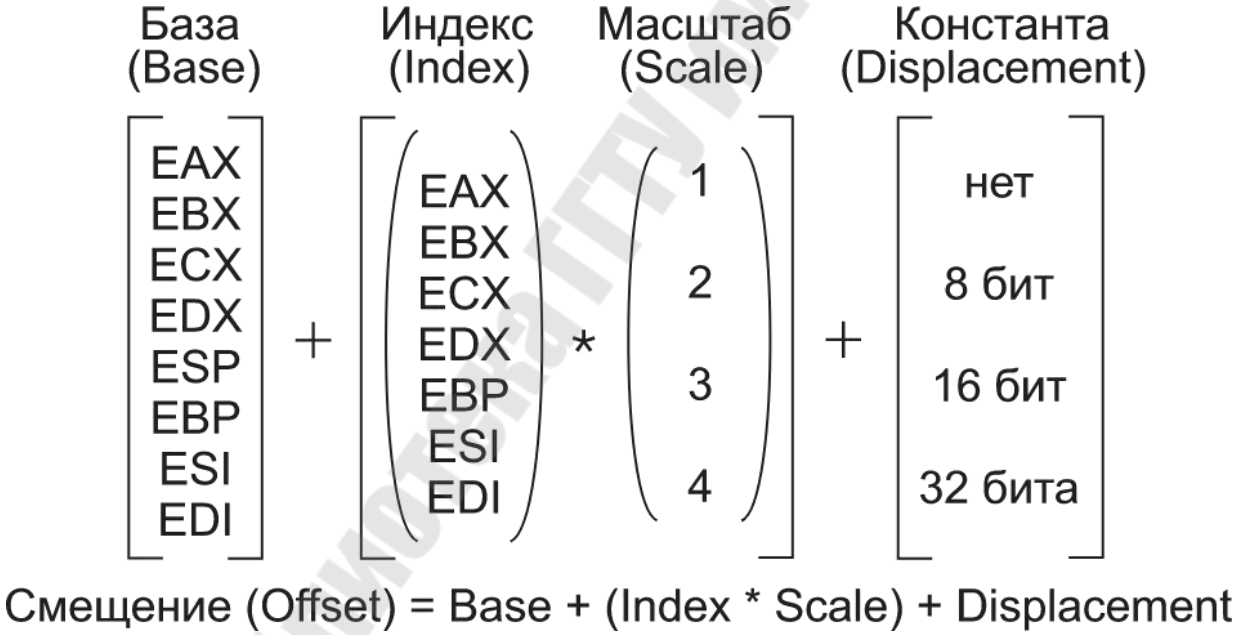
Организация памяти ЭВМ

Логический адрес – это адрес, с которым оперирует программное обеспечение.

Логический адрес =

= [16-битный селектор (указатель) сегмента]: [32(16)-битное смещение].

32(16)-битное смещение (offset):



Смещение (Offset) = Base + (Index * Scale) + Displacement

Организация памяти ЭВМ

Способы адресации 32-битных МП:

1) регистровая	*add eax, edx*
2) непосредственная	*add eax, 3A56C560h*
3) прямая	*add eax, [406023h]*
4) косвенно-регистровая	*add eax, [base]*
5) базовая	*add eax, [base] + disp*
6) индексная	*add eax, [index] + disp*
7) масштабиров. индексная	*add eax, [scale * index]+ disp*
8) базово-индексная	*add eax, [base][index]*
9) масштабиров. базово-индексная	*add eax, [base][scale * index]*
10) базово-индексная со смещен.	*add eax, [base][index] + disp*
11) масштабиров. базово-индексная со смещен	*add eax, [base][scale * index]+ disp*

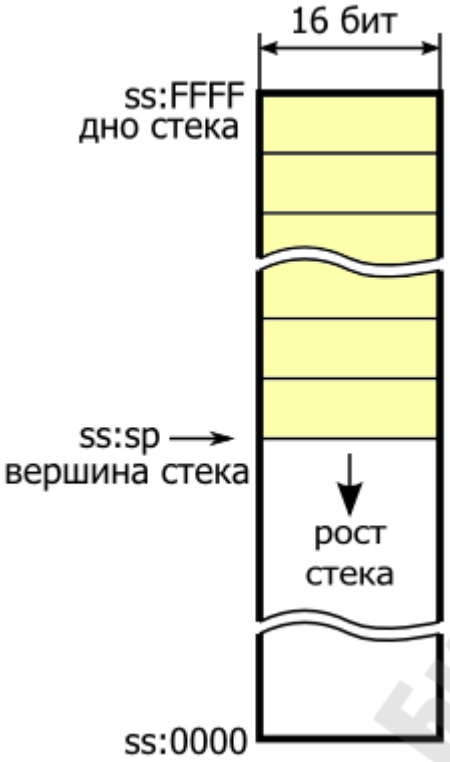
base – базовый регистр – любой 32-битный РОН
index – индексный регистр – любой 32-битный РОН, кроме ESP
scale – масштаб 2, 4, 8
disp – смещение 8, 16, 32-битное число

Организация памяти ЭВМ

Стек вызовов – в архитектуре x86 это структура данных, используемая для управления вызовами функций и хранения **локальных переменных**, параметров и адресов возврата.

Стек – это непрерывная область памяти, адресуемая регистрами *ESP* и *SS,* и организованная по принципу ***LIFO*** (***L****ast* ***I****n* – ***F****irst* ***O****ut*).

За одну операцию в стек можно поместить (забрать) слово (*2 байта*) или двойное слово (*4 байта*).



В архитектуре x86 (32-битный режим) дно стека размещают по адресу ***0xFFFFFFFF*** (4 гигабайта в адресном пространстве).

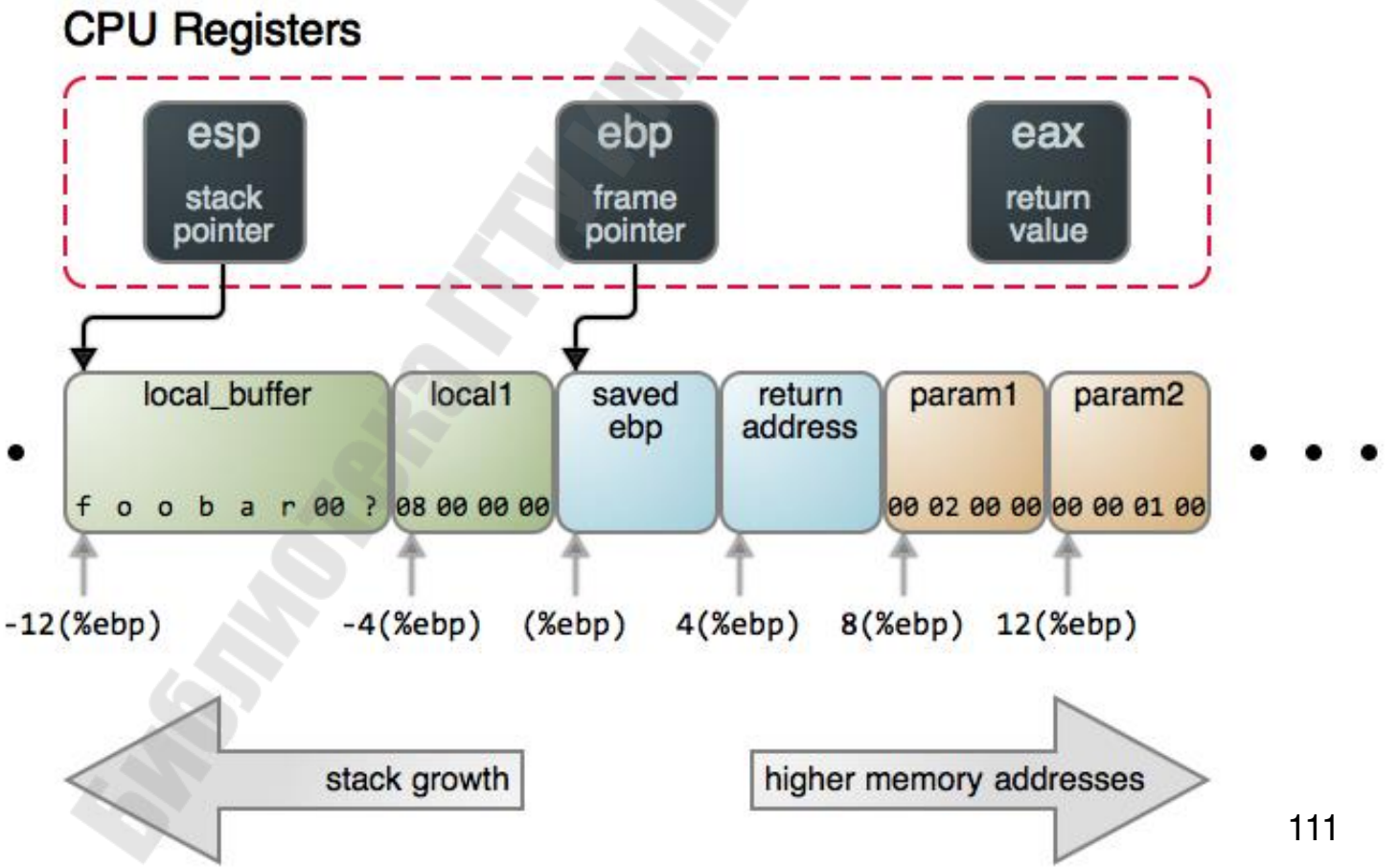
В x86_64 (64-битный режим) дно стека обычно располагается по адресу ***0x7FFFFFFFFFFFFFFF*** (наивысший адрес в 64-битной адресации).

Организация памяти ЭВМ

Стек поддерживается на аппаратном уровне регистрами:

ESP (*RSP*) – указатель стека – всегда указывает на вершину стека;
EBP – базовый указатель или указатель базы стекового кадра;
EAX – используется для хранения данных, возвращаемых функцией.

Кадр (фрейм) стека



Организация памяти ЭВМ

При вызове функции в стеке обычно сохраняются:

- ***адрес возврата***: Это адрес инструкции, куда нужно вернуться после завершения выполнения вызываемой функции. Он помещается на вершину стека и будет использоваться при последующем возврате из функции;

- ***состояние регистров***: В некоторых архитектурах процессоров регистры сохраняются на стеке при вызове функции. Это позволяет вызываемой функции использовать регистры для своих нужд, не опасаясь потерять значения из предыдущего контекста;

- ***аргументы функции***: Параметры, переданные в вызываемую функцию, могут быть сохранены на стеке. Это обеспечивает доступность параметров внутри функции и их восстановление после возврата;

- ***локальные переменные***: Если функция имеет локальные переменные, они могут быть размещены на стеке. Каждый вызов функции создает новый экземпляр локальных переменных на стеке, что позволяет функциям иметь собственные независимые значения переменных;

- ***другие контекстные данные***: В некоторых случаях могут быть сохранены другие контекстные данные, такие как указатели на структуры данных или состояние регистра флагов, в зависимости от требований архитектуры и компилятора.

Организация памяти ЭВМ

Роль регистра EBP в организации стека вызовов:

- ***установка базового указателя стека***: При входе в функцию регистр EBP обычно устанавливается равным значению регистра ESP. Это создает базу или точку отсчета для доступа к локальным переменным и параметрам функции;

- ***сохранение предыдущего контекста***: Значение регистра EBP сохраняется на стеке, чтобы запомнить предыдущий контекст перед входом в функцию. Это позволяет функции восстановить предыдущее значение EBP при выходе и вернуться к предыдущему контексту вызывающей функции;

- ***доступ к локальным переменным и параметрам***: Регистр EBP используется для доступа к локальным переменным и параметрам функции. Путем использования смещений относительно значения EBP можно получить доступ к нужным данным, которые были размещены на стеке при вызове функции;

- ***сдвиг стека***: При вызове функции и добавлении данных на стек, регистр *ESP* обычно сдвигается вниз, а регистр *EBP* остается неизменным. Это позволяет сохранить структуру стека и обеспечить доступ к предыдущим данным и параметрам.

Организация памяти ЭВМ

Соглашения о вызовах (*calling conventions*) — это правила, которые определяют, как вызываются функции и как передаются параметры между вызывающей и вызываемой функцией.

Соглашения о вызовах:

cdecl *(C Declaration)* - стандартное соглашение о вызовах для языка C.

Используется в большинстве компиляторов.

Параметры передаются справа налево.

Вызывающая функция очищает стек.

Поддерживает функции с переменным количеством параметров.

```
void  function(int a, int b);
```

stdcall: Параметры передаются справа налево (как в *cdecl*).

Вызываемая функция очищает стек.

Не поддерживает переменное количество параметров.

```
void  __stdcall   function(int a, int b);
```

fastcall - оптимизированное соглашение для повышения производительности.

Первые два параметра передаются через регистры (*ECX* и *EDX* в x86).

Вызываемая функция очищает стек.

```
void  __fastcall   function(int a, int b, int c);
```

a будет в *ECX*, *b* будет в *EDX*, *c* будет в стеке.

Организация памяти ЭВМ

Операции:

PUSH <reg> - уменьшает ESP и сохраняет значение на вершине стека.

POP <reg> - извлекает значение с вершины стека и увеличивает ESP.

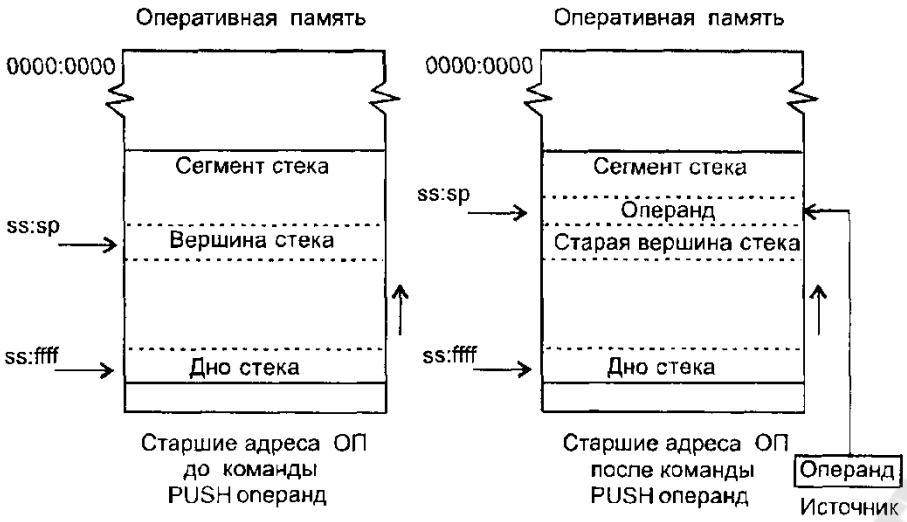
CALL <addr> - сохраняет адрес возврата в стеке и передает управление вызываемой функции.

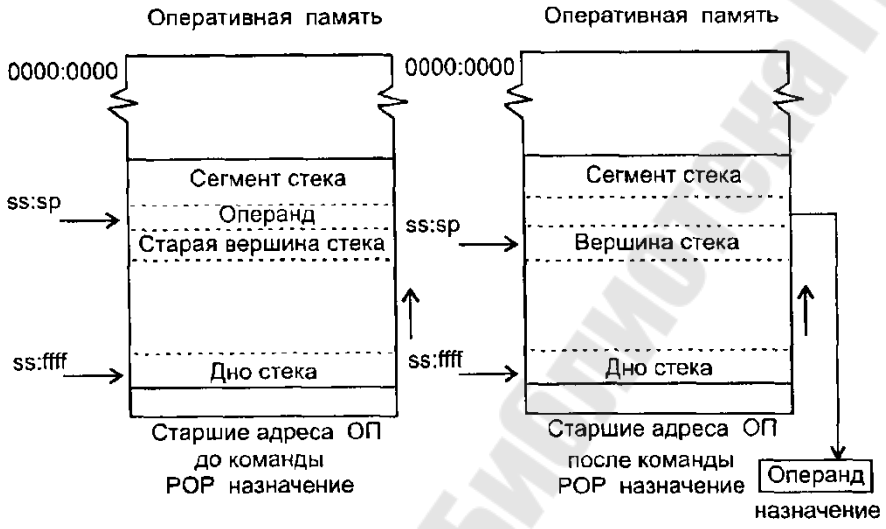
RET - извлекает адрес возврата из стека и передает управление обратно.

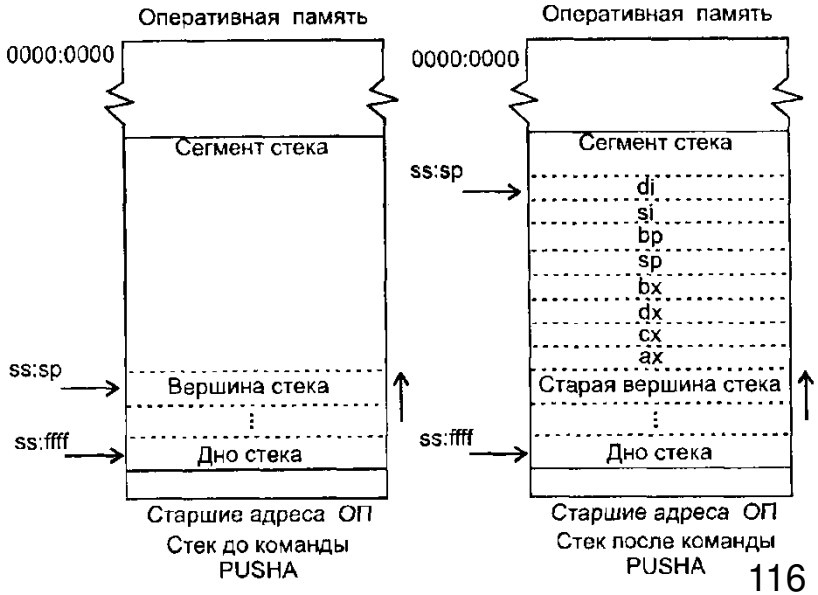
Разновидности команд PUSH и POP:

PUSH r/m	Помещение слова (из регистра или памяти) в стек
PUSH immed	Помещение слова в стек (286+)
PUSHA (PUSH All)	Помещение в стек регистров AX, BX, CX, DX, SI, DI, BP, SP (286+) или их 32-битных расширений (386+) (обратно по действию PUSHA)
POP r/m	Извлечение слова данных из стека в регистр или память
POPA (POP All)	Извлечение данных из стека в регистры AX, BX, CX, DX, SI, DI, BP, SP (286+) или их 32-битных расширений (386+)
PUSHF (PUSH Flags)	Помещение в стек регистра флагов
POPF (POP Flags)	Извлечение данных из стека в регистр флагов

Организация памяти ЭВМ







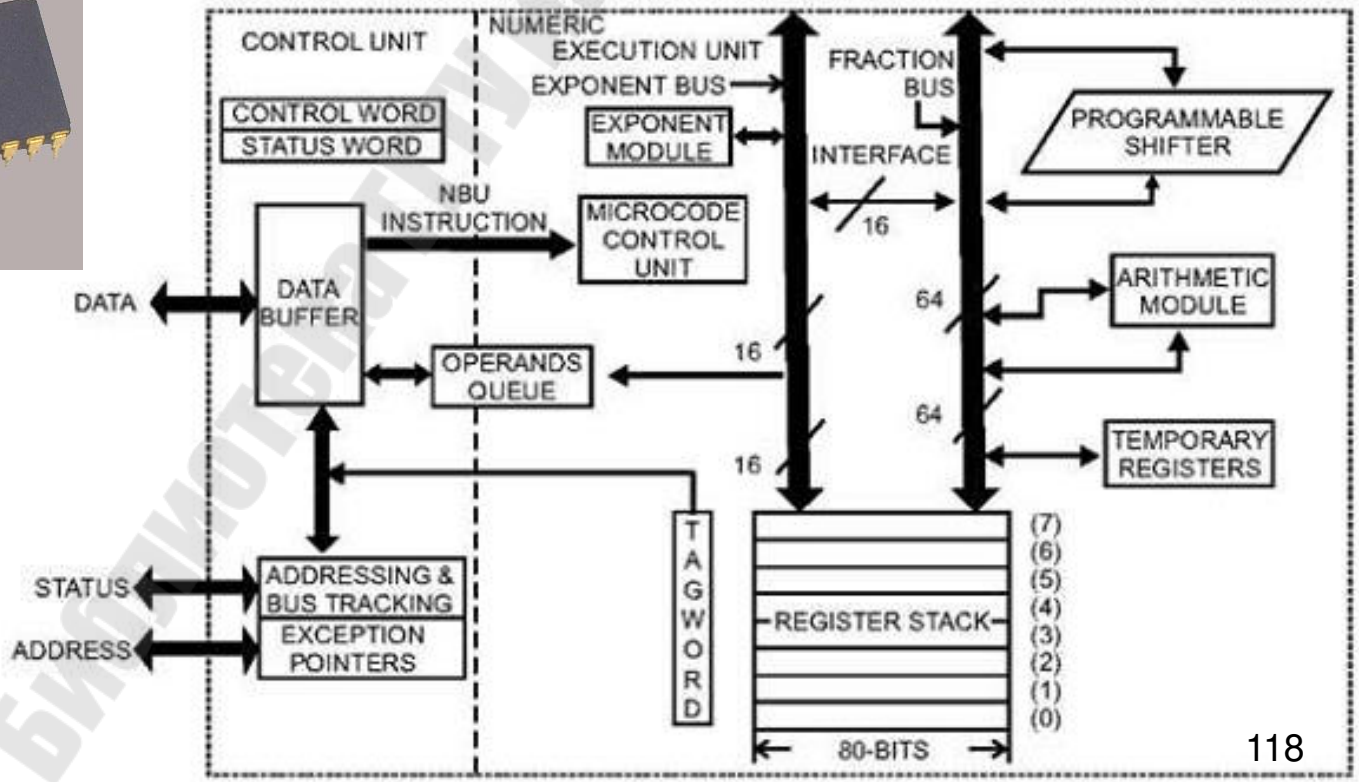
Тема 5
Арифметический сопроцессор

Арифметический сопроцессор

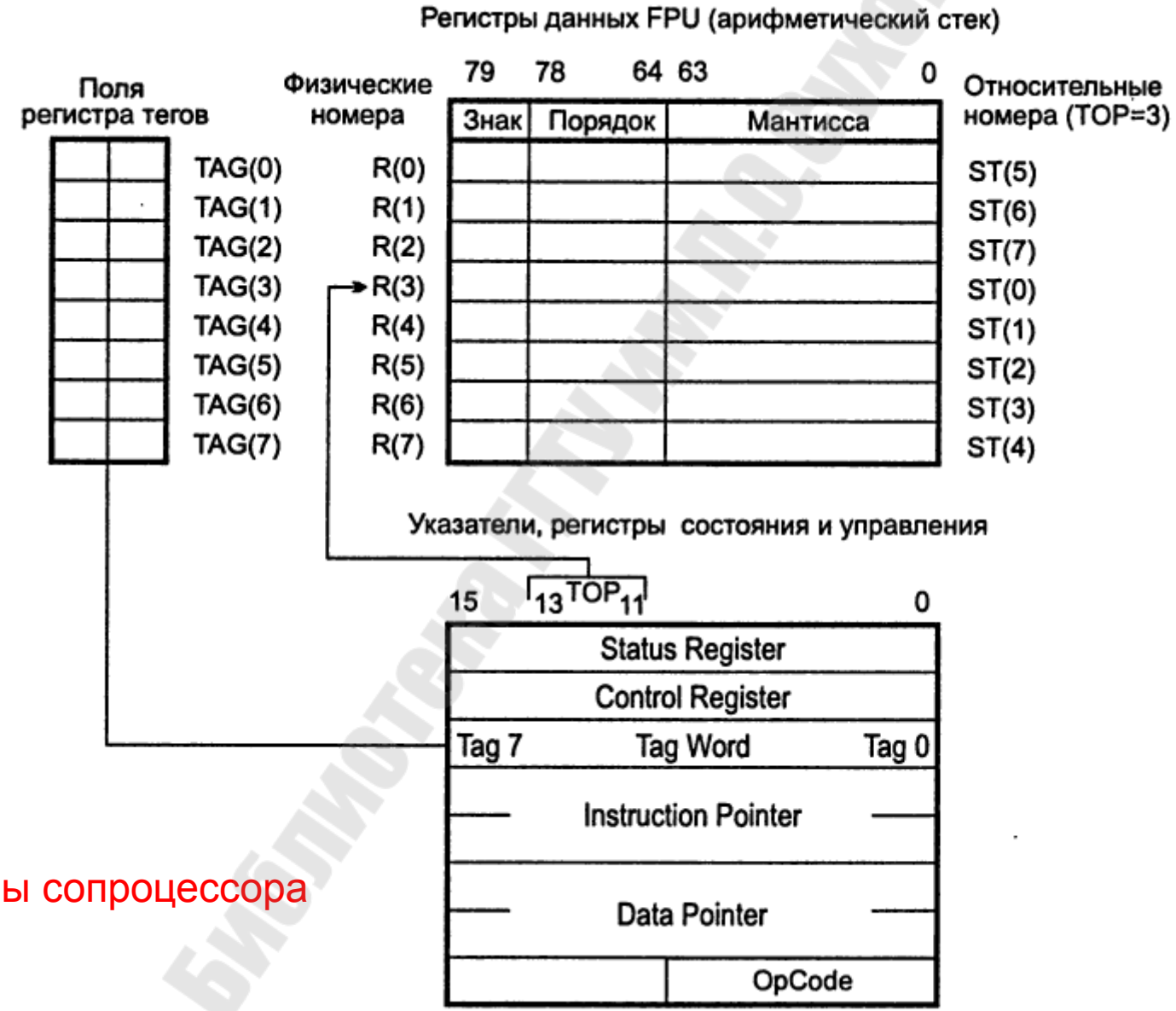
Сопроцессор предназначен для выполнения арифметических, тригонометрических, экспоненциальных, логарифмических и др. вычислений.

Поддерживает:

- 16-, 32-, 64-разрядные целые числа;
- 32-, 64-, 80-битные числа с плавающей точкой (стандарт IEEE754);
- 18-битные двоично-десятичные числа;
- R-режим и P-режим



Арифметический сопроцессор



Регистры сопроцессора

Арифметический сопроцессор

Регистр TAG содержит двухбитный тег для каждого регистра x87 FPU

TAG7	TAG6	TAG5	TAG4	TAG3	TAG2	TAG1	TAG0

Значение тегов описывает содержимое регистров

00: число считается допустимым и корректным;
01: число считается бесконечностью (положительной или отрицательной);
10: число считается неопределенным (NaN - Not-a-Number);
11: число считается ошибочным.

При пустом стеке ST(0) соответствует R7, при заполненном стеке ST(0) соответствует R0.

Арифметический сопроцессор

Слово состояния (Status Word Register):

B – бит занятости (для совместимости с *i8087*);
TOP – вершина арифметического стека;
C[3:0] – флаги для операций сравнения: C0 -> CF (перенос), C2 -> PF (чётность), C3 -> ZF (ноль);
ES – немаскируемые ошибки;
SF – ошибка в работе стекового регистра;
PE (*Precision Error*) – ошибка точности. Устанавливается, когда сопроцессору приходится округлять результат из-за того, что его точное представление невозможно. Так, сопроцессору никогда не удастся точно разделить 10 на 3;
UE (*Underflow Error*) – ошибка антипереполнения. Возникает, когда результат слишком мал (близок к нулю, но не 0);
OE (*Overflow Error*) – ошибка переполнения. Возникает в случае выхода операнда за максимально допустимый диапазон;
ZE (*divide by Zero Error*) – ошибка деления на 0;
DE (*Denormalized operand Error*) – денормализированный операнд;
IE (*Invalide operation Error*) – недопустимая операция (деление на 0, умножение на бесконечность, корень из отрицательного числа и т.п.)

Арифметический сопроцессор

Управляющее слово (Control Word Register):

RC – управление округлением:

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.

PC - управление точностью: 00 – длина мантиссы 24 бита;
10 – длина мантиссы 53 бита;
11 – длина мантиссы 64 бита.

OM (*Overflow Mask*):
0: исключение *Overflow* не будет генерироваться при переполнении.
1: исключение *Overflow* будет генерироваться при переполнении.

Арифметический сопроцессор

Команда *FINIT* — инициализация сопроцессора. Данная команда инициализирует управляющие регистры сопроцессора определенными значениями:

CWR = 037Fh:

- RC = 00b – округление к ближайшему целому;
- PM, UM, OM, ZM, DM, IM = 1 – все исключения замаскированы;
- PC = 11b – максимальная точность 64 бита.

SWR = 0h – отсутствие исключений и указание на то, что физический регистр стека сопроцессора R0 является вершиной стека и соответствует логическому регистру ST(0);

TWR = FFFFh – все регистры стека сопроцессора пусты;

DPR = 0h;

IPR = 0h.

Данную команду используют перед первой командой сопроцессора в программе и в других случаях, когда необходимо привести сопроцессор в начальное состояние.

Арифметический сопроцессор

FCLEX (*FPU Clear Exceptions*) очищает регистр статуса FPU, обнуляя все флаги исключений, такие как переполнение (*overflow*), деление на ноль (*zero divide*), недопустимая операция (*invalid operation*) и другие.

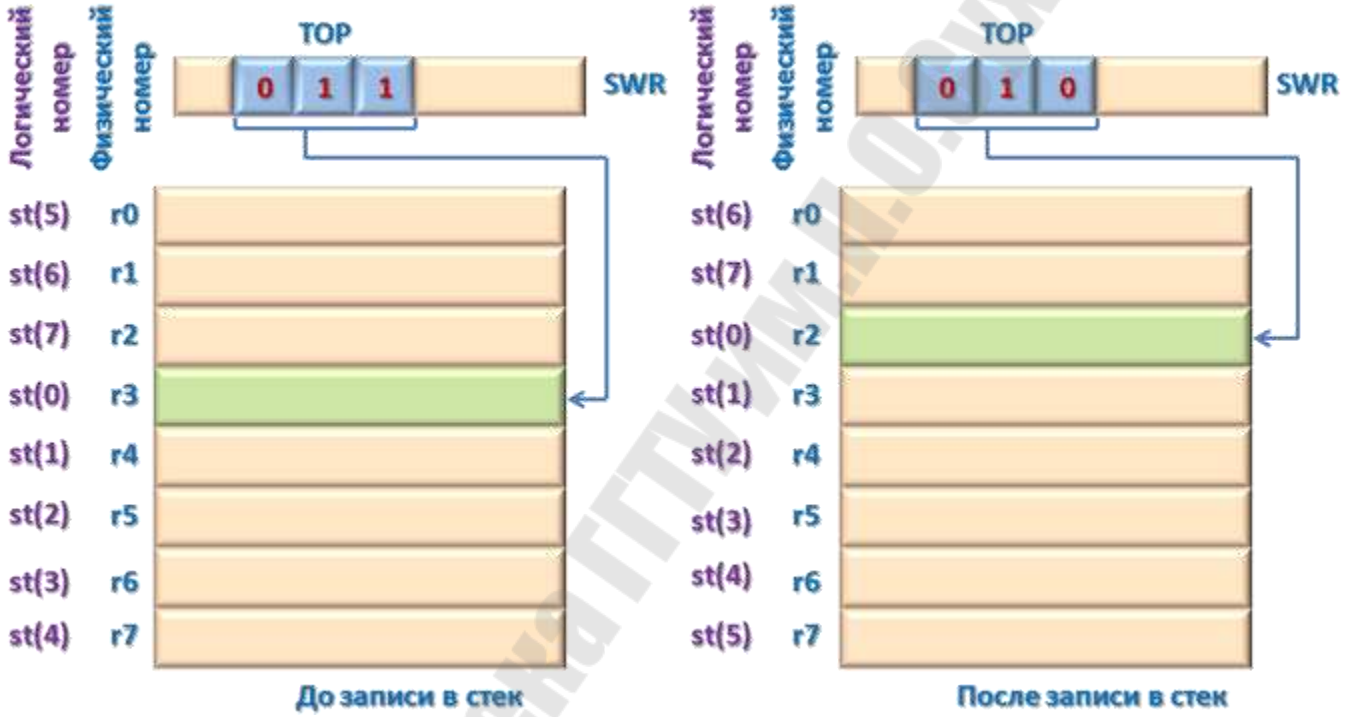
FSTSW (*FPU Store Status Word*) записывают текущее значение слова состояния FPU (SW) в память или в регистр AX.

FSTCW (*Fpu STore Control Word*) - это инструкция копирует значение регистра *Control Word* (CW) в указанный операнд в памяти.

FLDCW - замещает текущее значение управляющего регистра CW значением, содержащимся в 16-битном операнде-источнике. Команда обычно используется для установки или изменения режима работы FPU.

Арифметический сопроцессор

Принцип работы сопроцессора:



- регистровый стек сопроцессора организован по принципу кольца;
- процессор и сопроцессор имеют свои раздельные системы команд;
- процессор и сопроцессор являются самостоятельными вычислительными устройствами и работают параллельно;
- если код операции начинается с 11011, то выбранная команда попадает в сопроцессор.

Арифметический сопроцессор

FLD (FLoaD): загрузка значения из памяти или регистра в верхний регистр *ST(0)*.

При выполнении *FLD* значение из исходного источника (память или регистр) загружается в верхний регистр *ST(0)*. Все остальные значения в стеке сдвигаются вниз, так что значение, которое было в *ST(0)* перед выполнением *FLD*, перемещается в *ST(1)*, значение из *ST(1)* перемещается в *ST(2)*, и так далее.

Некоторые команды передачи данных вещественного типа (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) ↔ ST(i)	Обмен значений ST(0) и ST(i)

Арифметический сопроцессор

Некоторые арифметические команды вещественного типа.

Команда	Операнды	Пояснение	Описание
FADD	dst, src	dst = dst + src;	Сложение вещественное
FSUB	dst, src	dst = dst — src;	Вычитание вещественное
FMUL	dst, src	dst = dst * src;	Умножение вещественное
FDIV	dst, src	dst = dst / src;	Деление вещественное

Команда	Пояснение	Описание
FSQRT	ST(0) = √ST(0)	Вычисление квадратного корня
FABS	ST(0) = |ST(0)|	Вычисление модуля
FRNDINT	ST(0)=(ST(0))	Округление ST(0)

Арифметический сопроцессор

Пример команды сопроцессора

Сложение вещественных чисел:

FADD приемник, источник

Команда складывает источник и приемник и помещает результат в приемник.

Команда имеет следующие формы:
FADD источник — источником является 32- или 64-битная переменная, содержащая вещественное число, а приемником ST(0);
FADD ST(0),ST(n), FADD ST(n),ST(0) — источником и приемником являются регистры FPU;
FADD — без операндов эквивалентна *FADD ST(0), ST(1)*

Арифметический сопроцессор

Некоторые команды трансцендентных функций:

Команда	Пояснение	Описание
FSIN	ST(0) = sin(ST(0))	Вычисление синуса
FCOS	ST(0) = cos(ST(0))	Вычисление косинуса
FPTAN	ST(1) = tg(ST(0)); TOP – = 1; ST(0) = 1.0;	Вычисление тангенса
FPATAN	ST(1) = atan(ST(1) / ST(0)); TOP += 1;	Вычисление арктангенса
F2XM1	$ST(0) = 2^{ST(0)} - 1$;	Вычисление выражения $y = 2^x - 1$
FYL2X	x = ST(0); y = ST(1); TOP += 1; $ST(0) = y * \log_2 x$;	Вычисление выражения $y * \log_2 x$
FYL2XP1	x = ST(0); y = ST(1); TOP += 1; $ST(0) = y * \log_2(x + 1)$;	Вычисление выражения $y * \log_2 (x+1)$

Арифметический сопроцессор

Команды управления сопроцессором:

FDECSTP / FINCSTP – сдвиг стека;
FFREE – освобождение регистра;
FCLEX – сбросить статус;
FSTSW / FSTCW – считать статус/управление;
FLDCW – записать управление;
FNOP – пустая операция;
FSTENV – сохранить состояние (кроме данных);
FLDENV – восстановить состояние (кроме данных);
FSAVE – сохранить состояние полностью и сбросить;
FRSTOR – восстановить состояние полностью;
FWAIT / WAIT – задержать ЦП.

В современных процессорах x86-64 архитектуры FPU заменен на SIMD-расширение ***SSE*** (*Streaming SIMD Extensions*) и ***AVX*** (*Advanced Vector Extensions*). SIMD-регистры также могут использоваться для выполнения операций с плавающей запятой, но их работа и организация отличаются от стека FPU

Арифметический сопроцессор

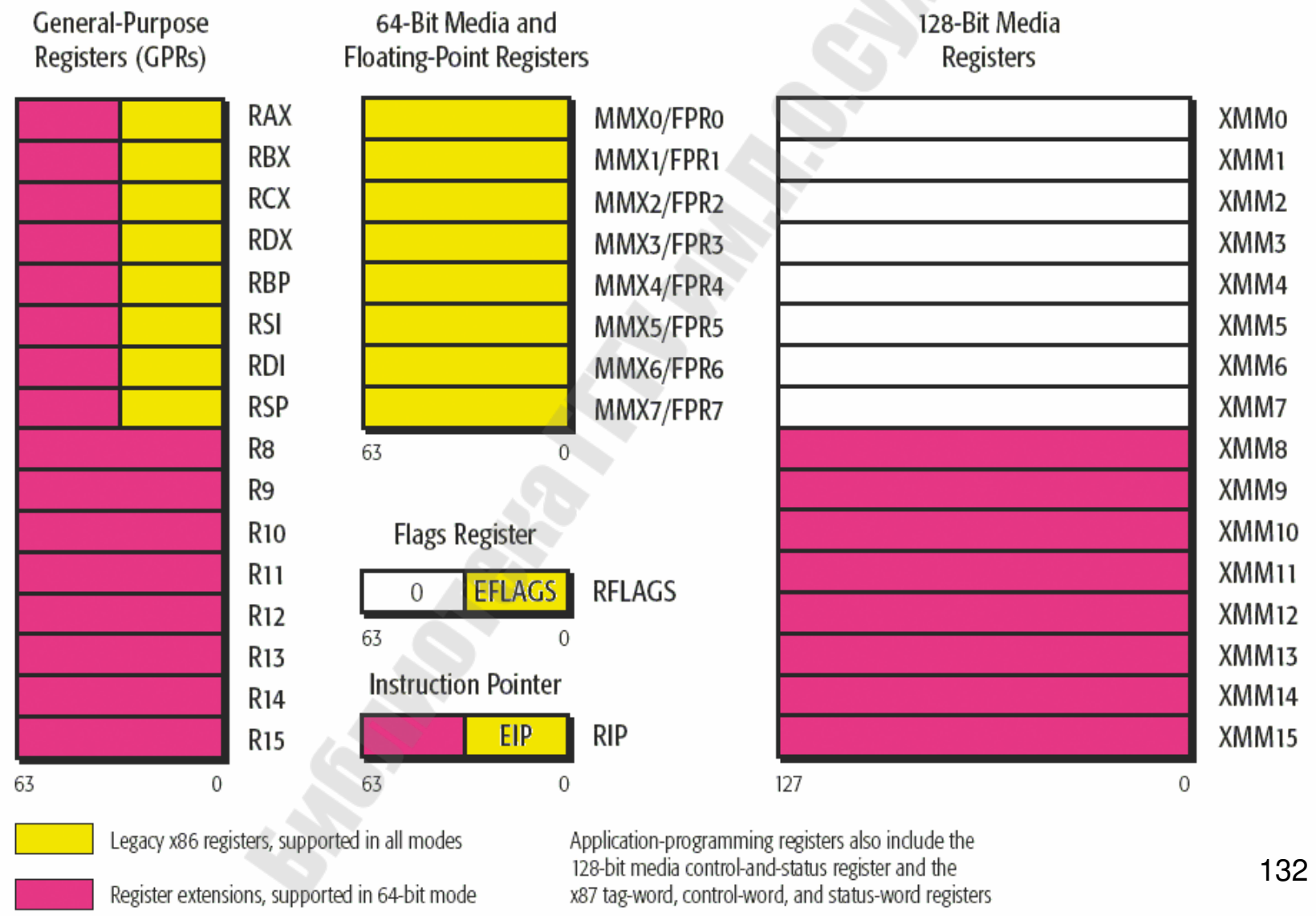
SIMD (*Single Instruction, Multiple Data*) - это технология, которая позволяет одной инструкции выполнять операции над несколькими элементами данных одновременно. *SIMD*-расширения используются для ускорения выполнения параллельных вычислений, таких как операции с плавающей запятой, обработка изображений, аудио и видео, компьютерная графика и другие задачи.

В архитектуре x86 процессоры поддерживают несколько наборов SIMD-расширений, таких как ***MMX*** (*Multimedia Extensions*), ***SSE*** (*Streaming SIMD Extensions*), ***AVX*** (*Advanced Vector Extensions*) и другие. Каждое расширение предоставляет **новые регистры** SIMD **и набор инструкций** для выполнения параллельных операций.

SSE расширение предоставляет регистры ***XMM***, каждый из которых может содержать 128 бит и разделен на подэлементы, в то время как *AVX* расширение предоставляет регистры ***YMM*** и ***ZMM***, которые имеют больший размер и могут содержать больше элементов данных.

Арифметический сопроцессор

Регистры процессора архитектуры x86-64 (AMD64, Intel64)



Тема 6
Система прерываний

Система прерываний

Прерывания (*interrupts*) + исключения (*exceptions*) = события (*events*)
Источники:
- внешний сигнал по входам *INTR* или *NMI*;
- исключение при выполнении команды;
- команда прерывания *INT n.*

Аппаратные прерывания – вызваны электрическими сигналами на входах МП.
Программные прерывания – вызваны командой *INT n*, где *n* – номер (вектор) прерывания. Не следует программно вызывать исключения, помещающие в стек код ошибки.

По прерыванию (исключению) МП сохраняет в стек *(E)FLAGS* и *CS:(E)IP*, а также код ошибки при исключениях в Р-режиме.

Обработчик прерывания (исключения) (*interrupt handler*) заканчивается командой *IRET*, по которой из стека восстанавливаются *CS:(E)IP* и *(E)FLAGS.*

Приоритет прерываний:
1. исключения;
2. немаскируемые прерывания;
3. маскируемые прерывания;
4. программные прерывания

Система прерываний

Адрес обработчика прерывания (подпрограммы) определяется по таблице векторов прерываний.

Вектор прерывания – 8-битный (0 - 255) номер указателя адреса обработчика:

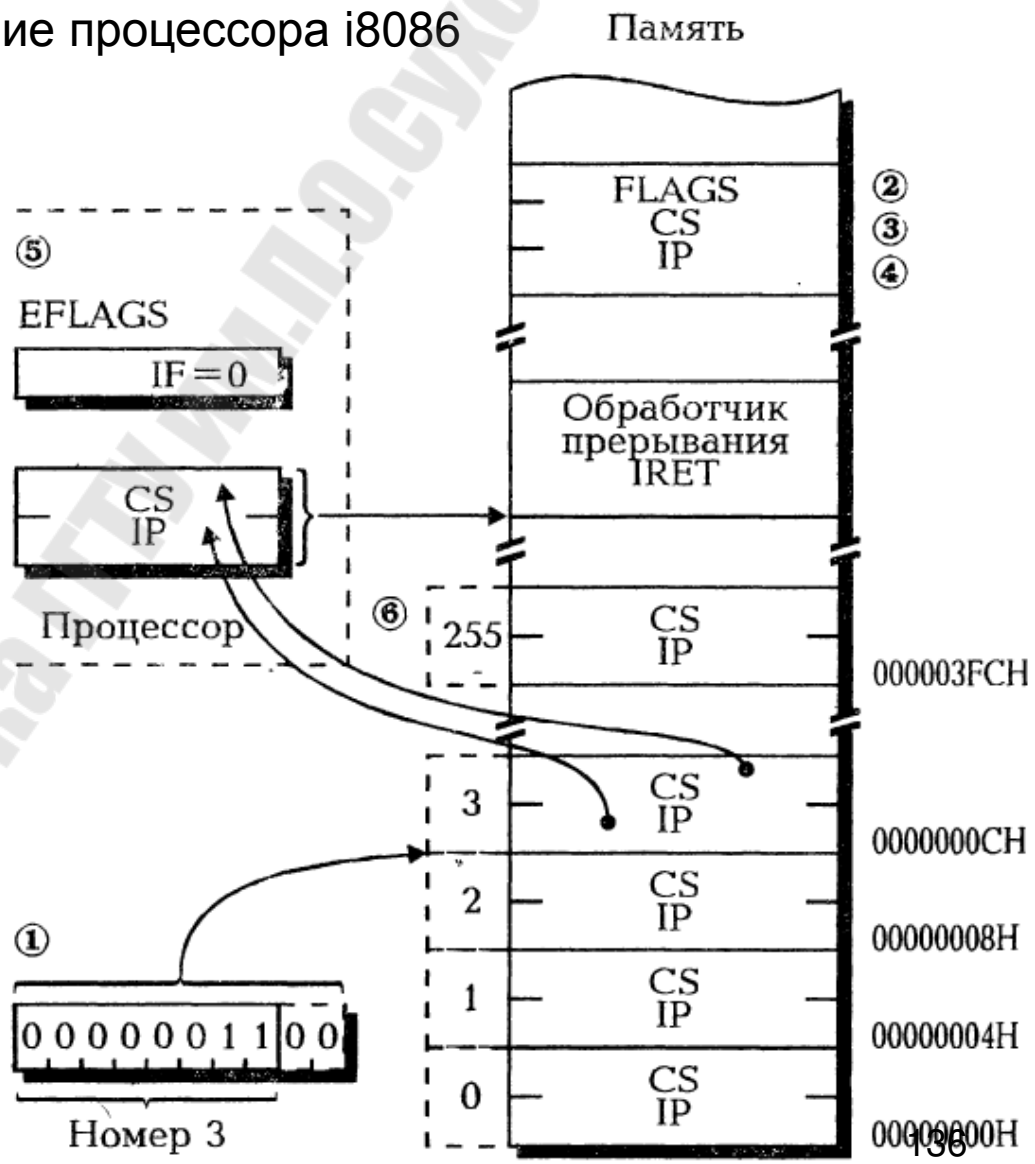
- задается внешним контроллером прерываний (маскируемые);
- имеет фиксированный адрес (2) (немаскируемые);
- передается в команде *INT n* (программные).

В R-режиме таблица располагается с адреса 0000 0000h и содержит 256 32-битных адресов (сегмент:смещение). Начало таблицы можно изменить командой *LIDT* (Load Interrupt Descriptor Table).

Система прерываний

Реакция на маскируемое прерывание процессора i8086 (вход *INTR* = 1; флаг *IF* = 1):

1 – получение вектора от i8959;
2, 3, 4 – помещает в стек FLAGS, CS, IP;
5 – сброс флага IF = запрет прерываний;
6 – чтение адреса обработчика;
7 – выполняется обработчик



Система прерываний

Маскируемые прерывания:

- вызываются установкой входа *INTR* (INTerrupt Request) МП в «1» при *IF* = 1;
- сохраняется в стек *(E)FLAGS* и *CS:(E)IP*, сбрасывается флаг *IF*, формируется сигнал *INTA* (INTerrupt Acknowledge);
- МП получает номер вектора прерывания от *PIC* (Programmable Interrupt Controller);
- обработка текущего прерывания может прерываться обработкой немаскируемого или маскируемого (при установке в обработчике флага IF) прерывания;

Немаскируемые прерывания:

- вызываются установкой входа *NMI* (Non Maskable Interrupt) МП в «1» не зависимо от флага *IF*;
- сохраняется в стек *(E)FLAGS* и *CS:(E)IP*, сбрасывается флаг *IF*, формируется сигнал *INTA* (INTerrupt Acknowledge);
- обработка прерывания с вектором 2;
- обработчик не может быть прерван до выполнения команды *IRET*;

Особые случаи (исключения):

- адреса векторов 0 – 31;
- не зависят от флага IF;
- в стек сохраняется адрес команды вызвавшей исключение – т. о. обработчик может исправить ситуацию и повторить команду

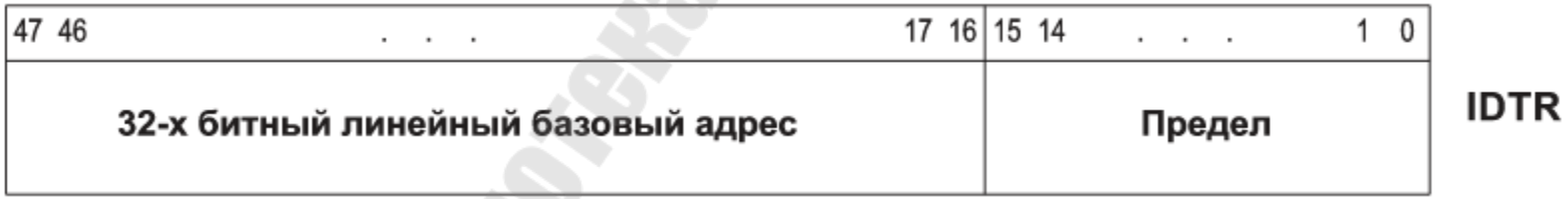
Система прерываний

В Р-режиме в ОЗУ хранится таблица дескрипторов прерываний – *Interrupt Descriptor Table* (IDT) – массив 64-битных чисел – дескрипторов шлюзов.

Дескриптор шлюза содержит селектор сегмента кода и адрес (смещение) обработчика в этом сегменте.

Перед тем как разрешить прерывания, необходимо адрес и размер таблицы *IDT* загрузить в регистр *IDTR* при помощи команды *LIDT*.

Регистр таблицы дескрипторов прерываний *IDTR* содержит 32-битный (24-битный для i80286) базовый адрес и 16-битный предел таблицы дескрипторов прерываний (*IDT*) (в R-режиме задает положение таблицы векторов прерываний).

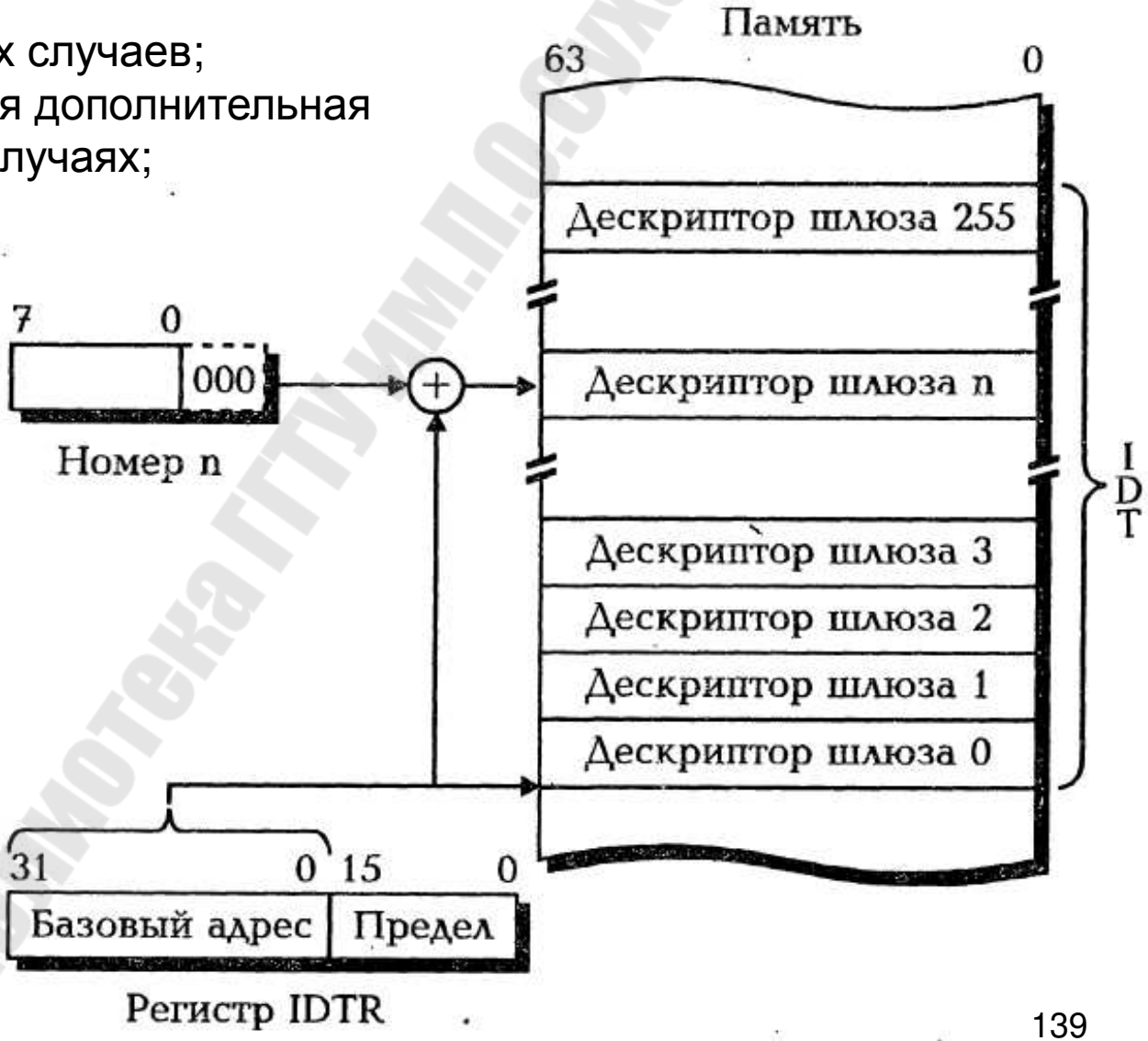


В качестве операнда в команде *LIDT* указывается адрес области в формате 16+32. Младшее слово области — размер *IDT*, двойное слово по старшему адресу — значение базового адреса начала этой таблицы. Данные два компонента должны быть сформированы в памяти заранее.

Система прерываний

Реакция на маскируемое прерывание процессора i486

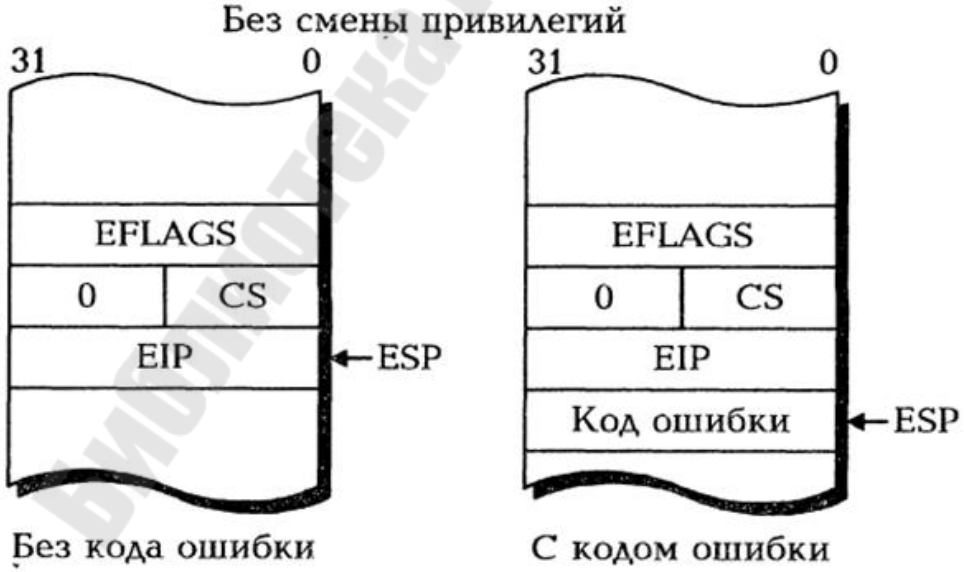
- увеличено число особых случаев;
- в обработчик передается дополнительная информация об особых случаях;



Система прерываний

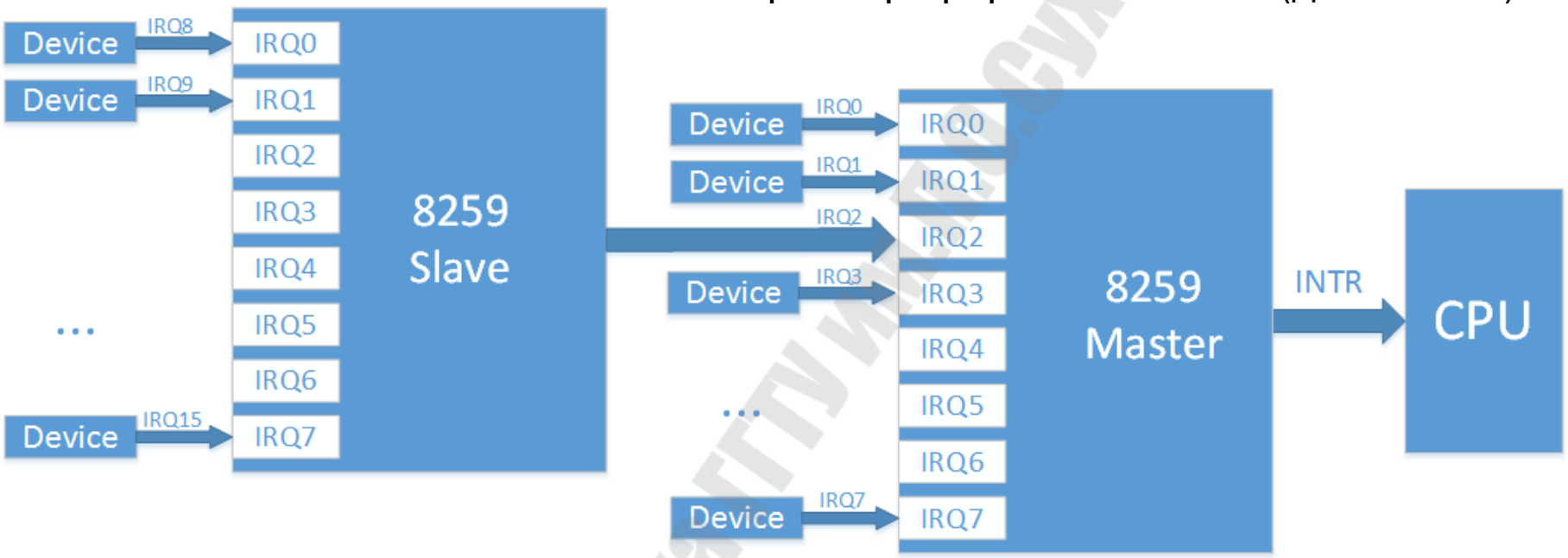
Структура дескрипторов шлюзов:

Стек обработчика особого случая



Система прерываний

Контроллер прерываний i8259 (до Pentium)

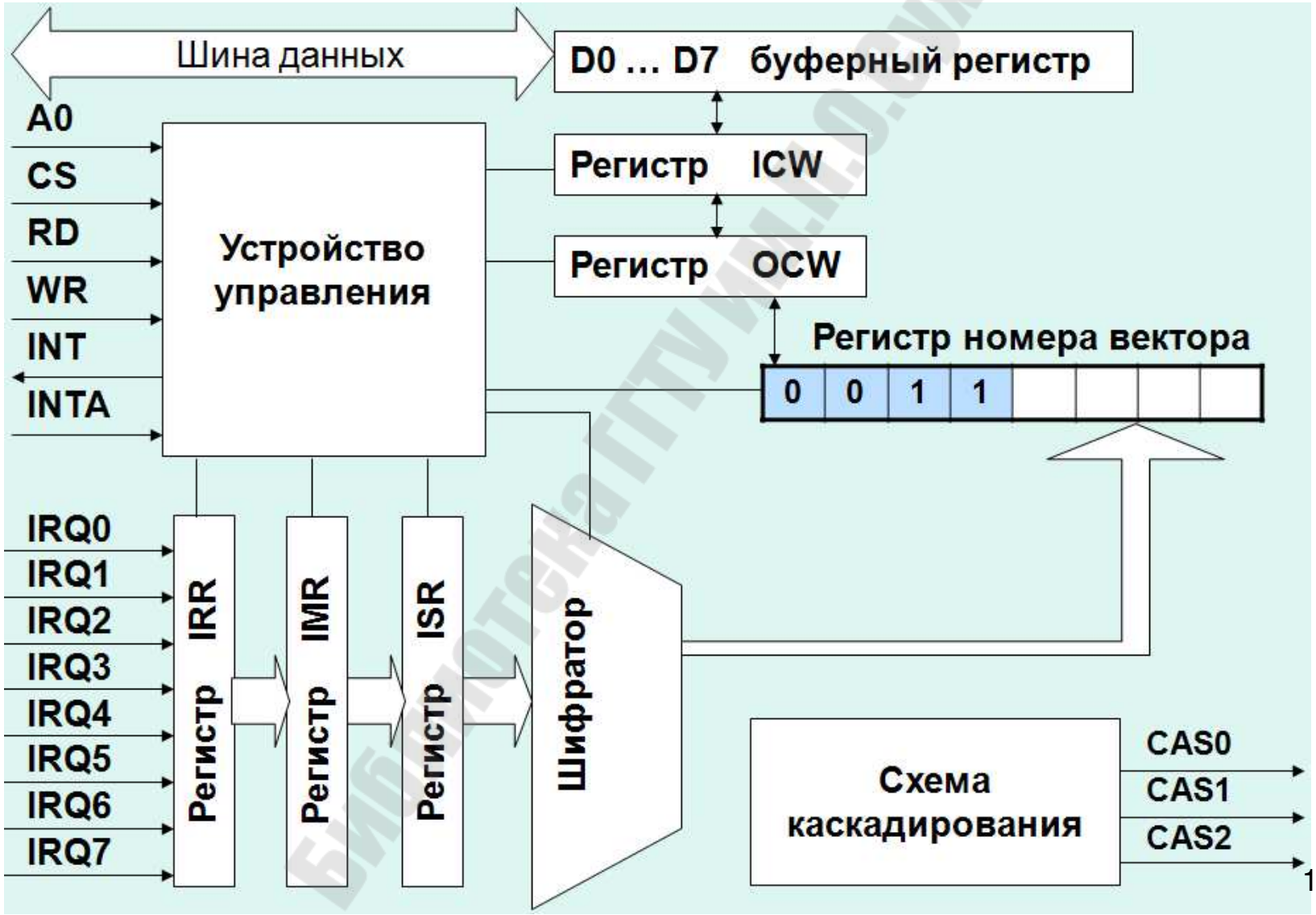


IRQ 0 — system timer
IRQ 1 — keyboard controller
IRQ 2 — cascade (прерывание от slave)
IRQ 3 — serial port COM2
IRQ 4 — serial port COM1
IRQ 5 — parallel port 2 and 3 or sound card
IRQ 6 — floppy controller
IRQ 7 — parallel port 1

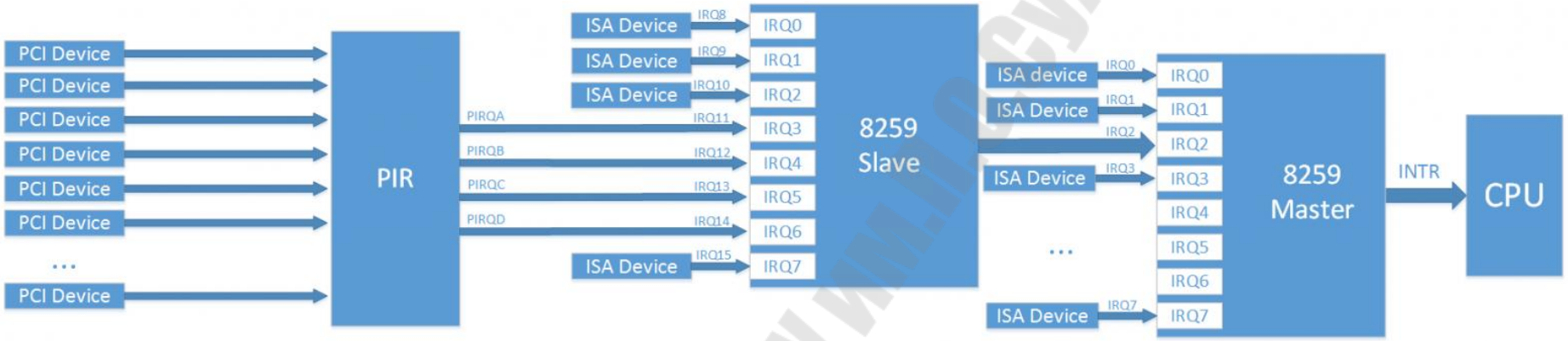
IRQ 8 — RTC timer
IRQ 9 — ACPI
IRQ 10 — open/SCSI/NIC
IRQ 11 — open/SCSI/NIC
IRQ 12 — mouse controller
IRQ 13 — math co-processor
IRQ 14 — ATA channel 1
IRQ 15 — ATA channel 2

Система прерываний

Контроллер прерываний i8259A



Система прерываний

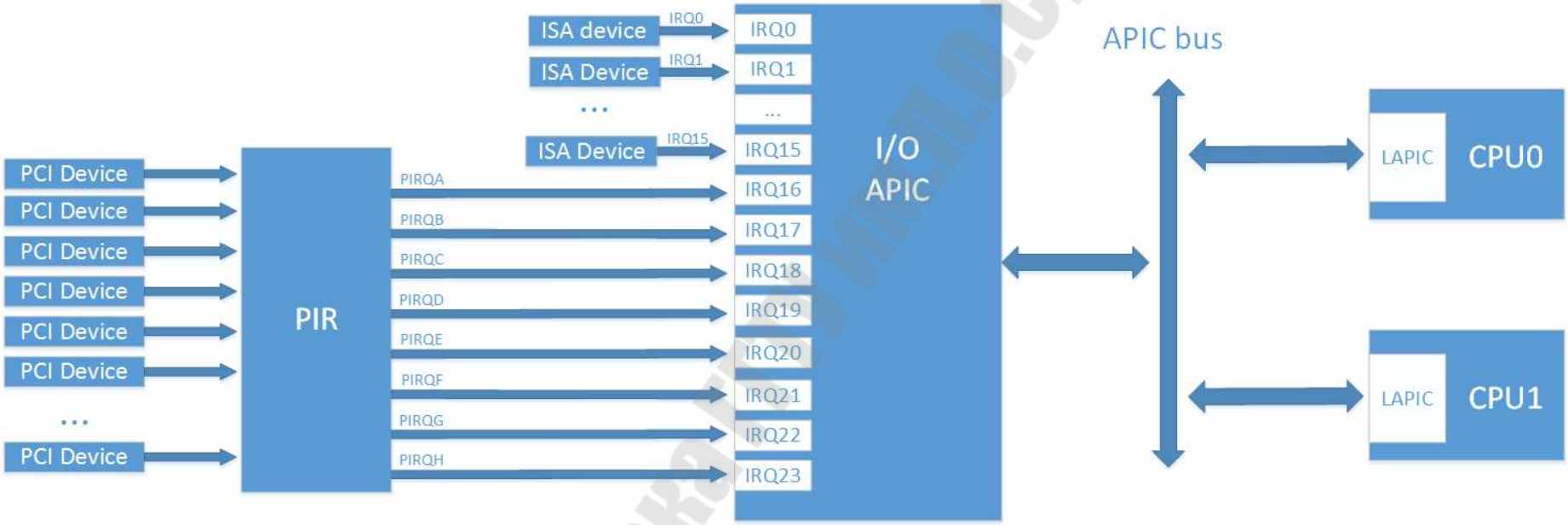


Допустим, свободны 4 линии прерываний на PIC контроллере, а PCI устройств 20 штук. Объединяем прерывания по 5 устройств на линию *PIRQx* и подключаем линии *PIRQx* к контроллеру.

При возникновении прерывания на линии *PIRQx* процессору придётся опросить все устройства подключённые к данной линии, чтобы понять от кого именно пришло прерывание.

Система прерываний

В многопроцессорных системах начиная с Pentium II применен усовершенствованный контроллер прерываний (Advanced PIC)



82489DX – I/O APIC + LAPIC

82093AA – I/O APIC

Для поддержки совместимости со старыми системами, прерывания 0~15 отвели под старые прерывания ISA. А прерывания от PCI устройств стали выводить на линии IRQ 16-23.

Система прерываний

Определение обработчика прерывания с именем *MyInterruptHandler* в *masm32*

```
MyInterruptHandler PROC
   ; инструкции по обработке прерывания
   ret
MyInterruptHandler ENDP
```

Регистрация обработчика прерывания *MyInterruptHandler*

```
include windows.inc
include kernel32.inc

.data
interruptNumber     DD      0                   ; номер прерывания

.code
main PROC
; Регистрация обработчика прерывания
    lea eax, MyInterruptHandler         ; загрузка адреса обработчика в EAX
    push eax                            ; передача адреса в функцию через стек
    push interruptNumber                ; передача номера в функцию через стек
    call SetVectoredInterruptHandler    ; вызов функции регистрации обработчика
```

Тема 7
Система ввода/вывода

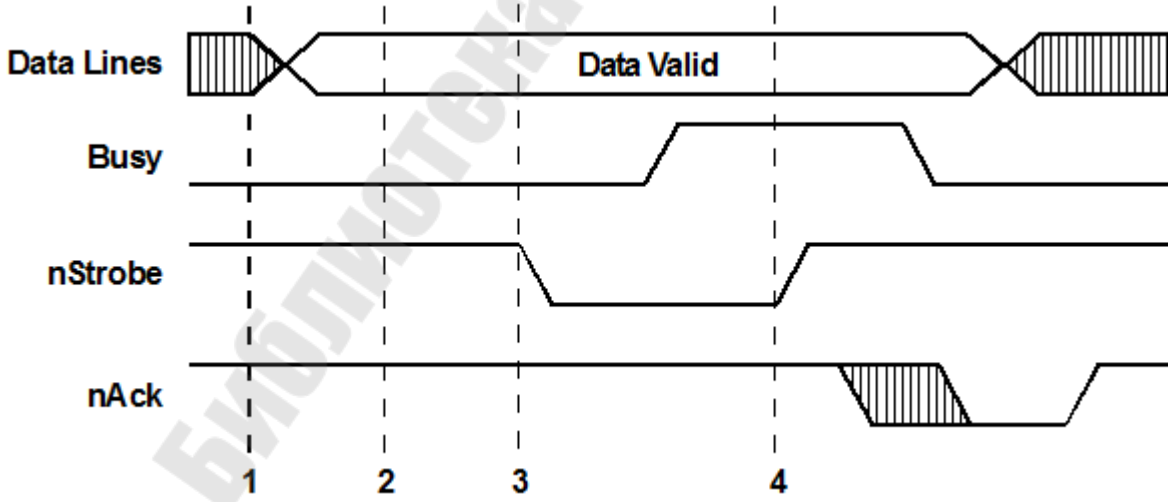
Система ввода/вывода

Режимы ввода/вывода:

1. программно управляемый;
2. по прерываниям;
3. прямой доступ к памяти

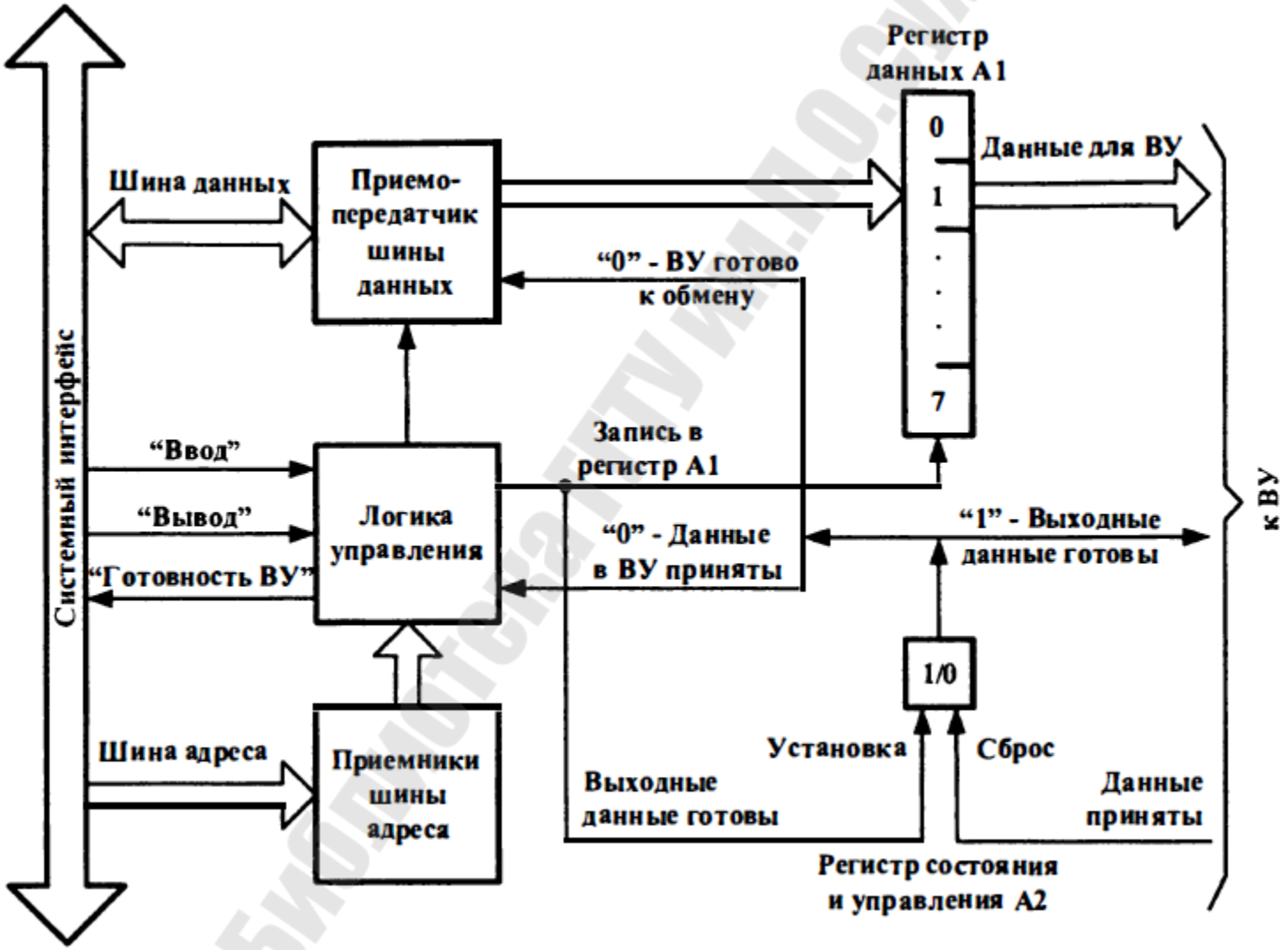
1. Программно управляемый – все действия по вводу/выводу реализуются командами программы. Для ввода/вывода используются команды *IN* или *OUT*. Такая передача – синхронная.

   Если перед передачей нужно убедиться в готовности ВУ к обмену, передача – асинхронная. Общее состояние устройства характеризуется сигналом готовности READY/BUSY.



Система ввода/вывода

Структура параллельного контроллера вывода:

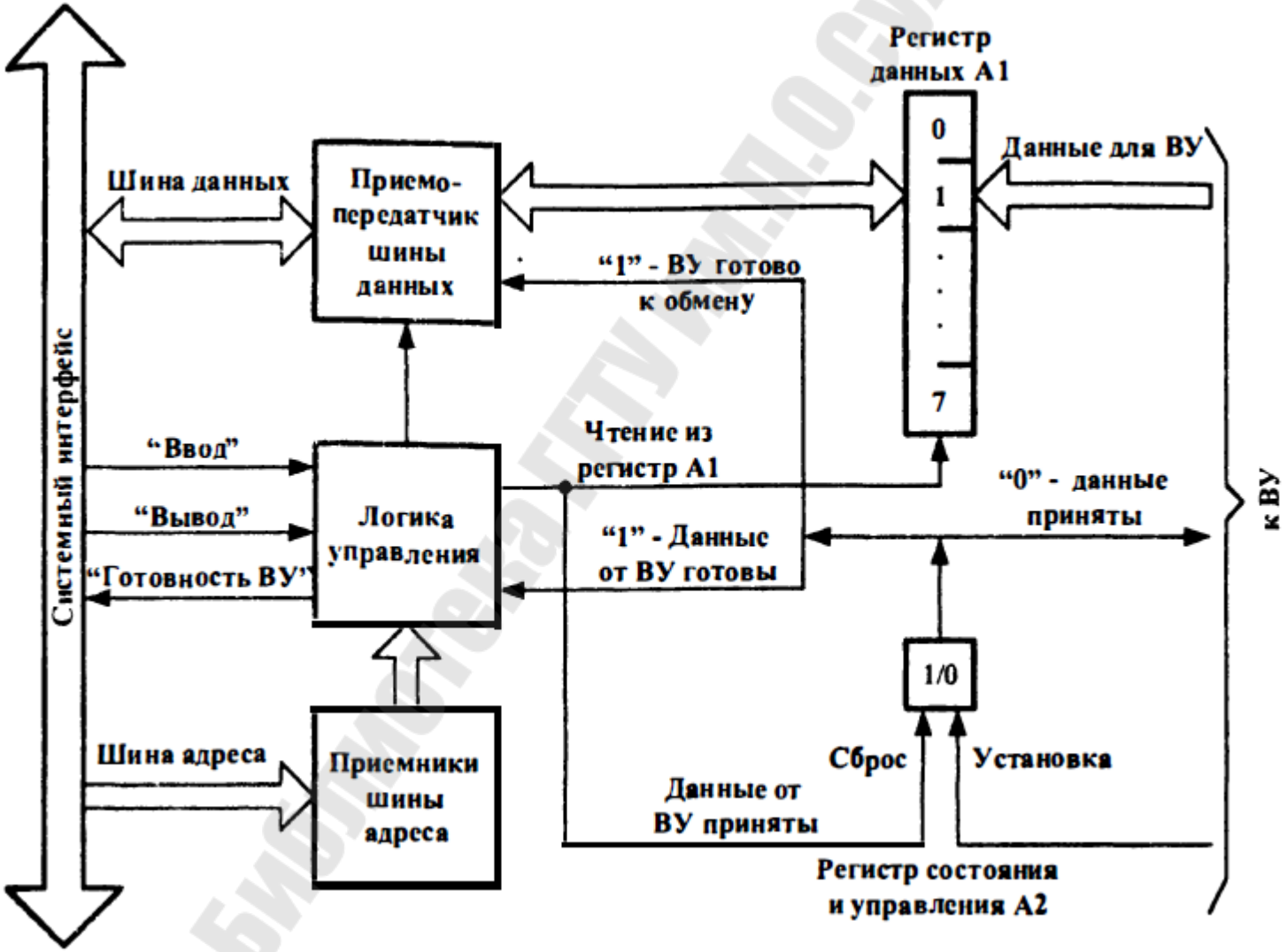


Фрагмент программы передачи данных в асинхронном режиме:

```
        mov dx, A2          ; адрес порта помещаем в DX
m1:
        in al, dx           ; чтение байта из порта с адресом A2
        test al, 1          ; проверка состояния порта
        jnz m1              ; переход на метку M1, если не ноль
        mov al, 64          ; выводимый байт помещаем в AL
        mov dx, A1          ; номер порта A1 в DX
        out dx, al          ; передаем AX в порт A1
```

Система ввода/вывода

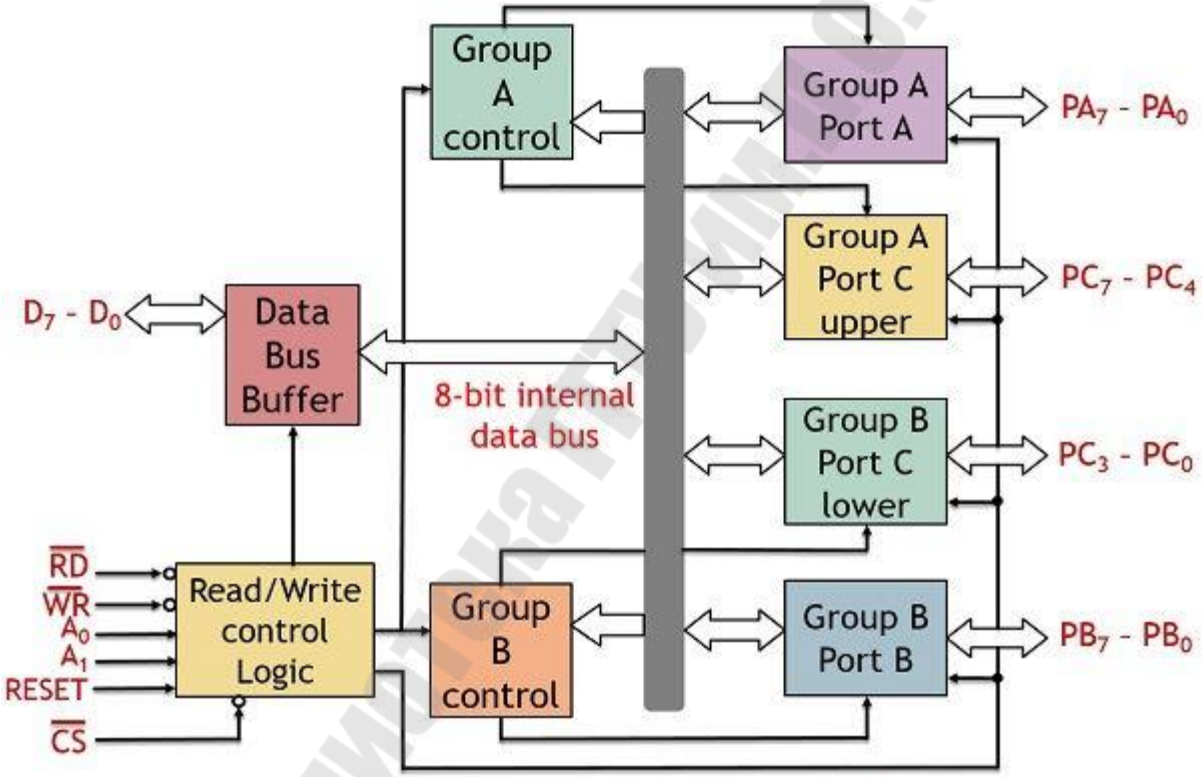
Структура параллельного контроллера ввода:



Фрагмент программы приема данных в асинхронном режиме:

```
        mov dx, A2          ; адрес порта A2 помещаем в DX
m1:
        in al, dx           ; чтение байта из порта с адресом A2
        test al, 1          ; проверка состояния порта
        jz m1               ; переход на метку M1, если не ноль
        mov dx, A1          ; адрес порта A1 в DX
        in al, dx           ; передаем в AX содержимое порта A1
```

Структура параллельного программируемого адаптера i8255:



Последовательная передача данных (протокол USART)

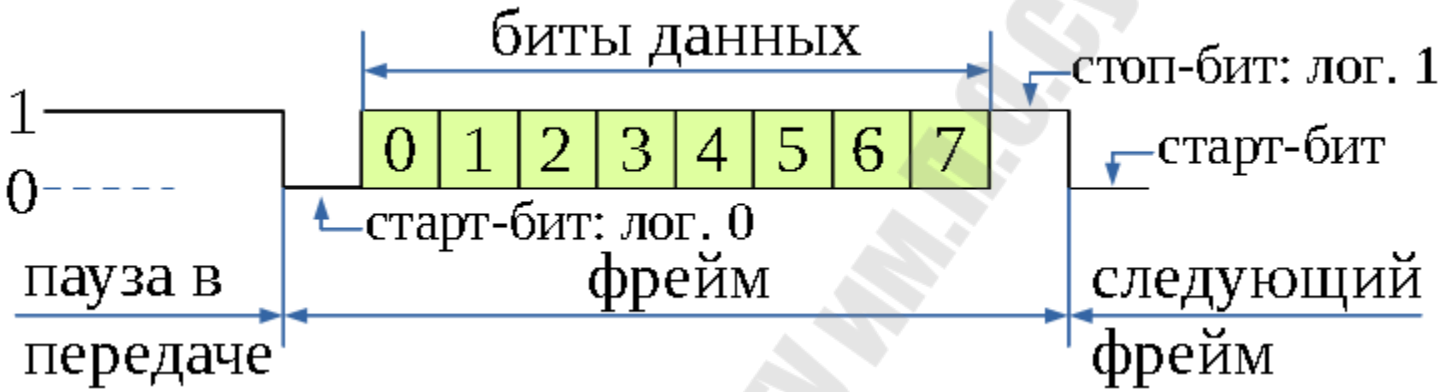
Ряд стандартных скоростей обмена: 1200, 2400, 4800, 9600, 19 200, 38 400, 57 600 и 115 200 бит/с.

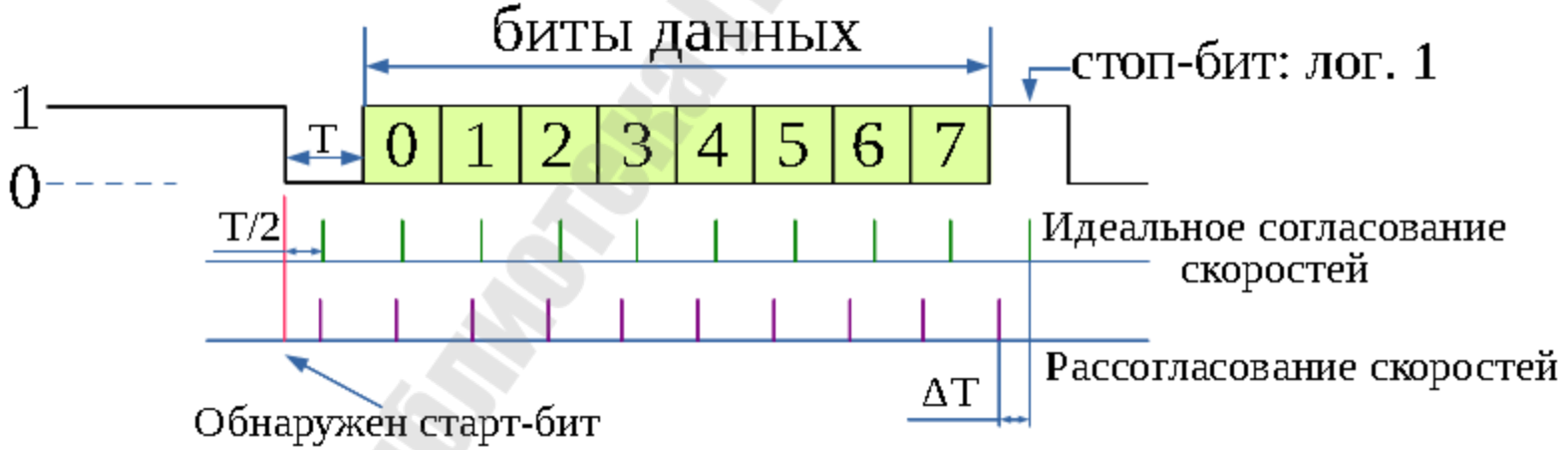
Количество бит данных может составлять 5, 6, 7 или 8.

Количество стоп-бит может быть 1, 1,5 или 2.

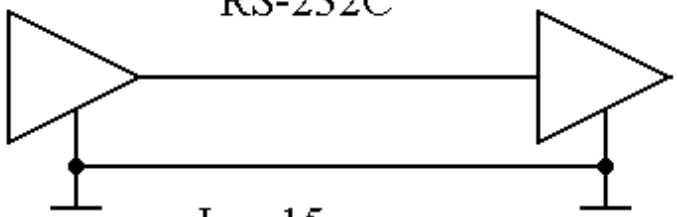
Примеры: последовательный коммуникационный порт (COM-порт), последовательные шины USB и FireWire, PCI Express, интерфейсы локальных и глобальных сетей.

Последовательная передача данных (протокол USART)



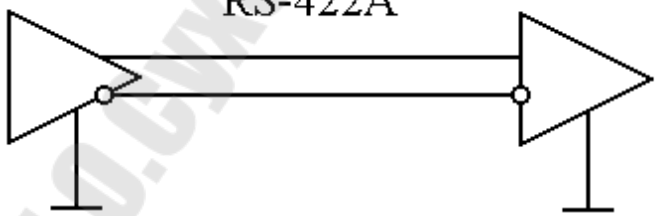


RS-232C



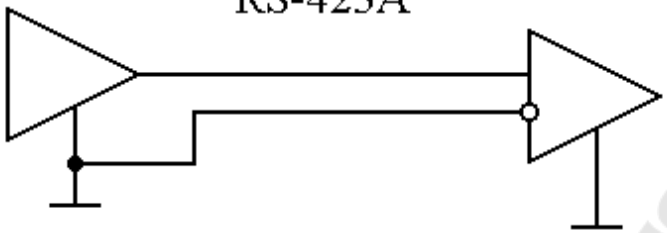
L = 15 м
V = 20 Кбит/с

RS-422A



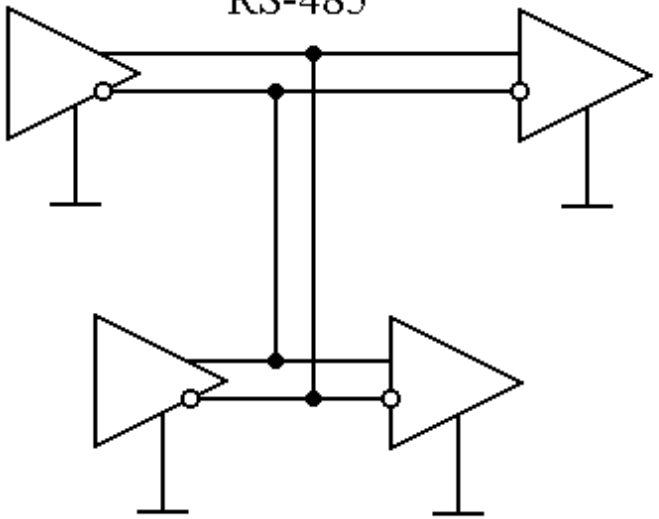
L = 12 м V = 10 Мбит/с
L = 120 м V = 1 Мбит/с
L = 1200 м V = 100 Кбит/с

RS-423A



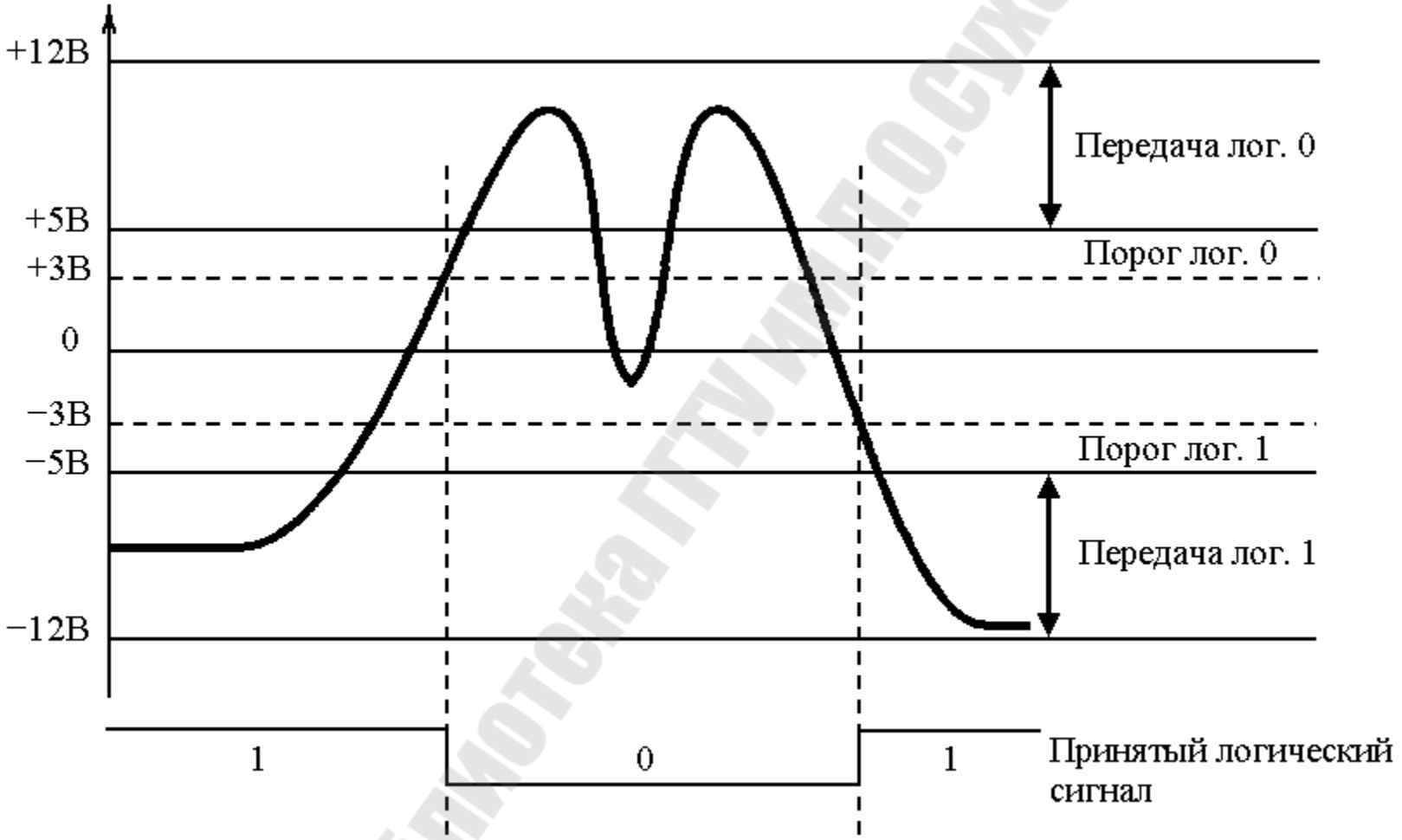
L = 9 м V = 100 Кбит/с
L = 91 м V = 10 Кбит/с
L = 1200 м V = 1 Кбит/с

RS-485



L = 12 м V = 10 Мбит/с
L = 120 м V = 1 Мбит/с
L = 1200 м V = 100 Кбит/с

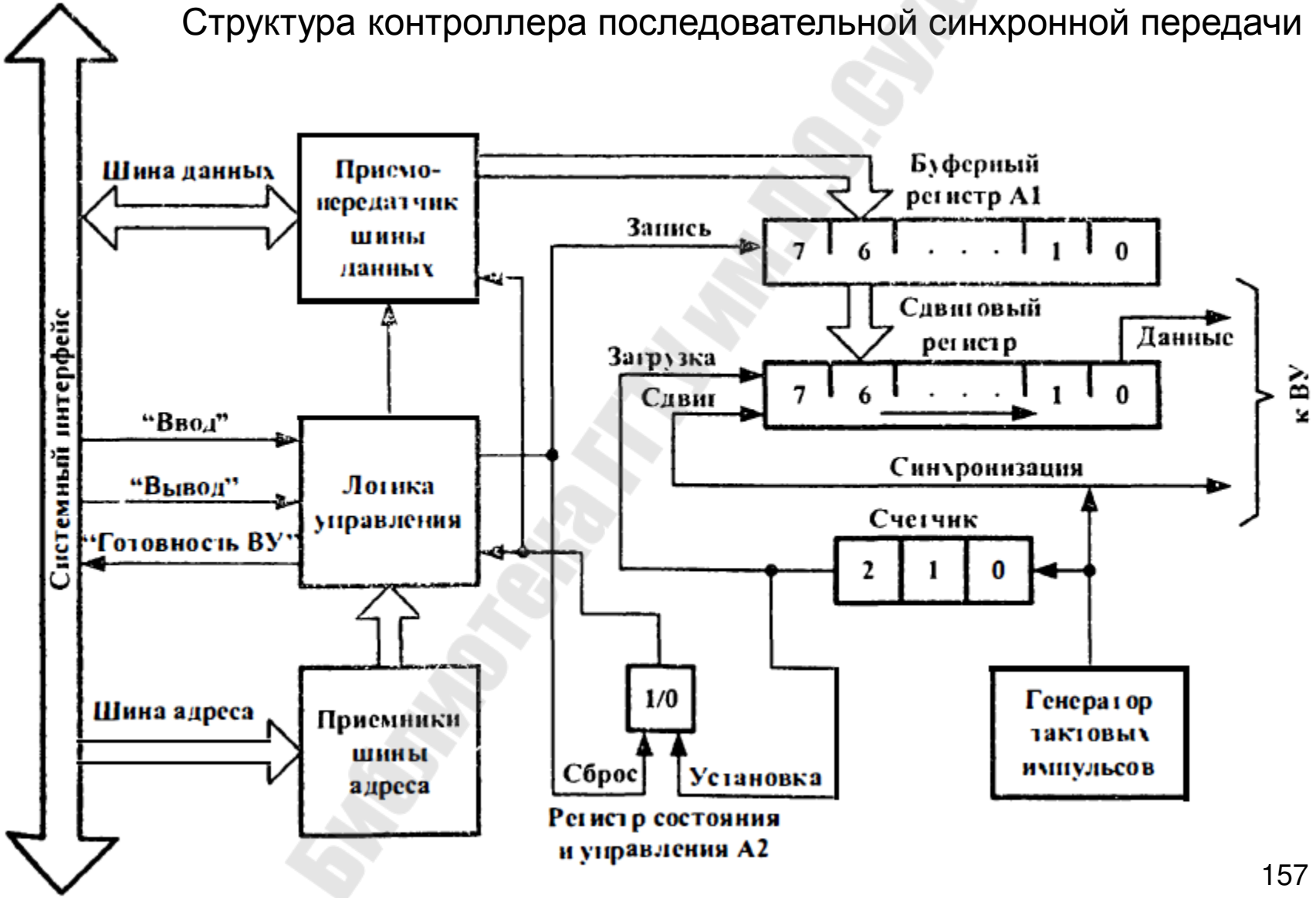
Стандарты последовательного интерфейса



Уровни сигналов RS-232C

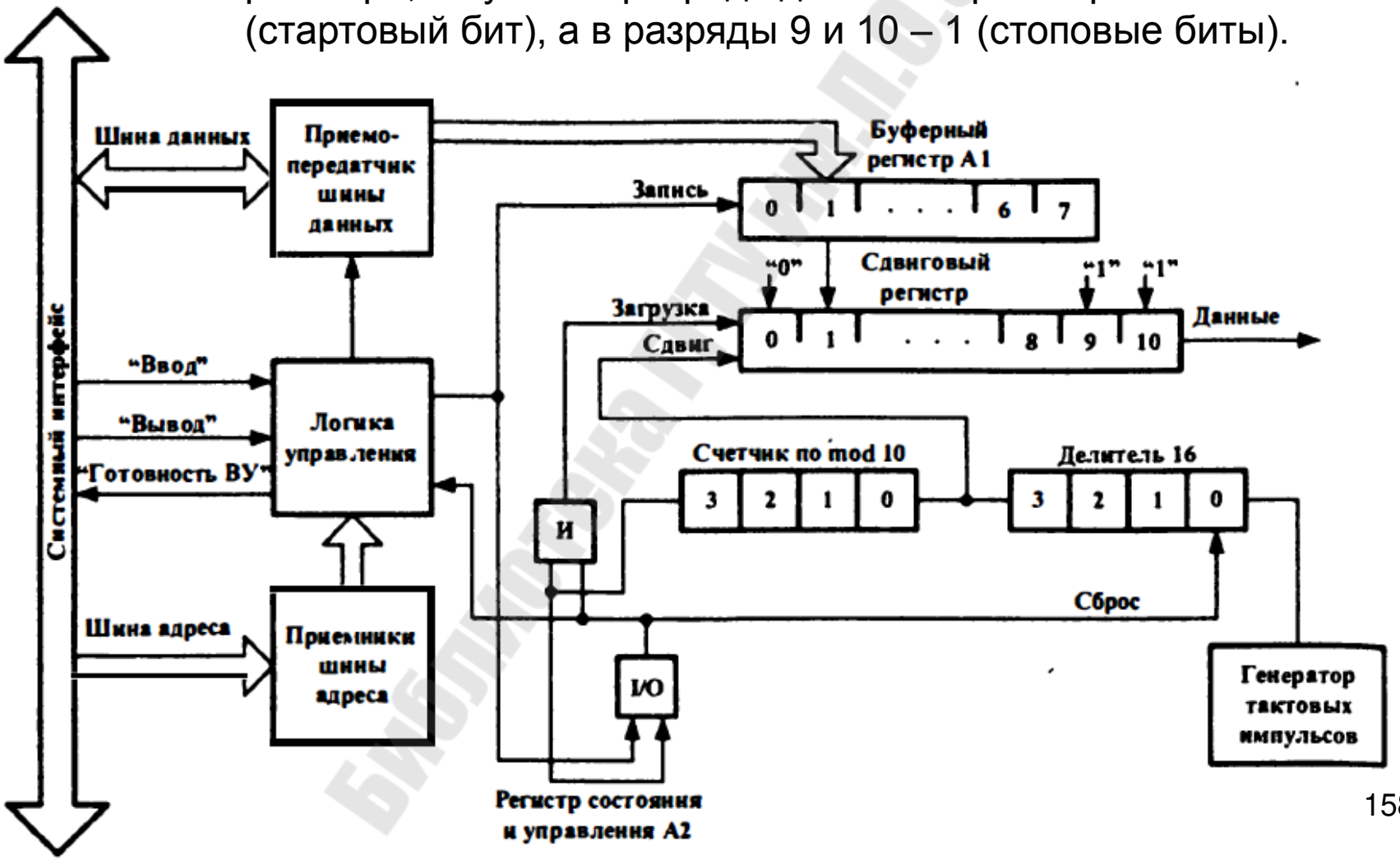
Система ввода/вывода

Структура контроллера последовательной синхронной передачи



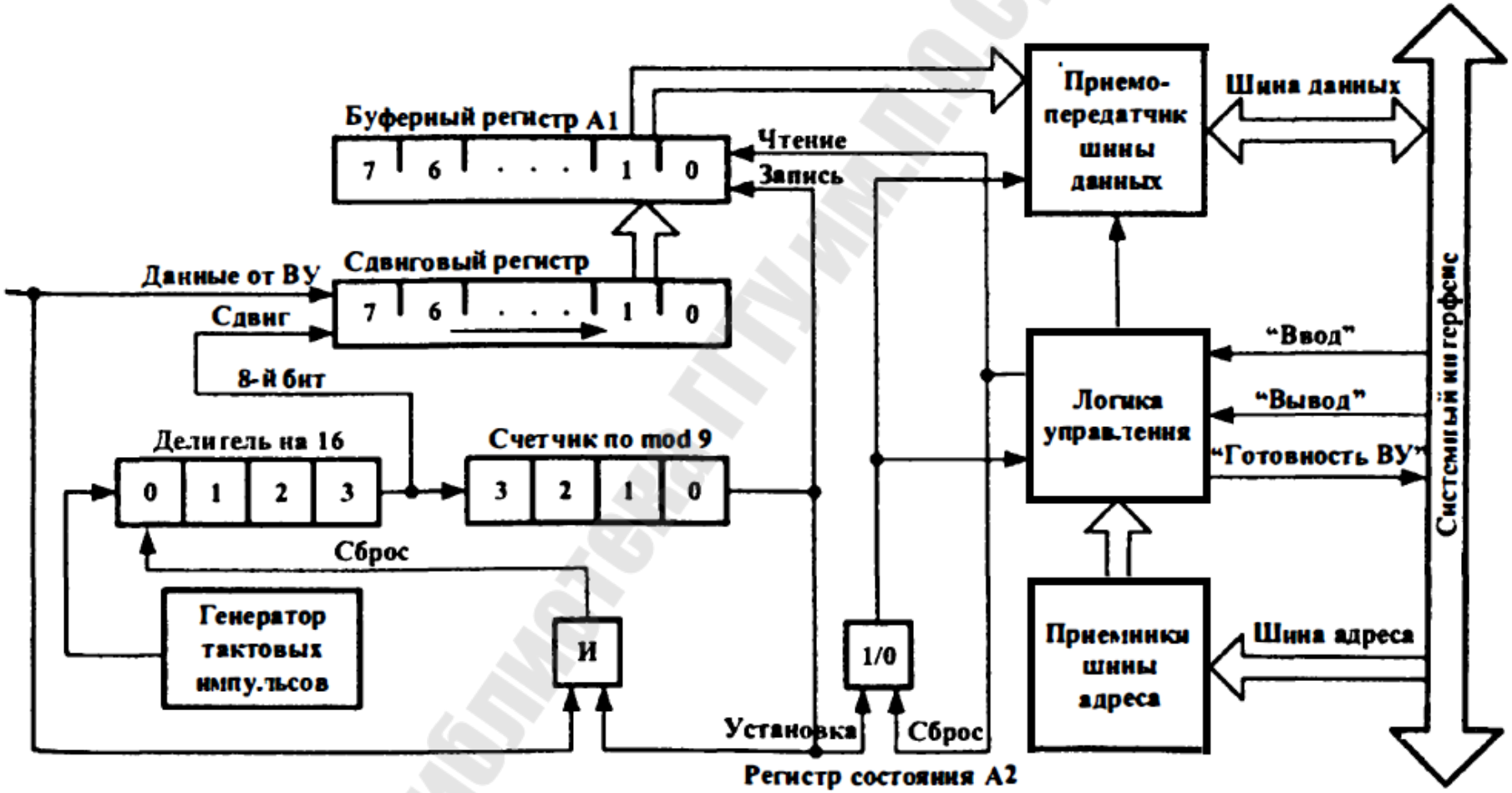
Структура контроллера последовательной асинхронной передачи

Передаваемый байт переписывается в разряды 1. . . 8 сдвигового регистра, в нулевой разряд сдвигового регистра записывается 0 (стартовый бит), а в разряды 9 и 10 – 1 (стоповые биты).

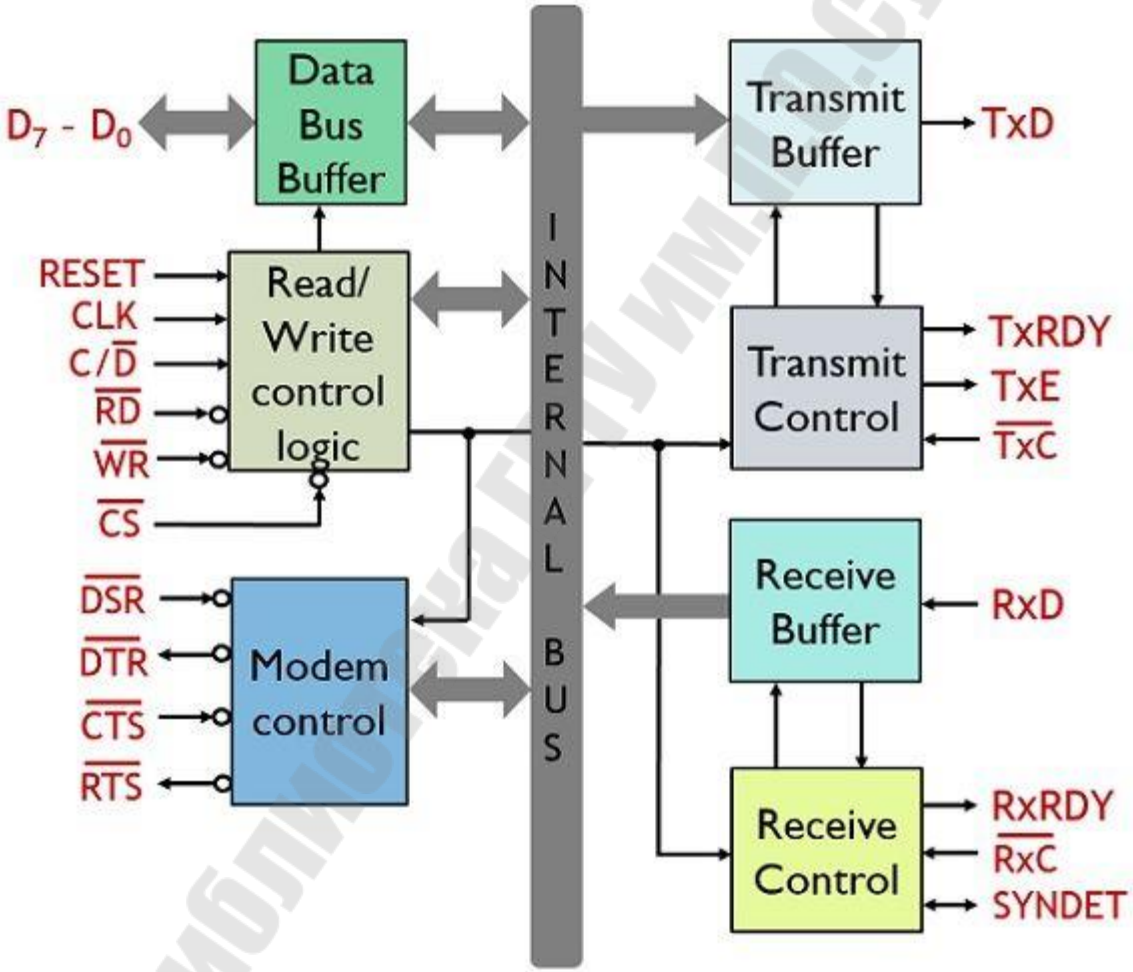


Система ввода/вывода

Структура контроллера последовательного асинхронного приема

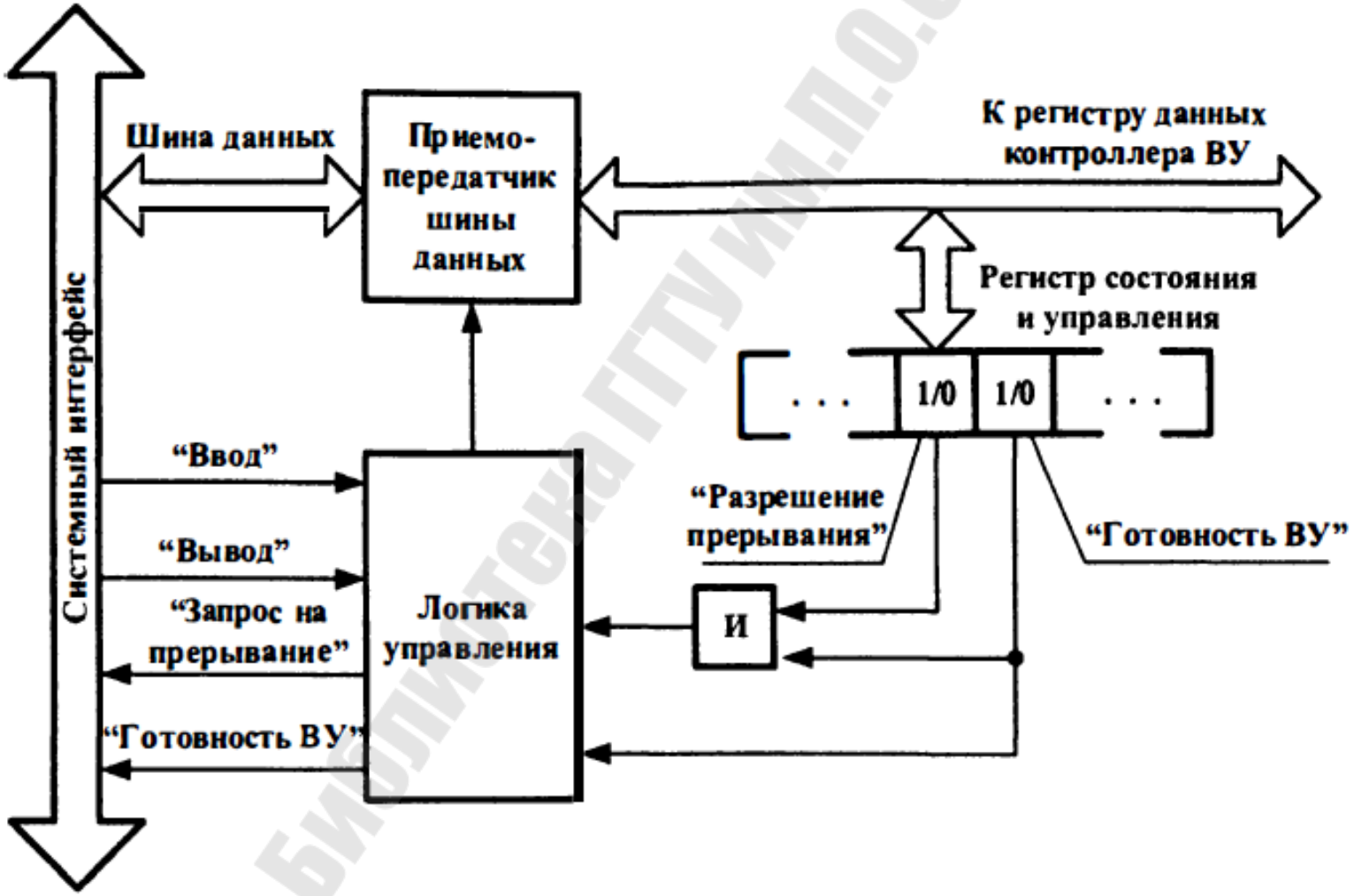


Структура контроллера последовательного связного адаптера i8251:

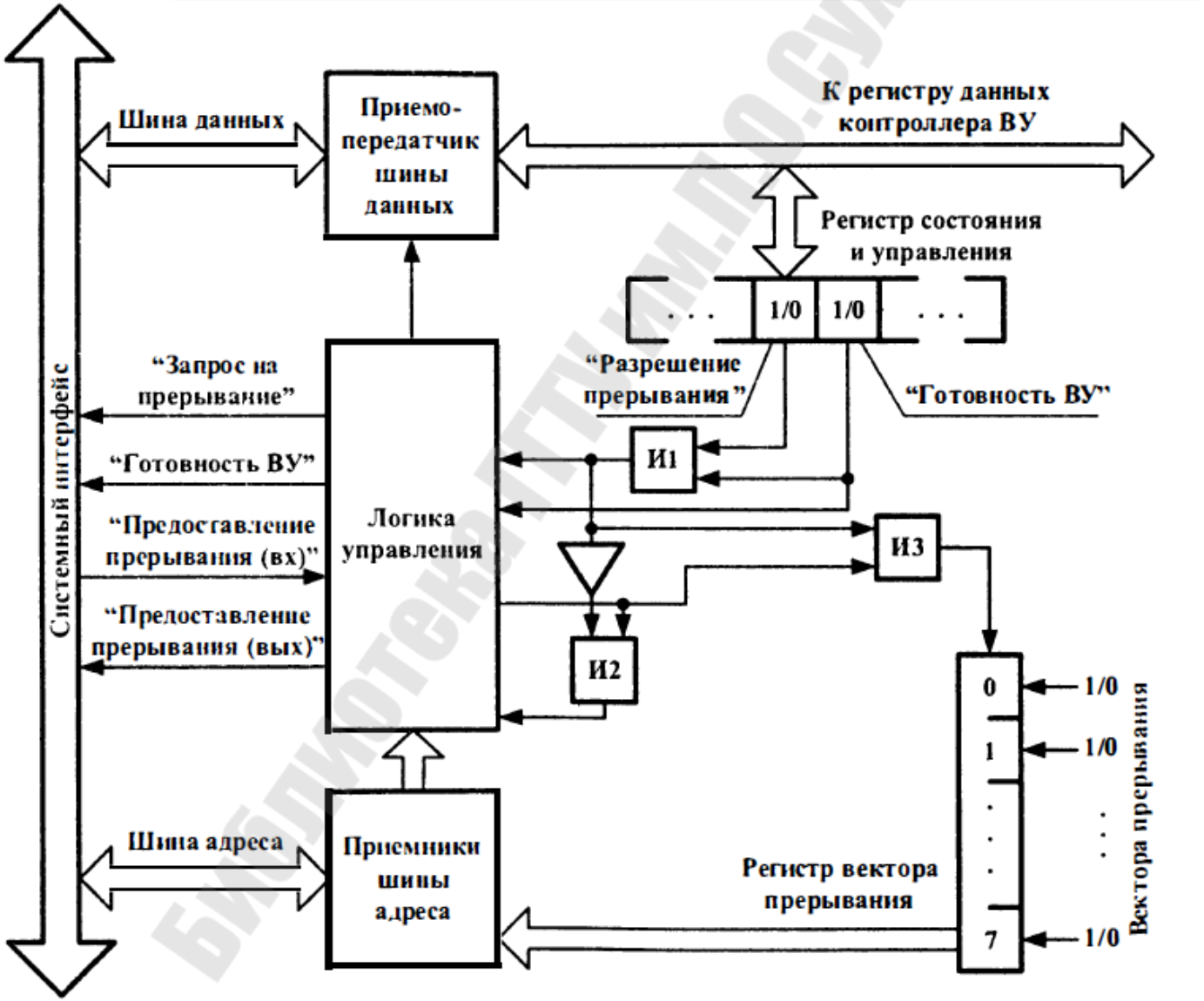


Система ввода/вывода

Структура контроллера внешнего устройства с разрядом «Разрешение прерывания» в регистре состояния

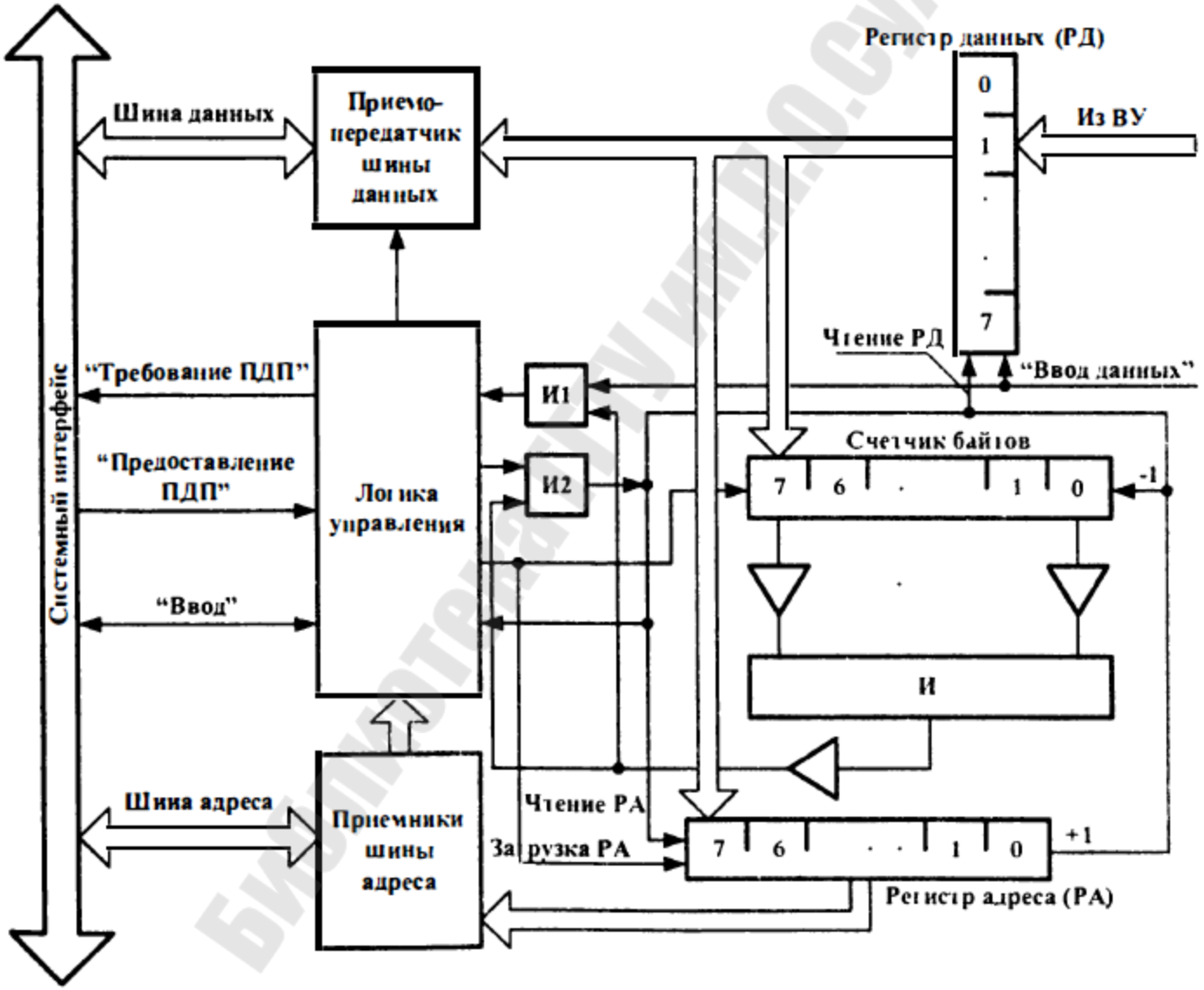


Формирование векторов прерываний в контроллере ВУ

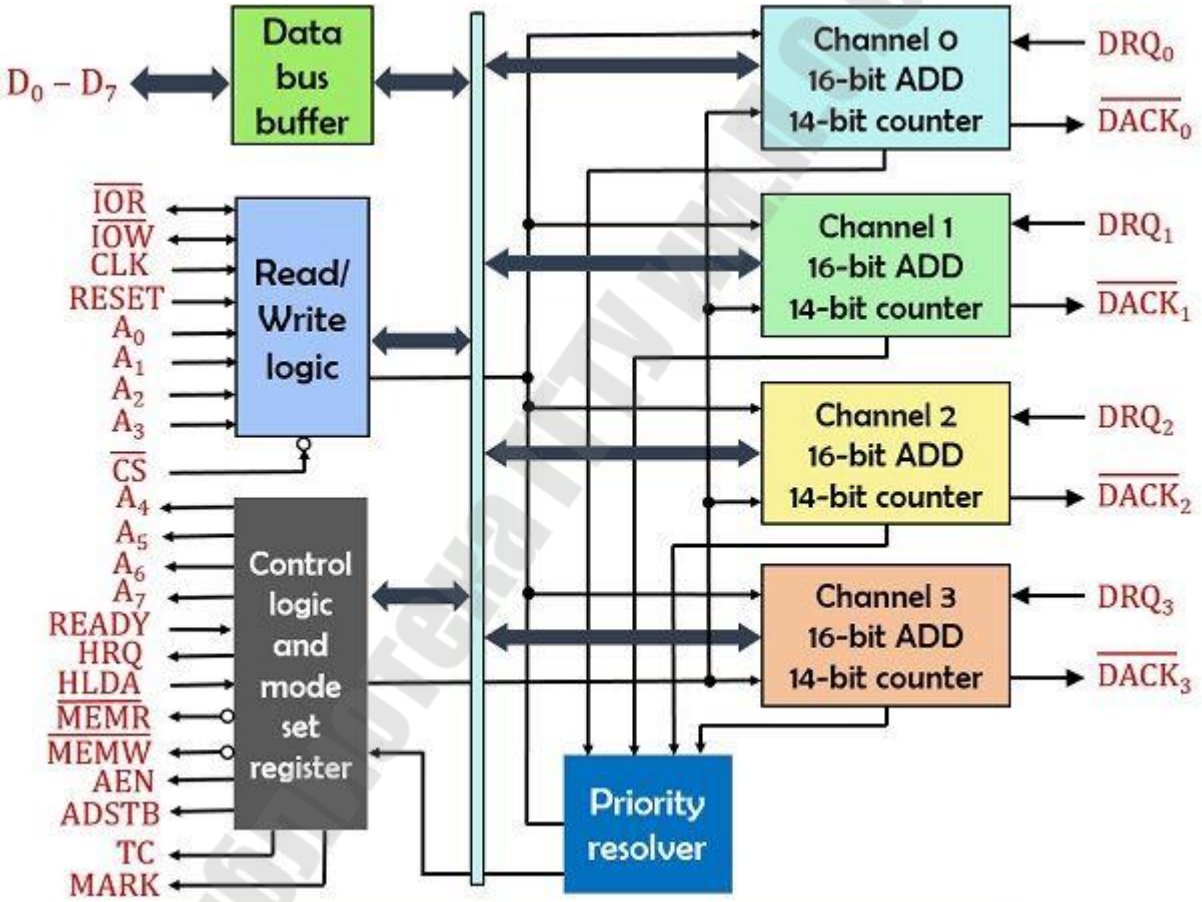


Система ввода/вывода

Структура контроллера ПДП для ввода данных из ВУ в режиме «захват цикла»

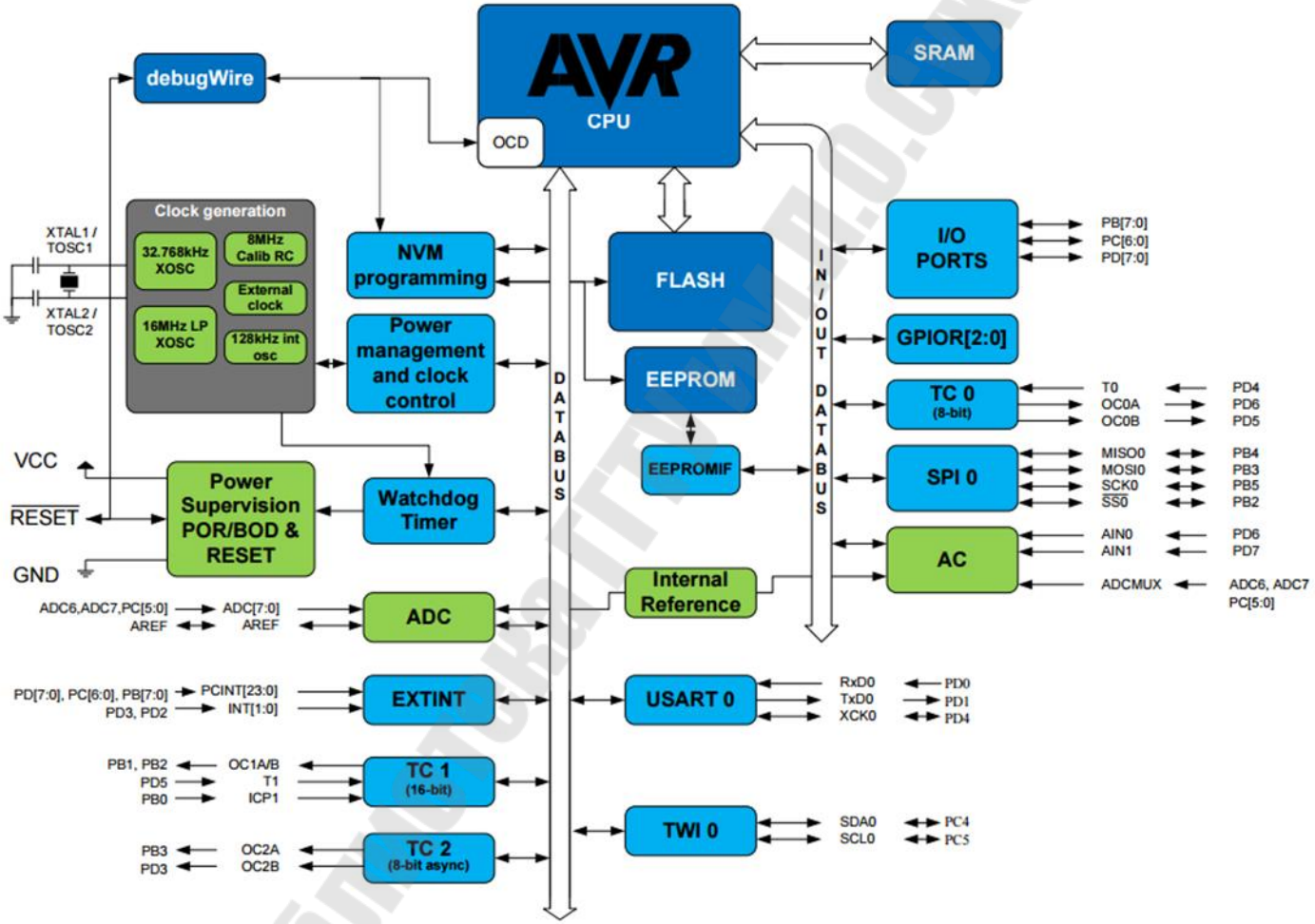


Структура контроллера ПДП i8257:



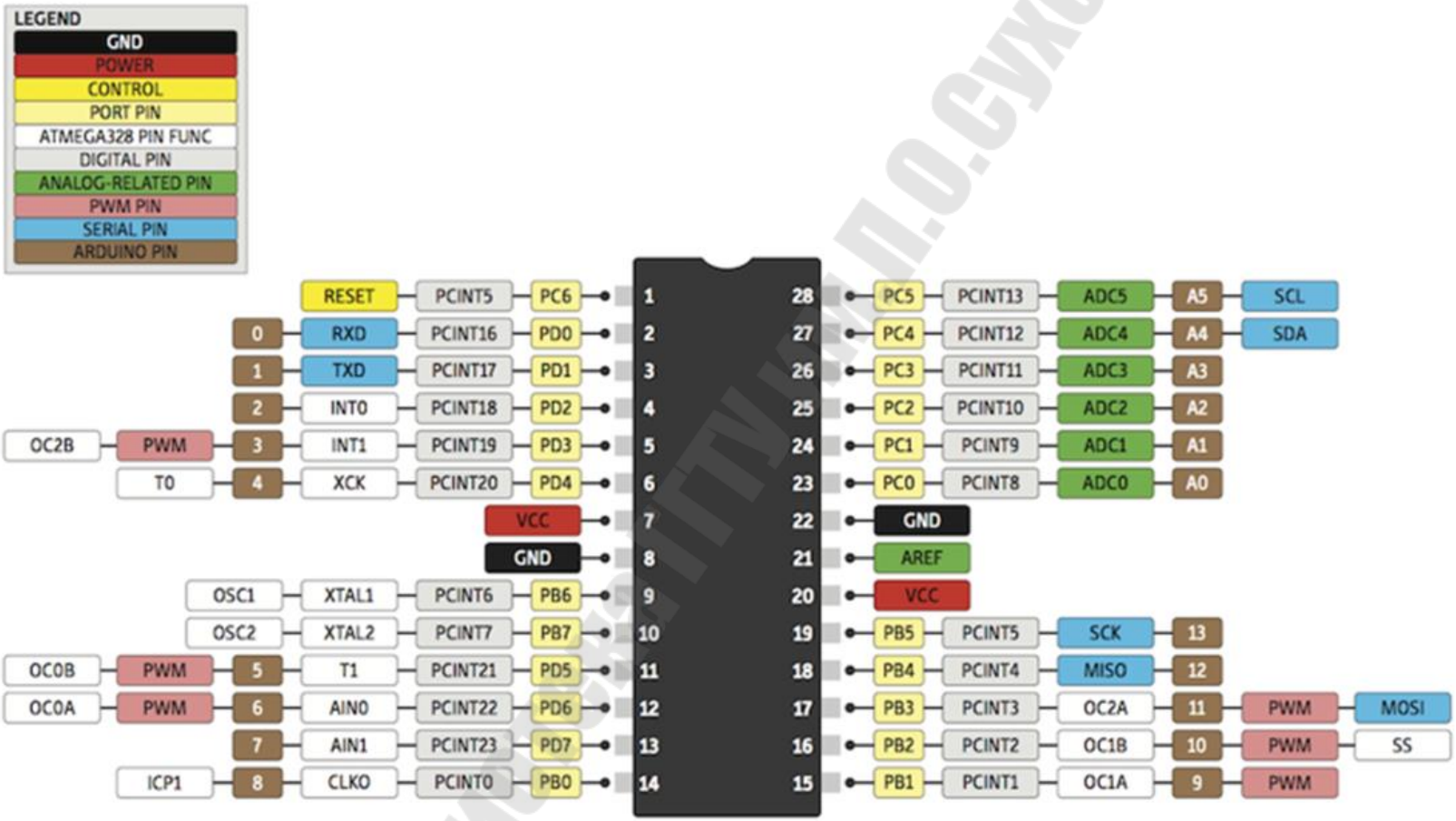
Тема 8
Архитектура микроконтроллеров

Архитектура AVR



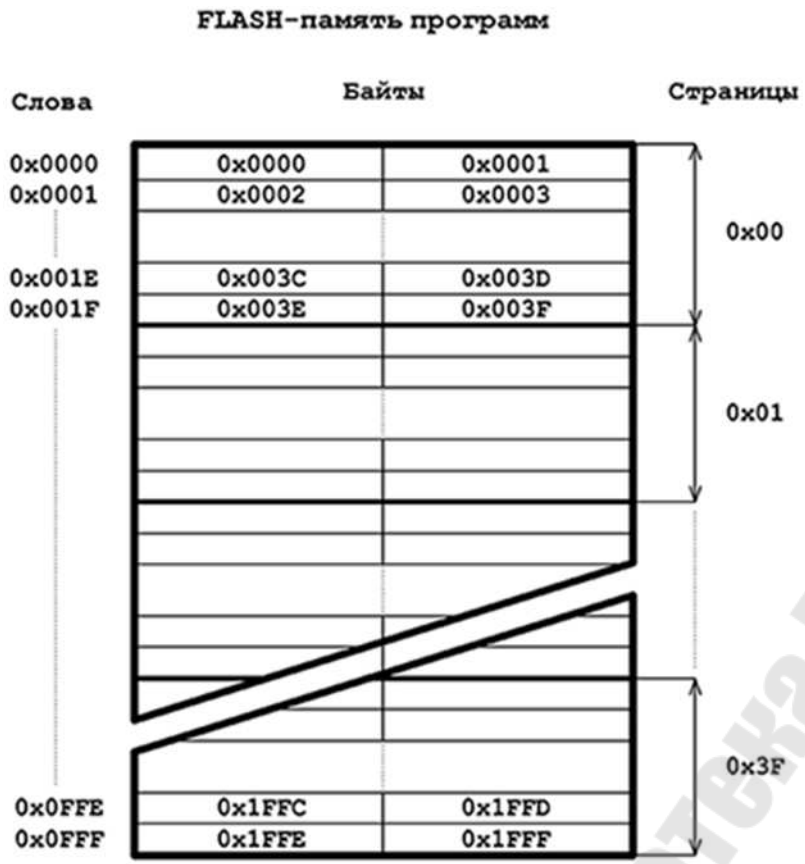
Архитектура микроконтроллера Atmega168

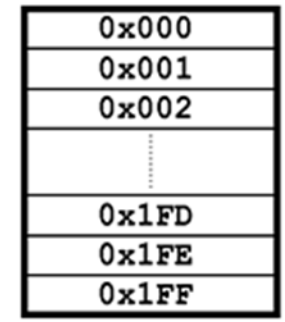
Архитектура AVR

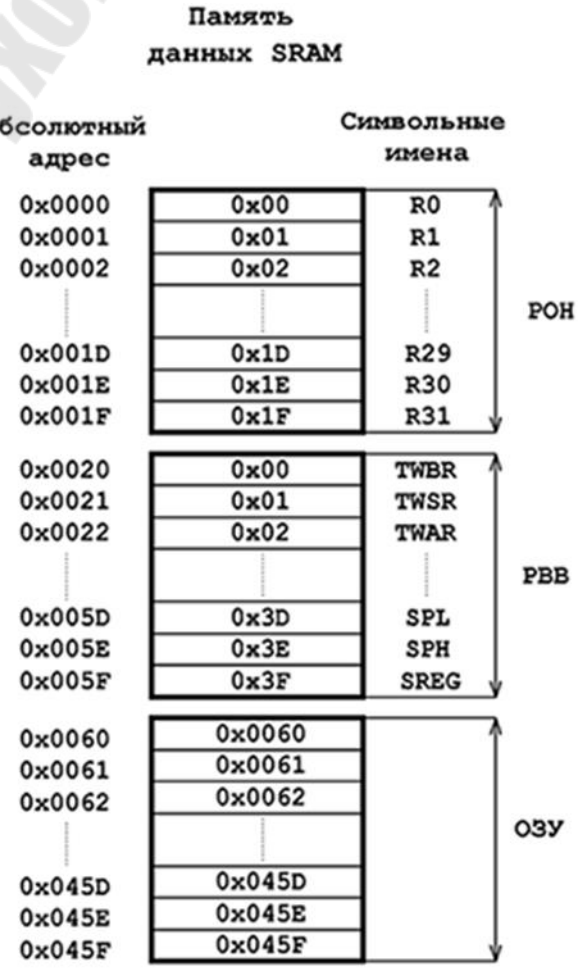


Распиновка микроконтроллера Atmega168

Архитектура AVR







Архитектура AVR

Младшие регистры	R0	Используется в команде LPM
	R1…R15	Используются произвольно (не работают с непосредственной адресацией)
Старшие регистры	R16…R25	Используются произвольно
	R26	Регистровая пара X
	R27	
	R28	Регистровая пара Y
	R29	
	R30	Регистровая пара Z
	R31	

Адресное пространство МК AVR

0x00…0x1F	Регистры общего назначения, R0…R31	R0…R15 младшие
		R16…R31 старшие
0x20…0x5F	Регистры ввода/вывода	0x00…0x3F для команд IN, OUT
0x60…	Внутреннее ОЗУ	Размер и наличие зависит от модели МК

Архитектура AVR

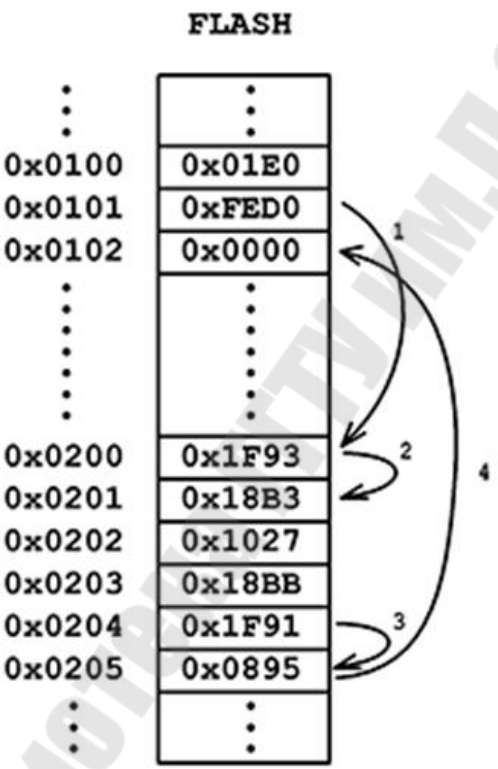
Разряд	Название	Описание
7	I	**Общее разрешение прерываний**. Для разрешения прерываний этот флаг должен быть установлен в «1». Разрешение/запрещение отдельных прерываний производится установкой или сбросом соответствующих разрядов регистров масок прерываний (регистра управления прерываниями). Если флаг сброшен, то прерывания запрещены независимо от состояния разрядов этих регистров. Флаг сбрасывается аппаратно после входа в прерывание и восстанавливается командой RETI для разрешения обработки следующих прерываний
6	T	**Хранение копируемого бита**. Этот разряд регистра используется в качестве источника или приемника команд копирования битов BLD (Bit LoaD) и BST (Bit STore). Заданный разряд любого РОН может быть скопирован в этот разряд командой BST или установлен в соответствии с содержимым данного разряда командой BLD
5	H	**Флаг половинного переноса.** Этот флаг устанавливается в «1», если произошел перенос из младшей половины байта (из 3-го разряда в 4-й) или заем из старшей половины байта при выполнении некоторых арифметических операций

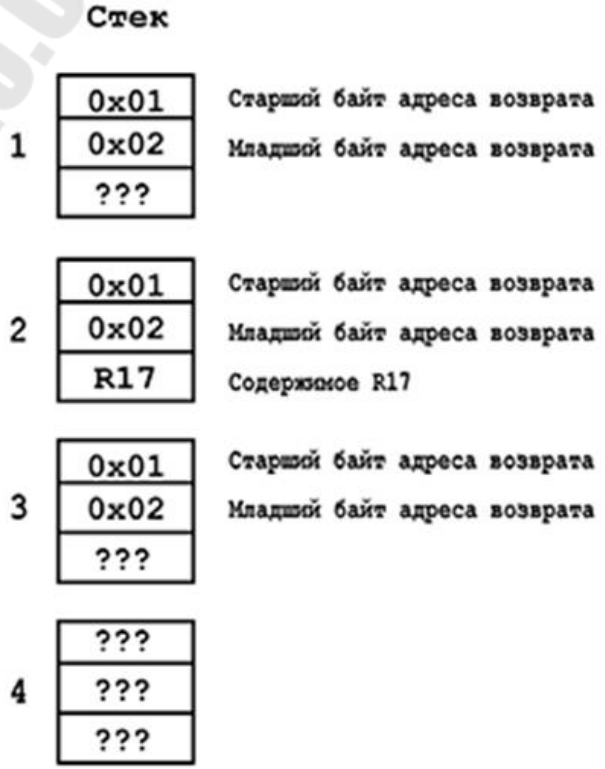
4	S	**Флаг знака**. Этот флаг равен результату операции «Исключающее ИЛИ» (XOR) между флагами N (отрицательный результат) и V (переполнение числа в дополнительном коде). Соответственно этот флаг устанавливается в «1», если результат выполнения арифметической операции меньше нуля
3	V	**Флаг переполнения дополнительного кода**. Этот флаг устанавливается в «1» при переполнении разрядной сетки знакового результата. Используется при работе со знаковыми числами (представленными в дополнительном коде). Более подробно — см. описание системы команд
2	N	**Флаг отрицательного значения.** Этот флаг устанавливается в «1», если старший (7-й) разряд результата операции равен «1». В противном случае флаг равен «0»
1	Z	**Флаг нуля**. Этот флаг устанавливается в «1», если результат выполнения операции равен нулю
0	C	**Флаг переноса.** Этот флаг устанавливается в «1», если в результате выполнения операции произошел выход за границы байта

Пример работы стека

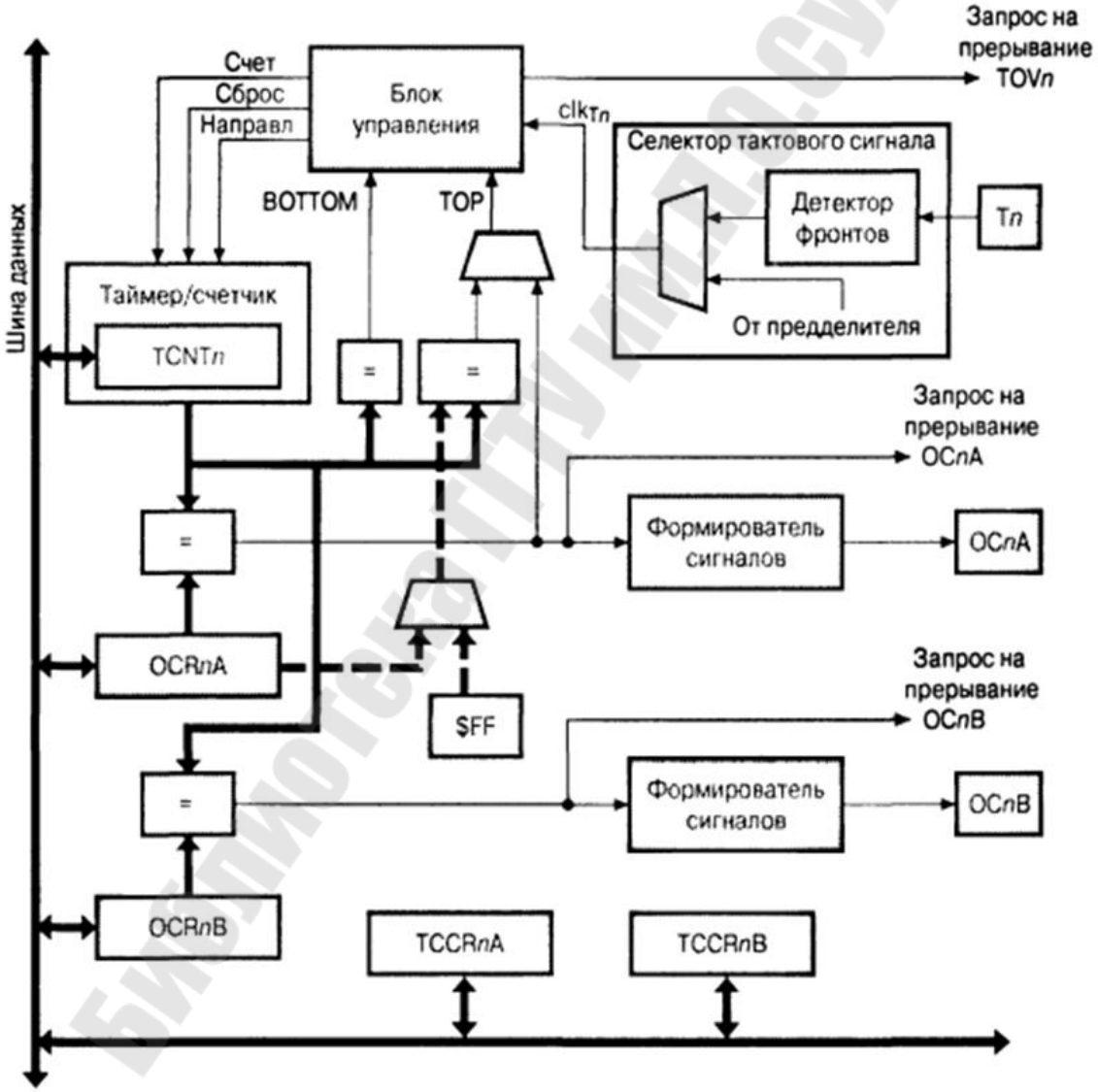
```
      .org 0x0100
     ldi R16,1<<PB0
     rcall toggl
     nop
           :
           :

      .org 0x0200
toggl:
     push R17
     in   R17,PORTB
     eor  R17,R16
     out  PORTB,R17
     pop  R17
     ret
           :
           :
```





Блок-схема 8-битного таймера



Регистры 8-битного таймера

Регистр TCCR*n*A

<table>
<tr><th>Регистр</th><th>Бит</th><th>Название</th><th colspan="5">Описание</th></tr>
<tr><td rowspan="12">TCCRnA</td><td>7, 6</td><td>COMnA1:
COMnA0</td><td colspan="5">Режим работы блока сравнения «A». Эти биты определяют поведение вывода OCnA при наступлении события «Совпадение». Влияние содержимого этих битов на состояние вывода зависит от режима работы таймера/счетчика</td></tr>
<tr><td>5, 4</td><td>COMnB1:
COMnB0</td><td colspan="5">Режим работы блока сравнения «B». Эти биты определяют поведение вывода OCnB при наступлении события «Совпадение». Влияние содержимого этих битов на состояние вывода зависит от режима работы таймера/счетчика</td></tr>
<tr><td>3, 2</td><td>—</td><td colspan="5">Зарезервированы, читаются как 0</td></tr>
<tr><td rowspan="10">1, 0</td><td rowspan="10">WGMn1:
WGMn0</td><td colspan="5">Режим работы таймера/счетчика. Эти биты совместно с битом WGMn2 регистра TCCRnB определяют режим работы таймера/счетчика следующим образом:</td></tr>
<tr><td>Номер режима</td><td>WGMn2</td><td>WGMn1</td><td>WGMn0</td><td>Режим работы таймера/счетчика Tn</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>Normal</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>Phase correct PWM</td></tr>
<tr><td>2</td><td>0</td><td>1</td><td>0</td><td>CTC (сброс при совпадении)</td></tr>
<tr><td>3</td><td>0</td><td>1</td><td>1</td><td>Fast PWM</td></tr>
<tr><td>4</td><td>1</td><td>0</td><td>0</td><td>Зарезервировано</td></tr>
<tr><td>5</td><td>1</td><td>0</td><td>1</td><td>Phase correct PWM</td></tr>
<tr><td>6</td><td>1</td><td>1</td><td>0</td><td>Зарезервировано</td></tr>
<tr><td>7</td><td>1</td><td>1</td><td>1</td><td>Fast PWM</td></tr>
</table>

Регистры 8-битного таймера AVR

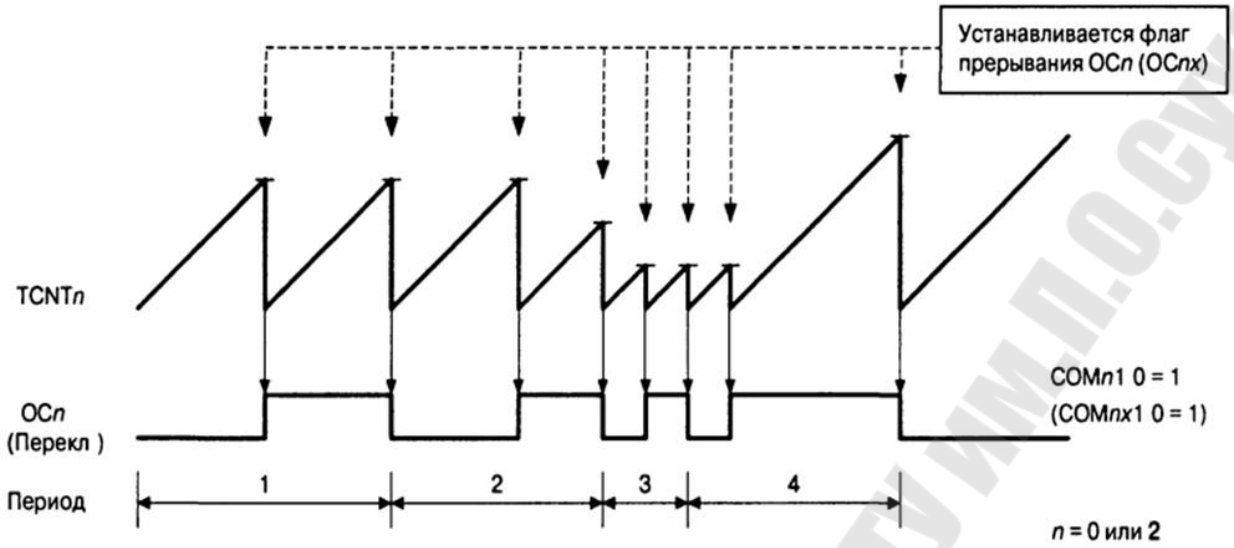
Регистр TCCR*n*B

TCCR*n*B	7	FOC*n*A	**Принудительное изменение состояния вывода OC*n*A (режимы Normal и CTC).** При записи лог. 1 в этот бит состояние вывода OC*n*A изменяется в соответствии с установками битов COM*n*A1:COM*n*A0. Прерывание при этом не генерируется и сброс таймера (в режиме CTC) не производится. В режимах Fast PWM и Phase Correct PWM этот бит должен быть сброшен в 0. При чтении бита всегда возвращается 0
	6	FOC*n*B	**Принудительное изменение состояния вывода OC*n*B (режимы Normal и CTC).** При записи лог. 1 в этот бит состояние вывода OC*n*B изменяется в соответствии с установками битов COM*n*B1:COM*n*B0. Прерывание при этом не генерируется и сброс таймера (в режиме CTC) не производится. В режимах Fast PWM и Phase Correct PWM этот бит должен быть сброшен в 0. При чтении бита всегда возвращается 0
	5, 4	—	Зарезервированы, читаются как 0
	3	WGM*n*2	**Режим работы таймера/счетчика.** Этот бит совместно с битами WGM*n*1:WGM*n*0 регистра TCCR*n*A определяют режим работы таймера/счетчика
	2...0	CS*n*2...CS*n*0	**Управление тактовым сигналом.** Эти биты определяют источник тактового сигнала таймера/счетчика. Действие этих битов зависит от исполнения таймера/счетчика и будет описано ниже

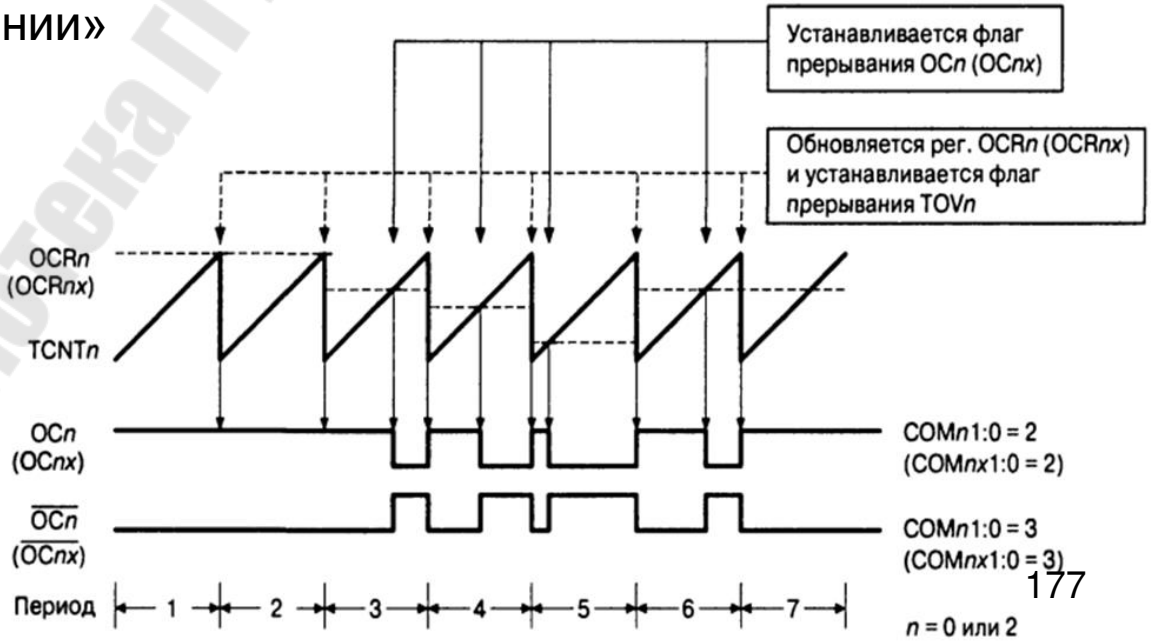
Режимы работы 8-битного таймера AVR

Номер режима	WGM*n*2[1)]	WGM*n*1	WGM*n*0	Режим работы таймера/счетчика T*n*	Модуль счета (TOP)	Обновление регистров OCR*nx*	Момент установки флага TOV*n*
0	0	0	0	Normal	$FF	Немедленно	$FF
1	0	0	1	Phase correct PWM	$FF	При TOP	$00
2	0	1	0	CTC (сброс при совпадении)	OCR*n* (OCR*n*A)	Немедленно	$FF
3	0	1	1	Fast PWM	$FF	При TOP	$FF
4[1)]	1	0	0	Зарезервировано	—	—	—
5[1)]	1	0	1	Phase correct PWM	OCR*n* (OCR*n*A)	При TOP	$00
6[1)]	1	1	0	Зарезервировано	—	—	—
7[1)]	1	1	1	Fast PWM	OCR*n* (OCR*n*A)	При TOP	TOP

[1)] В моделях ATmega48x/88x/168x, ATmega164x/324x/644x и ATmega640x/1280x/1281x/2560x/2561x.



Режим «сброс при совпадении»



Режим «быстрая ШИМ»

Аналого-цифровой преобразователь

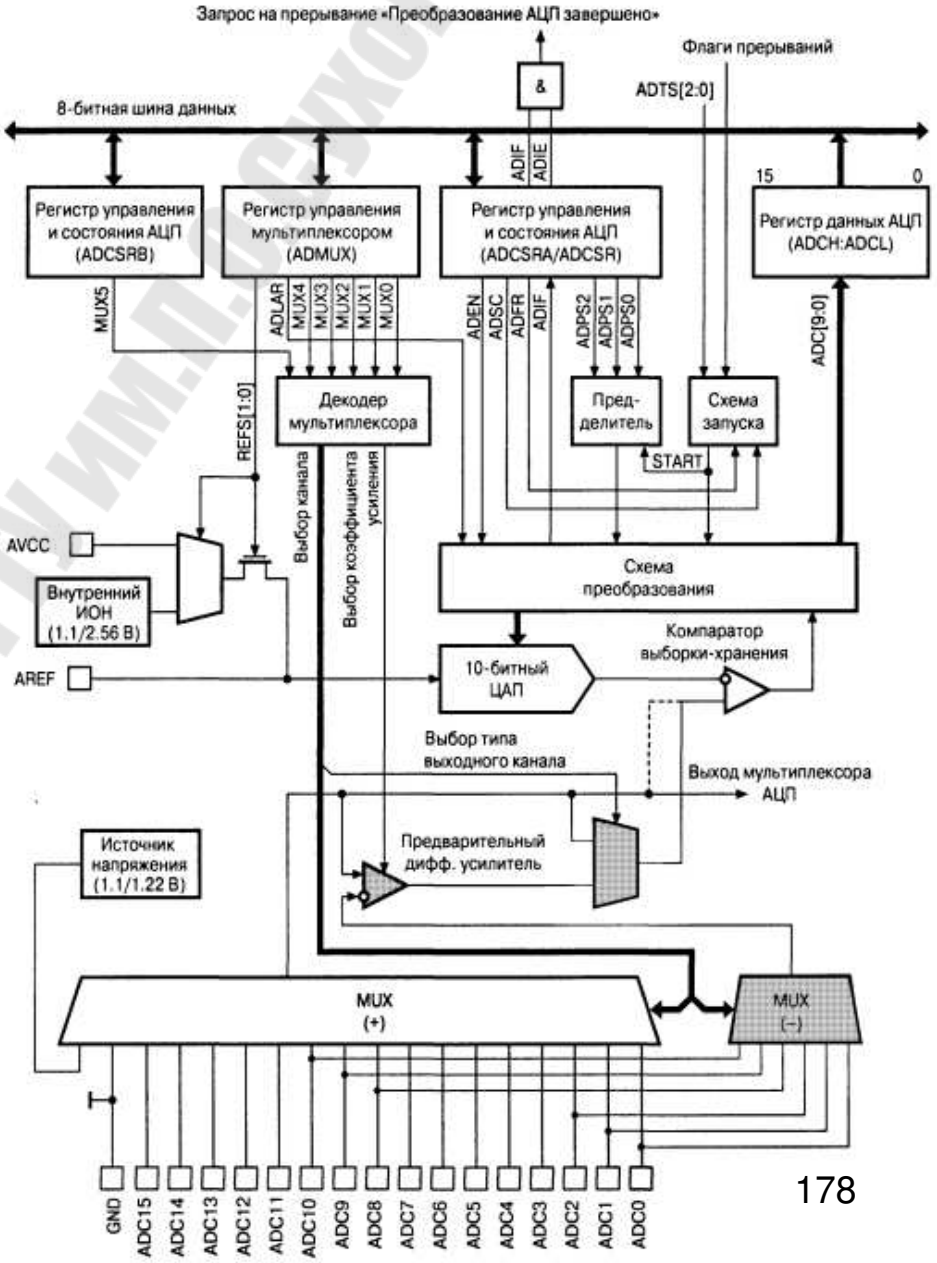
ADCSRA

Бит	Название	Описание
7	ADEN	Разрешение АЦП (1 — включено, 0 — выключено)
6	ADSC	Запуск преобразования (1 — начать преобразование)
5	ADATE (ADFR[2])	Выбор режима работы АЦП
4	ADIF	Флаг прерывания от компаратора
3	ADIE	Разрешение прерывания от компаратора
2...0	ADPS2:ADPS0	Выбор частоты преобразования

[1] В модели ATmega8x.
[2] В моделях ATmega8x и ATmega128x.

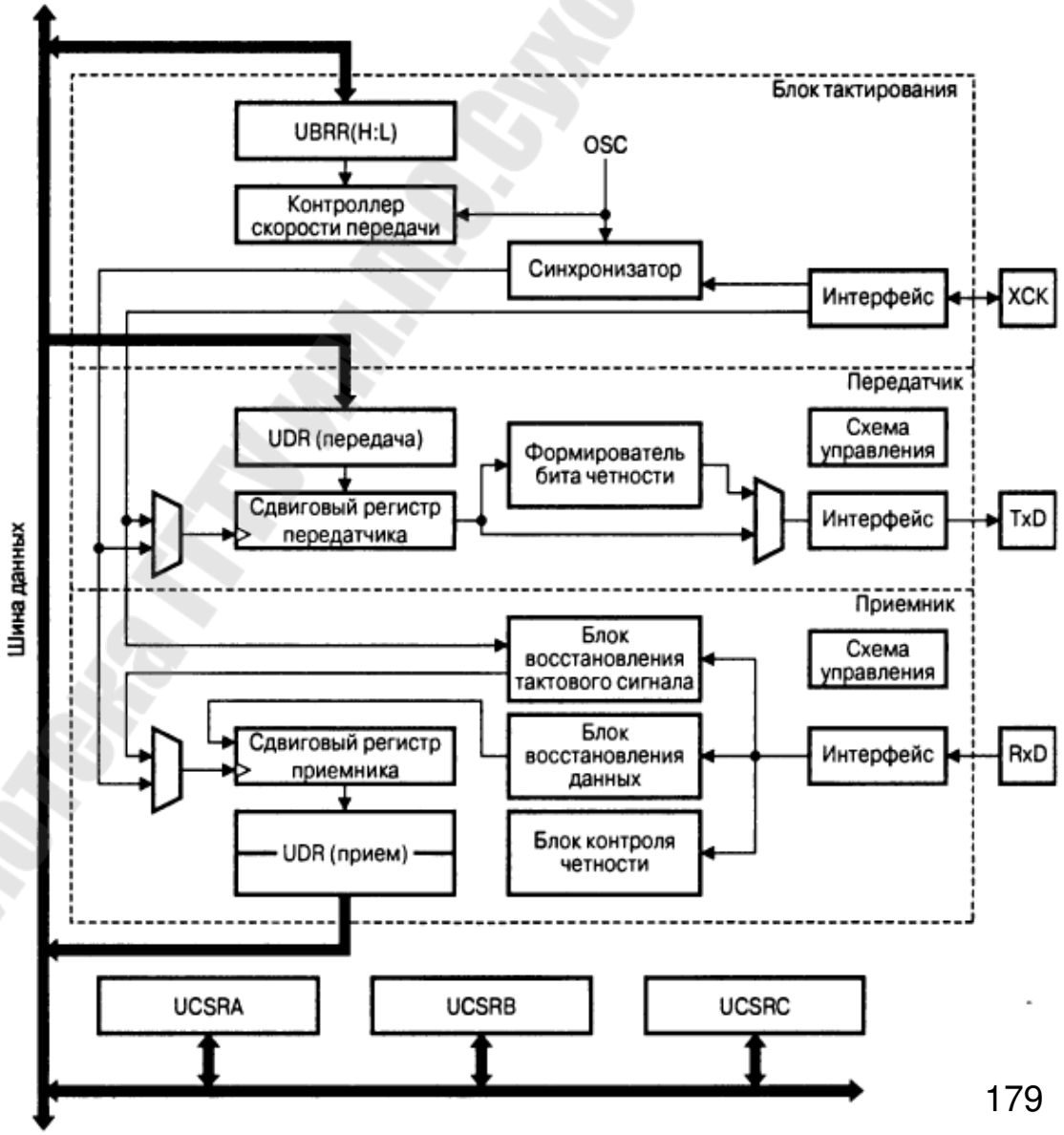
ADMUX

Бит	Название	Описание	Модель
7, 6	REFS1:REFS0	Выбор источника опорного напряжения	Все модели
5	ADLAR	Выравнивание результата преобразования	Все модели
4	—	Зарезервировано	ATmega8x, ATmega48x/88x/168x
	MUX4	Выбор входного канала	Остальные
3...0	MUX3...MUX0	Выбор входного канала	Все модели



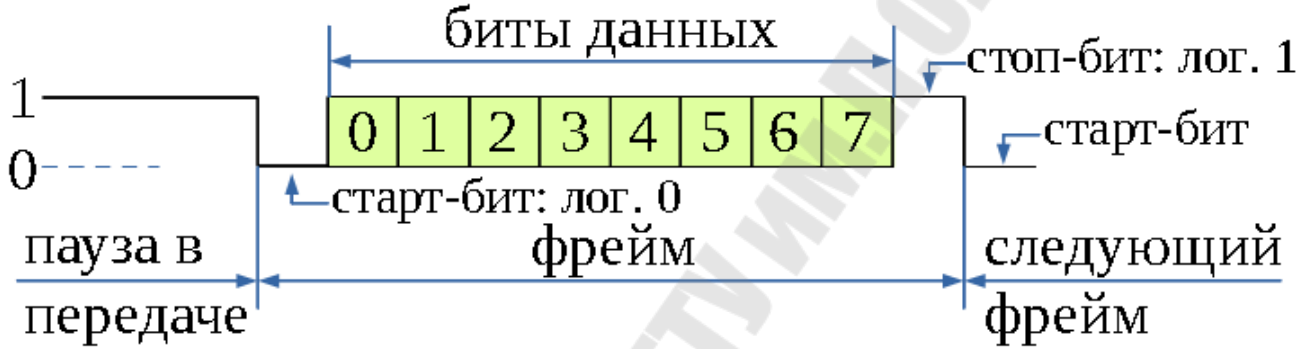
Архитектура AVR

Универсальный синхронно/асинхронный приёмо-передатчик (USART)

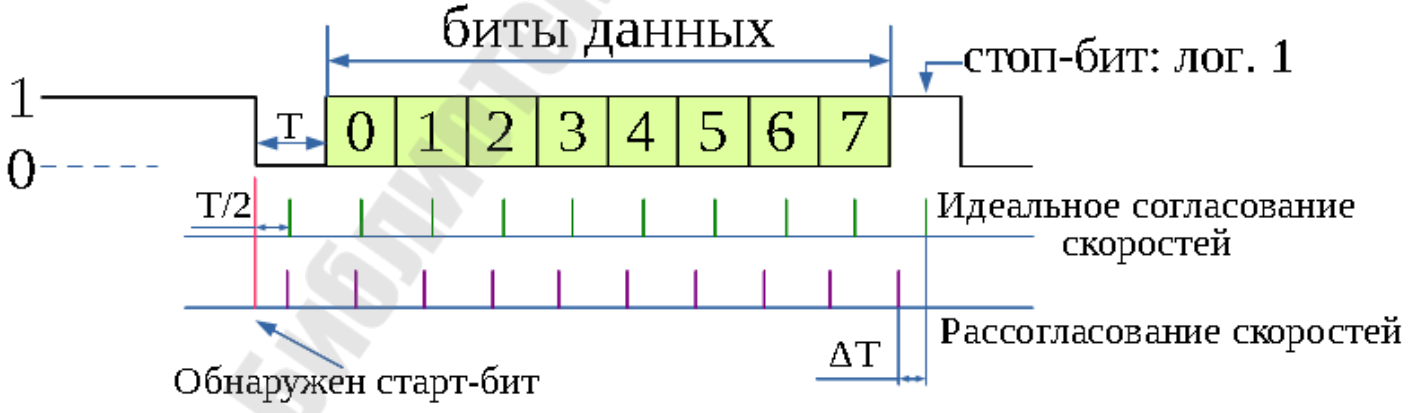


Последовательная передача данных (протокол USART)

асинхронный режим



синхронный режим



Литература

1. Таненбаум, Э. Архитектура компьютера / Э. Таненбаум, Т. Остин ; [перевел с англ. Е. Матвеев]. - 6-е изд.. - Санкт-Петербург [и др.]: Питер, 2014. - 811 с.
2. Буза, М. К. Архитектура компьютеров : учебник для вузов / М. К. Буза. - Минск : Вышэйшая школа, 2015. – 413 с.