ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Д. О. Гололобов

ОСНОВИ КОМП'ЮТЕРНОЇ ТЕХНІКИ ТА ПРОГРАМУВАННЯ МІКРОПРОЦЕСОРІВ

Навчальний посібник з курсів
«Математичні основи комп'ютерної техніки»,
«Математичні основи обчислювальної техніки»

Київ – 2019

УДК 004.2
Г61

Автор
Д. О. Гололобов, доцент кафедри системного аналізу
Державного університету телекомунікацій, кандидат фіз.-мат. наук

Рецензенти:
доктор фіз.мат. наук, проф., чл.-кор. НАН України П.С. Кнопов
(Інститут кібернетики ім. В.М. Глушкова);
доктор техн. наук, проф. О.В. Бушма
(Київський університет імені Бориса Грінченка)

Рекомендовано до друку вченою радою факультету інформаційних технологій (протокол №10 від 30 квітня 2018 року)

Г61 Основи комп'ютерної техніки та програмування мікропроцесорів : навч. посіб. / Д.О. Гололобов. – К. : Редакційно-видавничий центр Державного університету телекомунікацій, 2019. – 58с. : іл.

Висвітлено загальні відомості про системи числення, способи представлення цілих та дійсних чисел у пам'яті комп'ютера, види архітектур мікропроцесорів, модель будови статичної і динамічної пам'яті. Розглядаються основи програмування мікропроцесорів на мові Netwide Assembler (NASM) у 32-х розрядному режимі, а саме, синтаксис команд, робота із регістрами та пам'яттю, програмування лінійних, розгалужених і циклічних процесів, створення та обробка одновимірних і двовимірних масивів, використання макросів та підпрограм для оптимізації структури основної програми, система команд арифметичного співпроцесора. У достатній кількості наведено приклади програм і фрагментів коду.
Для студентів спеціальностей «Телекомунікації та радіотехніка», «Системний аналіз» та «Інженерія програмного забезпечення».

ЗМІСТ

ПЕРЕДМОВА 4
Розділ 1. ПРИНЦИПИ КОДУВАННЯ ІНФОРМАЦІЇ 6
1.1 Кодування інформації 6
1.2 Системи числення 7
1.3 Представлення цілих чисел у пам'яті комп'ютера 10
Розділ 2. АРХІТЕКТУРА МІКРОПРОЦЕСОРІВ 12
2.1 Загальна структура та схема роботи комп’ютера 12
2.2 Види архітектур мікропроцесорів 14
2.3 Регістри 64-розрядного мікропроцесора 16
2.4 Пам’ять комп’ютера 21
Розділ 3. ПРОГРАМУВАННЯ МІКРОПРОЦЕСОРІВ 24
3.1 Мова мікропроцесора та асемблер 24
3.2 Система команд мікропроцесора 25
3.3 Асемблер NASM у середовищі SASM 25
3.4 Програмування обчислювальних процесів 33
3.5 Масиви 39
3.6 Апаратний стек 44
3.7 Підпрограми 46
3.8 Макроси, їх відмінності від підпрограм 49
3.9 Кодування дійсних чисел 54
3.10 Арифметичний співпроцесор 58
БІБЛІОГРАФІЧНИЙ СПИСОК 64

ПЕРЕДМОВА

Курс «Обчислювальна техніка та мікропроцесори» (оновлений варіант має назву «Математичні основи комп'ютерної техніки») для спеціальностей напряму телекомунікацій викладається в Державному університеті телекомунікацій кафедрою системного аналізу (до квітня 2016 року – кафедра обчислювальної техніки) упродовж понад десятьох років. За цей час відбулися помітні зміни у сфері інформаційних технологій загалом і, в мікропроцесорній техніці, зокрема. У зв'язку з цим, курс потребував значного оновлення, яке й було виконано автором даного навчального посібника. Загалом, матеріал курсу, присвячений вивченню мікропроцесорів, було перероблено практично повністю. Вказані зміни й найшли своє віддзеркалення у посібнику. В останні роки у зв'язку з відкриттям в університеті нових спеціальностей «Системний аналіз» та «Інженерія програмного забезпечення» матеріал курсу викладається також і для них під назвами відповідно «Математичні основи комп'ютерної техніки» та «Математичні основи обчислювальної техніки».

Основні інновації посібника полягають у вивченні архітектури та програмування 32- і 64-розрядних мікропроцесорів під керівництвом операційної системи Microsoft Windows, а не 16-розрядних під керівництвом операційної системи Microsoft DOS (Disk Operating System), як було в курсі навчальної дисципліни раніше. Іншою важливою особливістю є використання для програмування мікропроцесорів діалекту асемблера NASM (Netwide Assembler) замість TASM (Turbo Assembler). Перехід до вивчення нового асемблера обумовлений, головним чином, наступною причиною. Асемблер TASM, який був і залишився пропрієтарним продуктом, і вивчення якого було досить поширеним у навчальних закладах вищої освіти на пострадянському просторі поряд з іншими продуктами (Turbo Pascal, Delphi тощо) компанії Borland (Borland Software Corporation), перестав бути актуальним. Це було викликано тим, що компанія Borland зазнала банкрутства, а права на її програмні продукти отримала компанія Embarcadero Technologies, яка відмовилася від подальшого розвитку та підтримки всіх продуктів лінійки Turbo.

Діалект асемблера NASM був обраний невипадково. По-перше, зазначений асемблер дозволяє писати 16-, 32- та 64-розрядні програми під DOS, Widows і UNIX-системи, є вільним (непропрієтарним) програмним забезпеченням, яке поширюється під GNU LGPL та BSD ліцензіями, підтримується і розвивається невеликою командою розробників на SourceForge.net. По-друге, порівняно із діалектами MASM (Macro Assembler) і TASM, асемблер NASM містить значно меншу кількість псевдоінструкцій (інструкцій асемблера, які не є реальними командами мікропроцесора) та вбудованих макрокоманд, що робить його помітно ближчим до машинної мови мікропроцесорів, ніж вказані асемблери. По-третє, для навчальних цілей існує досить зручне інтегроване середовище розробки SASM (Simple Assembler), яке й використовується в навчальному посібнику для демонстрації прикладів програм та результатів їх відлагоджування і виконання. Певним недоліком зазначеного вибору можна назвати хіба що дуже малу

кількість літератури з асемблера NASM. Цю проблему деякою мірою розв’язує і даний навчальний посібник.

Нарешті, зробимо одне досить важливе уточнення. У даному посібнику розглядаються центральні мікропроцесори загального призначення архітектур x86-x64 корпорації Intel. Так сталося історично і було викликано, головним чином, особливостями матеріальної бази університету, а також тим фактом, що відмінність між центральними процесорами загального призначення і цифровими сигнальними процесорами, які використовуються в галузі телекомунікацій, є хоча і помітною, але не критичною. Для пояснення цього відмітимо основні особливості сигнальних процесорів порівняно із центральними процесорами загального призначення.

Цифрові сигнальні процесори призначені для цифрової обробки інформації, яка передається на відстань. Зокрема, вони використовуються в цифровій телефонії, бездротових локальних мережах, модемах, стільниковому зв’язку, цифровому радіомовленні та телебаченні, мультимедійних системах тощо. У зв’язку з цим до них, крім вимог, що є спільними для всіх мікропроцесорів загалом (швидкість, надійність, низьке енергоспоживання тощо), до цифрових сигнальних процесорів виставляють також вимоги щодо оптимізації швидкого виконання основних операцій цифрової обробки інформації:

- цифрової фільтрації;
- кореляції та згортки;
- швидкого перетворення Фур’є;
- формування сигналів;
- демодуляції;
- кодування та декодування;
- обробки зображень.

Математично перелічені операції зводяться до операцій множення та додавання багатокомпонентних векторів і матриць, тому головною відмінністю між архітектурами центральних процесорів загального призначення і сигнальних мікропроцесорів є використання в останніх апаратного помножувача для швидкого виконання операції множення у той час, як у перших множення виконується суматором.

Сучасні цифрові сигнальні процесори, в основному, виконуються на основі RISC-архітектури, що передбачає невеликий набір простих команд мікропроцесора, шляхом композиції яких можна реалізувати складні команди. При цьому простота команд забезпечує простоту декодування і зменшення часу їх виконання, що і є досить важливим для цифрової обробки інформації. Слід зазначити, що натомість сучасні центральні процесори загального призначення, як правило, виконуються як CISC-процесори з RISC-ядром. Для програмування сигнальних процесорів зазвичай використовуються мови C та Assembler.

Розділ 1. ПРИНЦИПИ КОДУВАННЯ ІНФОРМАЦІЇ

1.1 Кодування інформації

Кодуванням називають спосіб представлення інформації у пам'яті комп'ютера.
Інформація буває різних типів: текстова (числова та символьна), графічна, звукова, відео тощо. Для уніфікації інформації (представлення різних типів інформації у єдиному вигляді) використовують **двійкову форму представлення інформації**, яка полягає в записі будь-якої інформації у вигляді послідовності (тільки) двох символів (нулів та одиниць). Зручність такого кодування полягає у простоті реалізації арифметичних і логічних операцій. Недоліком є довжина кодів. Проте, технічно простіше оперувати з великою кількістю простих однотипних елементів, ніж з невеликою кількістю складних.
Кількість інформації, яка кодується однією двійковою цифрою – нулем або одиницею, називають **бітом.**
Звідси маємо означення: **біт** – один двійковий розряд.
Таким чином, будь-яку кількість інформації будь-якого типу можна представити (а отже, й виміряти в бітах). Для вимірювання великої кількості інформації використовують похідні від біта одиниці виміру:
1 байт = 2^3 біт = 8 біт;
1 кбайт = 2^{10} байт = 1024 байт;
1 Мбайт = 2^{10} кбайт = 1024 кбайт;
1 Гбайт = 2^{10} Мбайт = 1024 Мбайт;
1 Тбайт = 2^{10} Гбайт = 1024 Гбайт
тощо.
Текстову інформацію кодують за допомогою **кодових таблиць (кодувань).**
Історично першою кодовою таблицею (кодуванням) була 8 бітова (яка містила 2^8 = 256 знаків) **ASCII (American Standard Code for Information Interchange – американський стандартний код для обміну інформацією),** яка схематично зображена на рис. 1.1.
Ця таблиця містила кодування для 7 біт (128 знаків) символів (цифри, букви латинського алфавіту, різні знаки, символи тощо), яких вистачало для тогочасних комп'ютерів. Згодом почали використовувати залишкові 128 знаків, що викликало появу нових кодувань – ISO-8859-1, кириличних **cp1251** та **KOI8** тощо. Зрештою, з'явився новий стандарт (не плутати із кодуваннями – стандарт визначає зв'язок між символом і деяким числом, а формат, згідно з яким ці символи перетворюються на байти, визначається кодуваннями) **Unicode** із кодуваннями **UTF-8, UTF-16, UTF-32.**
Числова інформація кодується простим переведенням її у двійкову систему числення. Для представлення графічної, звукової та відеоінформації у двійковій формі використовуються різні складні механізми, які є предметом розгляду цифрової обробки сигналів.

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Рис. 1.1. ASCII

1.2 Системи числення

Система числення (СЧ) – спосіб зображення чисел і відповідні йому правила дій над числами. Розрізняють **позиційні** та **непозиційні** СЧ.

Позиційною називається СЧ, в якій значення кожної цифри залежить від місця її розташування в числі. Наприклад, у десятковій СЧ: у числі 739 цифра 3 має значення 30, а в числі 1320 – значення 300.

СЧ, в якій значення цифри не залежить від її місця розташування в числі, називається **непозиційною.** Прикладом таких систем є **римська** СЧ: як у числі XCV, так й у числі CDI цифра C має значення 100.

Цифри римської СЧ і їх відповідність десятковим:

M – 1000; D – 500; C – 100; L – 50; X – 10; V – 5; I – 1.

Розряд числа є структурним елементом представлення чисел у позиційних СЧ, він відповідає позиції цифри у числі.

Розряди цілих чисел нумеруються додатними цілими числами справа наліво, починаючи з нуля. Розряди цілої частини дійсних чисел нумерується так само, як і розряди цілих чисел. Розряди дробової частини дійсних чисел нумеруються зліва направо від'ємними цілими числами, починаючи з «-1». Наприклад, для числа 124,25 маємо наступну нумерацію розрядів:

2 1 0 -1 -2

1 2 4, 2 5

Важливою властивістю **Q-кової позиційної СЧ** є те, що будь-яке записане в цій системі число можна подати в розгорнутій формі у вигляді многочлена, упорядкованого за спадними степенями основи Q:

$$q_n q_{n-1} q_{n-2} \dots q_0 q_{-1} q_{-2} \dots q_{-m} = q_n Q^n + q_{n-1} Q^{n-1} + \dots q_0 Q^0 + \dots q_{-1} Q^{-1} + q_{-2} Q^{-2} + \dots q_{-m} Q^{-m},$$

де q_i – цифри Q-кової СЧ.

Коефіцієнт Q^i називають **вагою (ваговим коефіцієнтом) і-го розряду.**

Розглянемо СЧ, які найчастіше використовуються на практиці, а саме, двійкову, десяткову і шістнадцяткову.
Двійкова СЧ містить тільки дві цифри 0 та 1.
Вісімкова СЧ містить вісім цифр: 0, 1, 2, 3, 4, 5, 6, 7.
Десяткова СЧ містить десять цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
Шістнадцяткова СЧ містить шістнадцять цифр:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
Приклади деяких інших позиційних СЧ і їх цифри:
трійкова: 0, 1, 2;
четвіркова: 0,1,2,3;
дванадцяткова: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B;
двадцяткова: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, I, J.
Відповідність між початковими числами десяткової, двійкової, вісімкової і шістнадцяткової СЧ:

десяткова	**Двійкова**	**Вісімкова**	**шістнадцяткова**
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	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

Арифметичні дії. Додавання і віднімання у позиційних СЧ відбувається порозрядно (запис X_y означає число X, записане в СЧ з основою y).
Приклади.

$$\begin{array}{r} 10010_2 \\ -\ 1001_2 \\ \hline 1001_2 \end{array} \qquad \begin{array}{r} 10010_2 \\ +\ 1001_2 \\ \hline 11011_2 \end{array}$$

$$\begin{array}{r} 14001_8 \\ -\ 5751_8 \\ \hline 6030_8 \end{array} \qquad \begin{array}{r} 14001_8 \\ +\ 5751_8 \\ \hline 21752_8 \end{array} \qquad \begin{array}{r} F000A_{16} \\ -\ BFC1_{16} \\ \hline E4049_{16} \end{array} \qquad \begin{array}{r} F000A_{16} \\ +\ BFC1_{16} \\ \hline FBFCB_{16} \end{array}$$

Переведення чисел:
– з *q*-кової СЧ у десяткову:

$$A_q = \left(a_{n-1}q^{n-1} + a_{n-2}q^{n-2} + \ldots + a_2q^2 + a_1q^1 + a_0q^0\right)_{10},$$

де A_q – початкове n-розрядне число в q-ковій СЧ.
Приклад.

$$1214_5 = 1 \cdot 5^3 + 2 \cdot 5^2 + 1 \cdot 5^1 + 4 \cdot 5^0 = 125 + 50 + 5 + 4 = 184.$$

– з десяткової СЧ у *q*-кову:
початкове число в десятковій СЧ націло ділиться на q. Отримані залишки, розташовані у порядку від останнього до першого, утворюють шукане число в q-ковій СЧ.
Приклад: $122_{10}=1322_4$

122	4		
-120	30	4	
2	-28	7	4
	2	-4	1
		3	

– переведення чисел із двійкової СЧ у вісімкову та шістнадцяткову:
(правило переведення ґрунтується на тому факті, що $2^3 = 8$; $2^4 = 16$)
для переведення у вісімкову (шістнадцяткову) СЧ, початкове число в двійковій СЧ розбивають відповідно на тріади, тобто по три розряди (тетради, тобто по чотири розряди), справа наліво (якщо зліва останні цифри розбиття не утворюють тріаду / тетраду, то їх доповнюють до тріади / тетради незначущими нулями), у відповідність кожній із яких ставлять цифру вісімкової (шістнадцяткової) СЧ (ця цифра є результатом переведення значення відповідної тріади / тетради з двійкової у вісімкову / шістнадцяткову СЧ).
Приклад.

$$\underline{0}\underline{10}\underline{011}\underline{001},\underline{111}\underline{001}\underline{000}_2 = 231,710_8$$
$$2 \quad 3 \quad 1 \quad 7 \quad 1 \quad 0$$

$$\underline{1001}\underline{1001},\underline{1110}\underline{0100}_2 = 99,E4_{16}$$
$$9 \quad 9 \quad E \quad 4$$

– переведення чисел із вісімкової та шістнадцяткової СЧ у двійкову:

у початковому числі порозрядно замість кожної із цифр вісімкової / шістнадцяткової СЧ записують відповідне двійкове трьохрозрядне / чотирьохрозрядне число, тобто тріаду / тетраду.

– переведення чисел із вісімкової СЧ у шістнадцяткову і навпаки:

початкове число у вісімковій / шістнадцятковій СЧ спочатку переводять у двійкову СЧ, потім із двійкової переводять у шістнадцяткову / вісімкову.

1.3 Представлення цілих чисел у пам'яті комп'ютера

Як правило, **додатні числа** зображуються прямим, а **від'ємні** – доповняльним кодом.

Формати представлення цілих чисел:

byte (байт) – 8 розрядів (1 байт);
word (слово) – 16 розрядів (2 байти, або 1 слово);
double word (подвійне слово) – 32 розряди (4 байти, або 2 слова);
quad word (почетверене слово) – 64 розряди (8 байтів, або 4 слова).

Наприклад, у форматі байт можна записати: у знаковій формі числа від -2^7 до (2^7-1) (або від -128 до 127), у беззнаковій – від 0 до (2^8-1) (або від 0 до 255).

Розглянемо знакову форму запису цілих чисел.

Прямий код відповідає звичайному двійковому запису числа з відведенням старшого розряду під знак (0 – «+»; 1 – «-»).

Для повного заповнення розрядів заданого формату, двійкові числа доповнюються незначущими нулями.

Обернений код (або інверсний код, або доповнення до одиниці) – інверсія прямого коду: всі нулі початкового числа у прямому коді замінюються одиницями, а одиниці – нулями.

Доповняльний код (доповнення до двійки) – це обернений код, до молодшого значущого розряду якого додається одиниця за правилами двійкової арифметики.

Наприклад,

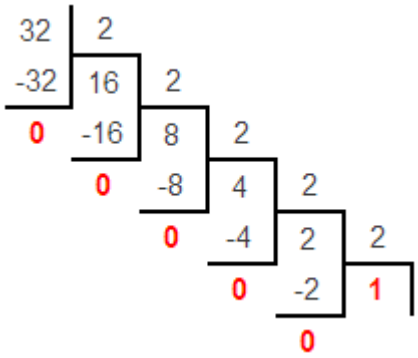
Прямий код числа -7_{10}: 0111
Обернений код числа -7_{10}: 1000
Доповняльний код числа -7_{10}: 1001

Прямий код числа 1001 – це число $1001_2 = 9_{10}$, яке відстоїть на числовій осі від числа $(-7)_{10}$ рівно на $|-7|+|9|=16=2^4$. Отже, доповняльний код числа доповнює його прямий код до числа 2^n, де n – деяке натуральне число.

Приклад. Зобразити у пам'яті комп'ютера число -32 у форматі:

a) byte (байт);
b) word (слово).

Розв'язання. Спочатку переводимо абсолютне значення заданого числа, тобто число 32 у двійкову СЧ:



Маємо $32_{10}=100000_2$.

Далі записуємо прямий, обернений і доповняльний коди числа -32:

a)

```
  1 | 0100000   - прямий код
  1 | 1011111   - обернений код
+             1
  1 | 1100000  - доповняльний код
```

b)

```
  1 | 000000000100000     - прямий код
  1 | 111111111011111     - обернений код
+                   1
-------------------------
  1 | 111111111100000     - доповняльний код
```

Відповідь:

a)

1	1	1	0	0	0	0	0

b)

1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	1

Розділ 2. АРХІТЕКТУРА МІКРОПРОЦЕСОРІВ

2.1 Загальна структура та схема роботи комп'ютера

Комп'ютер (електронно-обчислювальна машина, ЕОМ) — апаратно-програмний комплекс (система), що призначений для автоматичного виконання чітко визначеної заданої сукупності операцій, в основі кожної з яких лежить певний алгоритм.

Алгоритм — однозначна чітко визначена послідовність дій, виконання якої забезпечує отримання конкретного результату.

Властивості алгоритму:

- скінченність;
- дискретність;
- детермінованість (визначеність);
- зрозумілість;
- масовість (застосовність до різних наборів початкових даних);
- результативність (завершення алгоритму супроводжується певними конкретними результатами).

Програма — запис алгоритму в зрозумілому для сприйняття комп'ютером вигляді.

Центральним структурним елементом комп'ютера є мікропроцесор.

Мікропроцесор, або центральний процесор (Central Processing Unit, CPU; надалі МП) – програмно керований пристрій, який

а) виконує обробку цифрової інформації,

б) одночасно керує процесом цієї обробки.

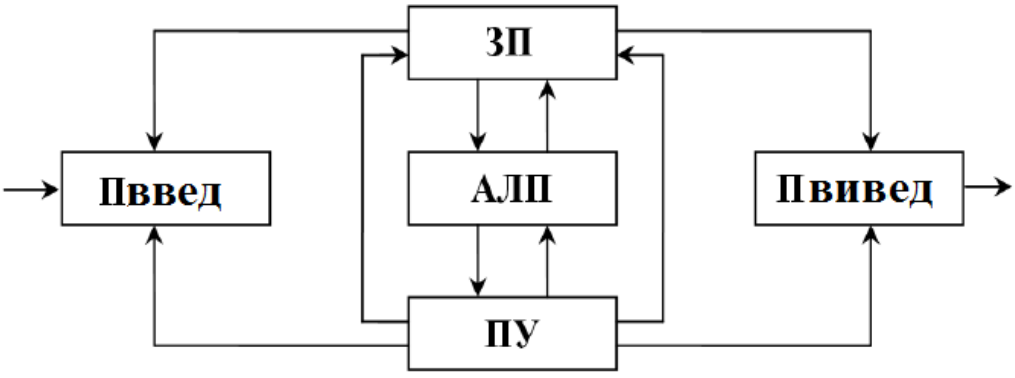


Рис. 2.1. Узагальнена структура комп'ютера

У найзагальнішому випадку комп'ютер (рис. 2.1), містить наступні складові:

- процесор:
  - ПУ – пристрій управління (або керуючий пристрій, КП),
  - АЛП – арифметико-логічний пристрій;
- ЗП – запам'ятовуючий пристрій – оперативний та постійний запам'ятовуючі пристрої (скорочено відповідно, ОЗП та ПЗП);
- ПВВ (пристрої введення-виведення):

Пввед – пристрої введення,

Пвивед – пристрої виведення.

Загальна схема роботи комп'ютера: програма (та дані) завантажуються до пам'яті комп'ютера, після чого МП починає її виконання.

Для організації взаємодії функціональних блоків комп'ютера між собою використовується система, яка називається **шиною**. Зручно вважати, що комп'ютер має три шини – **шину даних (ШД), шину адрес (ША)** та **шину управління (ШУ)** (рис. 2.2), хоча реально в сучасних системах існує одна **шина-магістраль**. Ця шина мультиплексується, тобто використовується для різних цілей у різні моменти часу.

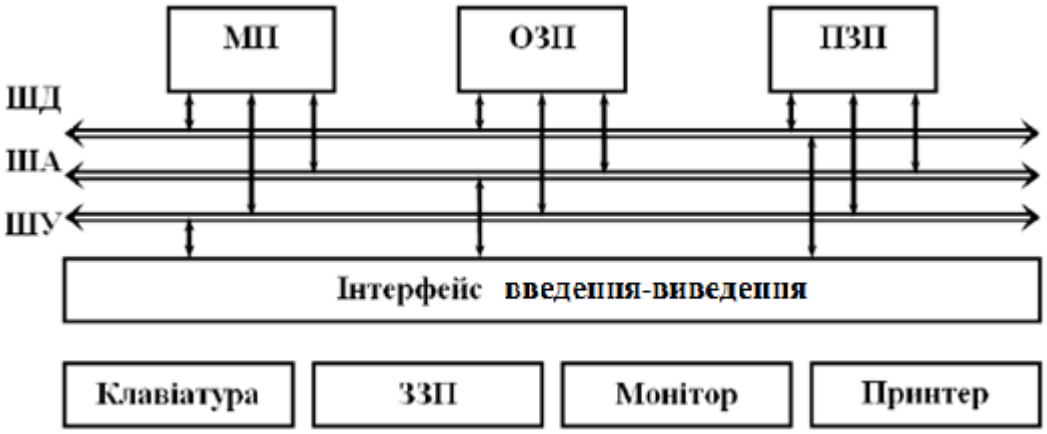


Рис. 2.2. Шини комп'ютера

ША відповідає на питання: куди направити оброблену інформацію або звідки інформацію взяти для обробки? Шина однонапрямлена в бік від МП.

ШД існує для передачі або прийому інформації (яка може знаходитись у пам'яті комп'ютера або у портах зовнішніх пристроїв), тому ШД двонапрямлена.

ШУ визначає моменти часу передачі інформації, а моменти часу у свою чергу визначаються командами МП. ШУ відповідає на питання: коли необхідно прийняти або передати інформацію?

Інтерфейс введення-виведення призначений для узгодження роботи пристроїв введення (клавіатура, зовнішні запам'ятовуючі пристрої та ін.) і пристроїв виведення (монітор, принтер та ін.) з МП (ПВВ досить часто мають такі керуючі сигнали, які відрізняються від керуючих сигналів МП).

Робота комп'ютера у загальному випадку, відбувається за наступними кроками:

1. Програма та дані через пристрої введення або ЗЗП (зовнішні запам'ятовуючі пристрої), інтерфейс введення-виведення та ШД потрапляють до ОЗП.
2. До МП надходить сигнал виконати програму.
3. Із МП по ША до ОЗП надходить адреса першої команди програми.
4. Перша команда програми з ОЗП по ШД надходить до МП, де розкодовується.
5. Адреси операндів по ША з МП надходять до ОЗП.
6. Операнди із ОЗП надходять до МП по ШД.
7. У МП виконується операція, адреса результату з МП по ША надходить до ОЗП і, одночасно, з МП по ШД до ОЗП надходить сам результат, де і відбувається його збереження.

8. Із МП по ША до ОЗП надходить адреса наступної команди. Далі будуть виконуватися пп. 4–8 доки не виконається вся програма.

2.2 Види архітектур мікропроцесорів

Незалежно від того, як виглядає корпус МП, всередині обов'язково знаходиться тонка пластинка кристалічного кремнію прямокутної форми площею всього декілька квадратних міліметрів, на якій за допомогою складного, багатоступеневого і високоточного технологічного процесу створено сотні мільйонів мікротранзисторів та інших схемних елементів. Ураховуючи це, вивчення повної структури МП не є доцільним, натомість вивчають лише його програмну модель.

Архітектура (програмна модель) МП – сукупність відомостей про склад його компонентів, до яких має доступ користувач (програміст), організацію обробки в ньому інформації та обмін інформацією із периферійними пристроями.

Архітектура МП включає до себе регістри, керуючі схеми, арифметико-логічні пристрої, запам'ятовуючі пристрої та магістралі, що з'єднують все вище перелічене. Отже, цей термін виходить за межі виключно МП як такого.

У вузькому сенсі терміном **«програмна модель»** називають структуру регістрів МП. Таке вживання зазначеного терміну є досить поширеним у сучасній літературі.

Класичні архітектури. У 30 роках XX ст. Пентагон (інша назва Міністерства оборони США) доручив Гарвардському та Прінстонському університетам розробити електромеханічну обчислювальну систему для військо-морської артилерії.

Прінстонський університет запропонував комп'ютер, який мав спільну пам'ять для збереження програм і даних. **Прінстонська** архітектура отримала також назву **архітектури фон-Неймана** за іменем наукового керівника її розробки. Гарвардський університет запропонував розробку комп'ютера (**гарвардська архітектура**), в якому для збереження програм, даних і стека використовувалися окремі банки пам'яті. Прінстонська архітектура виграла змагання, оскільки реалізація комп'ютера на її основі більше відповідала можливостям тогочасних технологій. Використання спільної пам'яті виявилося кращим варіантом внаслідок ненадійності лампової електроніки. Гарвардська архітектура майже не використовувалася до кінця 70 рр. XX ст.

Архітектури CISC та RISC. На основі архітектури фон-Неймана у кінці 70-х – на початку 80-х рр. XX ст. були створені МП з архітектурою RISC. Після появи RISC-процесорів усі інші МП були названі CISC-процесорами.

RISC (Restricted (reduced) instruction set computer; комп'ютер зі скороченим набором команд) – архітектура МП, у якій швидкодія збільшується за рахунок спрощення інструкцій таким чином, щоб їх декодування було більш простим, а час виконання – найменшим.

Характеристики RISC-процесорів:

- набір команд скорочено до 70-100 (замість декількох сотень у CISC-процесорів);

- більшість команд виконується за один такт (**такт** відповідає одному періоду імпульсів тактового генератора і є основною одиницею виміру часу виконання команд процесором);
- усі команди обробки даних оперують тільки вмістом регістрів МП, а для звернення до більш повільної оперативної пам'яті передбачені виключно інструкції типу «завантажити до регістра» та «записати до пам'яті»;
- команди мають простий і чітко заданий формат;
- з набору команд виключені рідковживані інструкції, а також команди, які не вписуються у прийнятий формат;
- состав системи команд повинен бути зручним для застосування оптимізуючих компіляторів із мов високого рівня.

CISC (Complex instruction set computer – комп'ютер із складним набором команд) - архітектура МП, яка характеризується:

- складними та багатоплановими інструкціями;
- великим набором різних інструкцій;
- нефіксованою довжиною інструкцій;
- багатоманітністю режимів адресації.

Усі сучасні МП x86, x86-x64, починаючи з Intel Pentium Pro є **CISC-процесорами з RISC-ядром**, тобто безпосередньо перед виконанням, складні CISC-інструкції перетворюються на більш простий набір внутрішніх інструкцій RISC, для чого використовують записані в розміщеному всередині ядра процесора ПЗП набори мікрокоманд – серій простих інструкцій, що в сукупності виконують ті ж дії, що й одна складна інструкція.

Підвищення продуктивності МП. Основними способами збільшення продуктивності МП є:

- **Підвищення тактової частоти** (кількості операцій (тактів) у секунду; основний спосіб на протязі тривалого періоду; обмеження – проблеми з охолодженням).
- **Конвеєризація** (одночасна обробка МП декількох інструкцій; можлива для незалежних одна від одної інструкцій).

Для прикладу візьмемо програму, що складається з п'ятьох незалежних інструкцій (рис. 2.3), в якій (для спрощення) кожний блок МП виконує інструкцію за один такт. На рис. 2.3:

IF (Instruction Fetch) – отримання інструкції,
ID (Instruction Decode) – декодування інструкції,
EX (Execute) – виконання,
MEM (Access memory) – доступ до пам'яті,
WB (Register write back) – запис до регістра.

Порівнюючи 9-ть (у разі використання конвеєра) проти 25-ти тактів (якщо конвеєр не використовувати), отримуємо виграш у швидкості виконання програми у 2,78 разів. У ідеальному випадку, враховуючи, що інструкції процесором виконуються неперервно, МП може бути зайнятий на 100%, при цьому, чим довший конвеєр (більша кількість блоків), тим більший виграш.

команда	такт								
	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i+2		IF	ID	EX	MEM	WB			
i+3			IF	ID	EX	MEM	WB		
i+4				IF	ID	EX	MEM	WB	
i+5					IF	ID	EX	MEM	WB

Рис. 2.3. П’ятирівневий обчислювальний конвеєр RISC-процесора

На практиці ця цифра знижується за рахунок наявності у програмах умовних переходів (оскільки МП не знає, за якою адресою знаходиться нова команда) та взаємопов’язаності інструкцій. Для вирішення зазначених проблем вводять блоки передбачення умовних переходів та аналізу незалежності інструкцій.

- **Суперскалярність** (найбільш навантажені блоки виготовляють у декількох екземплярах, за рахунок чого ці блоки не будуть затримувати весь конвеєр внаслідок паралельної обробки декількох інструкцій (за умови їх незалежності);
- **Багатоядерність** (один МП виготовляється з декількома фізичними ядрами, кожне з яких по суті є окремим МП, здатним виконувати обчислення незалежно від інших, наприклад, кожне з ядер може виконувати свій власний набір програм, а саме, на одному ядрі може працювати операційна система, на інших – якісь програми користувача.
- **Hyper-Threading** (технологія корпорації Intel, яка дозволяє кожному ядру МП виконувати два потоки обчислень одночасно, роблячи з одного фізичного ядра два віртуальні, кожне з яких має власний набір регістрів, лічильник команд та блок роботи із перериваннями, однак всі інші ресурси є спільними для обох ядер, внаслідок чого приріст продуктивності буде нижчим, ніж у випадку використання двох фізичних ядер, а якщо інструкції різних потоків використовують однотипні блоки, то він взагалі може зійти до нуля; за наявності Hyper-Threading операційна система бачить одне фізичне ядро, як два окремих ядра).
- **Turbo Boost** (технологія корпорації Intel; так званий «штатний розгін») із контролем напруги, сили струму, температури тощо; МП може підвищувати свою тактову частоту, відключаючи при цьому декілька ядер; на відміну від звичайного розгону не призводить до незворотної зміни температурного регламенту і збільшення енергоспоживання.

2.3 Регістри 64-розрядного мікропроцесора

Регістр МП – ділянка пам’яті всередині самого МП, яка використовується ним для проміжного збереження інформації. Довжина регістра може бути 8 (0...7), 16 (0...15), 32 (0...31) або 64 (0...63) біт.

Схема регістрів 64-розрядного МП наведена на рис. 2.4.

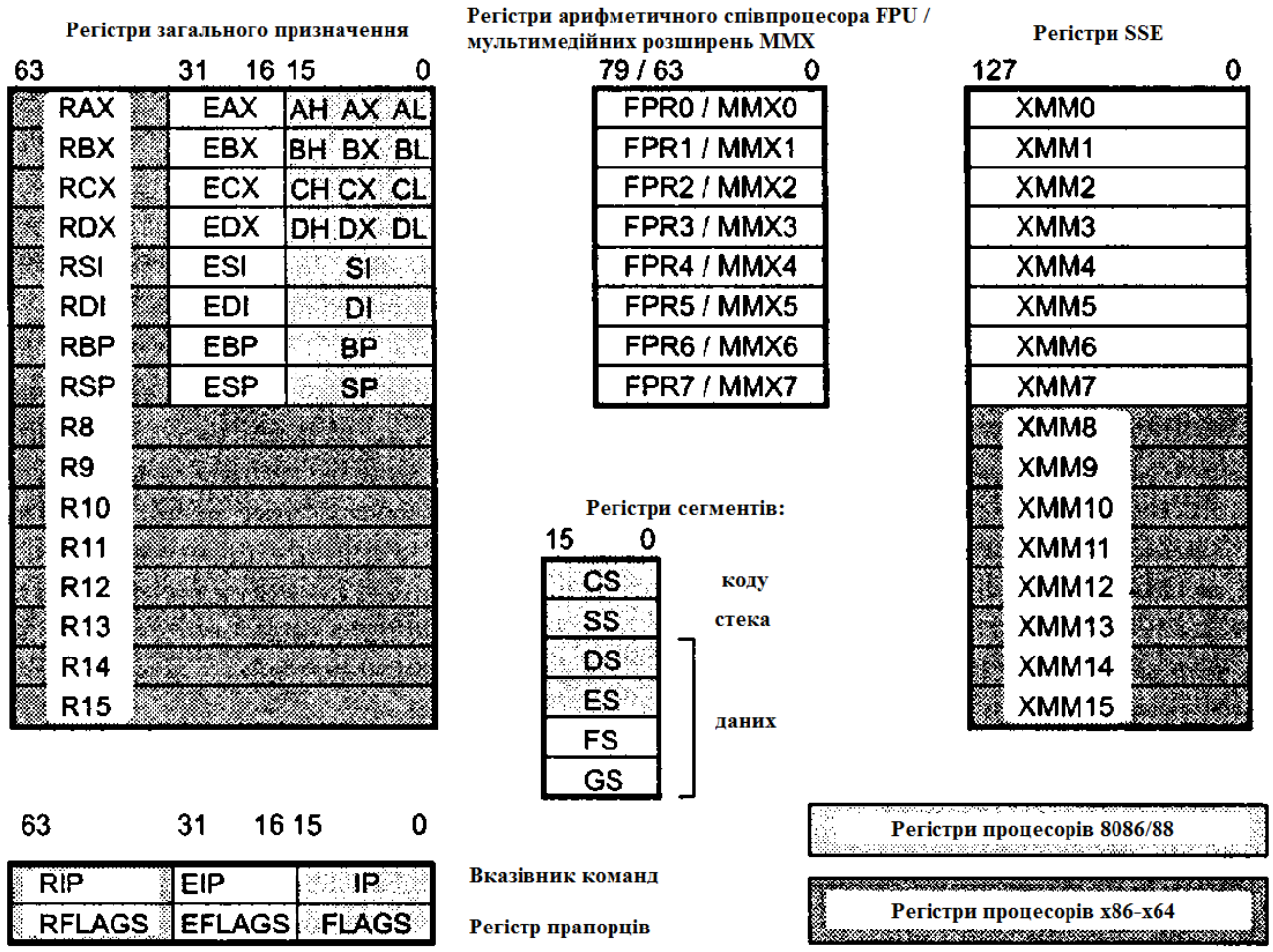


Рис. 2.4. Регістри 64-розрядного мікропроцесора

МП x86-x64 працює або в 64-розрядному з підтримкою нових регістрів та інструкцій, або в 32-розрядному режимі для оберненої сумісності старих програм. 64-розрядний режим повністю позбавлений використання сегментних регістрів, що означає загибель сегментної моделі. Також зменшене використання стека за рахунок можливості передавати більшість параметрів коду через регістри, минаючи стек (заштовхуючи параметри до стека, доводиться звертатися за адресами пам’яті, а звернення за адресами пам’яті, відсутніми у кеші МП, займає багато часу). Для доступу до 64-розрядних регістрів команди повинні містити префікс REX.

Регістри МП розділяються на:

- регістри загального призначення;
- вказівник команд;
- регістр прапорців;
- сегментні регістри.

Приклади умовних позначень регістрів:

AH, AL, CH, CL, ... – 8-розрядні регістри;

AX, CX, SP, IP, ... – 16-розрядні регістри;

EAX, ECX, ESP, EIP, ... – 32-розрядні регістри (додається буква E);

RAX, RCX, RSP, RIP, ... – 64-розрядні регістри (додається буква R).

Регістри загального призначення. До регістрів загального призначення відноситься група з 16-х регістрів, кожен із яких має розмір 64 біт і може бути розділений на дві чи більше частин (для забезпечення оберненої сумісності, тобто підтримки 32-, 16- та 8-розрядних команд).
Регістри RSI, RDI, RSP та RBP дозволяють звертатися до молодших 32 бітів за іменами ESI, EDI, ESP та EBP і до молодших 16 бітів – за іменами SI, DI, SP та BP відповідно. Регістри RAX, RBX, RCX та RDX дозволяють звертатися до молодших 32 бітів за іменами EAX, EBX, ECX та EDX, до молодших 16 бітів – за іменами AX, BX, CX та DX і до молодших 8 бітів – за іменами AH/AL, BH/BL, CH/CL та DH/DL (рис. 2.5).

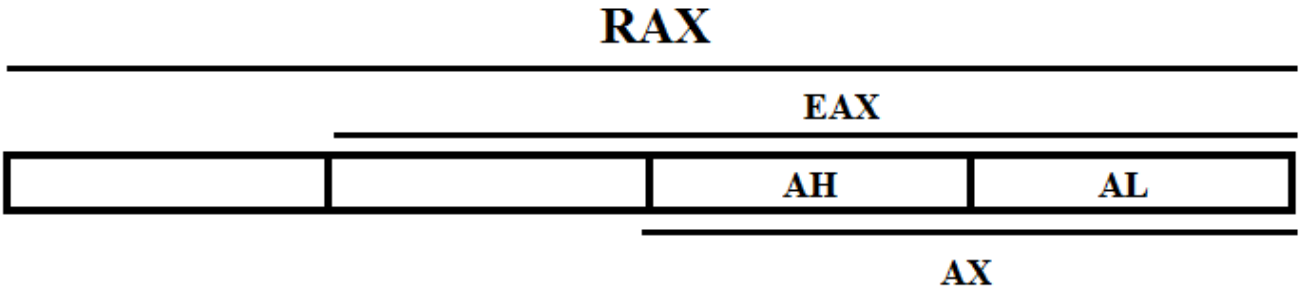


Рис. 2.5 Схема поділу регістрів загального призначення на прикладі RAX

Назви регістрів загального призначення, як правило, походять від їх призначення:

- RAX/EAX/AX/AH/AL (*accumulator register*) – акумулятор;
- RBX/EBX/BX/BH/BL (*base register*) – регістр бази;
- RCX/ECX/CX/CH/CL (*counter register*) – лічильник;
- RDX/EDX/DX/DH/DL (*data register*) – регістр даних;
- RSI/ESI/SI (*source index register*) – індекс джерела;
- RDI/EDI/DI (*destination index register*) – індекс приймача (отримувача);
- RSP/ESP/SP (*stack pointer register*) – регістр вказівника стека;
- RBP/EBP/BP (*base pointer register*) – регістр вказівника бази стека;
- R8-R15 – додаткові регістри, без спеціальних імен і призначення.

Акумулятор використовується для збереження проміжних даних.
Регістр бази використовується для збереження адреси/частини деякого об'єкта у пам’яті.
Лічильник використовується в командах із повтореннями, наприклад, при роботі з циклами та зсувами.
Регістр даних використовується для збереження проміжних даних, як і акумулятор.
Індекс джерела та індекс приймача (отримувача) використовуються для підтримки рядкових операцій, тобто операцій, що здійснюють послідовну обробку рядків сегментів, кожен із яких може мати довжину 32, 16 або 8 біт. ESI/SI в ланцюжкових операціях зберігає поточну адресу в рядку-джерелі. EDI/DI в ланцюжкових операціях зберігає поточну адресу в рядку-приймачі.
Регістр вказівника стека зберігає вказівник вершини стека у поточному сегменті стека.

Регістр вказівника бази стека призначений для організації довільного доступу до даних всередині стека.

Не дивлячись на наведену спеціалізацію, всі регістри загального призначення можна використовувати в будь-яких машинних операціях. Утім, слід враховувати той факт, що деякі команди працюють тільки з певними регістрами, наприклад, команди ділення та множення використовують регістри EAX та EDX для збереження початкових даних і результату операції, команди керування циклом використовують регістр ECX в якості лічильника циклу тощо. Існує нюанс й у використанні регістрів у якості бази, тобто сховища для адреси оперативної пам'яті. У якості регістрів бази можна використовувати будь-які регістри загального призначення, але при використанні EBX, ESI, EDI, EBP розмір машинної команди звичайно буває менший.

Вказівник команд. Регістр RIP/EIP/IP (Instruction Pointer, вказівник команд) є 64-розрядним і містить зміщення наступної команди, що підлягає виконанню. Цей регістр безпосередньо недоступний програмісту, але завантаження та зміна його значення здійснюються різними командами керування, до яких відносять команди умовних і безумовних переходів, виклику процедур та повернення з них.

Регістр прапорців RLAGS/EFLAGS/FLAGS (flag register). Регістр прапорців (рис. 2.5) містить інформацію про поточний стан МП. Використовуючи цей регістр, можна одержувати інформацію про результати виконання команд і впливати на стан самого МП.

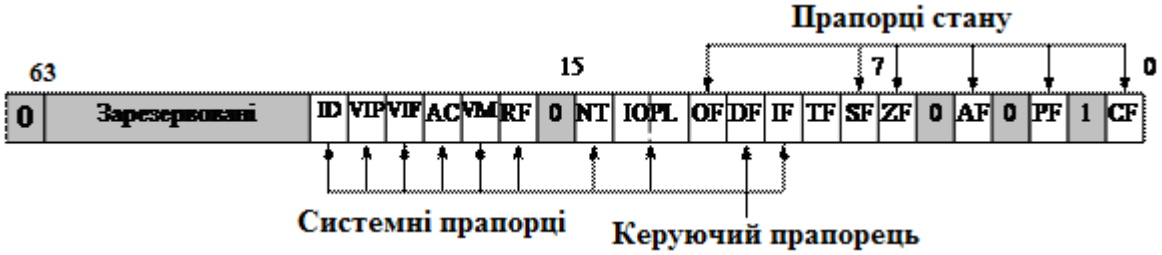


Рис. 2.5. Регістр прапорців

Окремі біти регістра RLAGS мають певне функціональне призначення і називаються **прапорцями.**

Виходячи з особливостей використання, прапорці регістра прапорців можна розділити на три групи:

- прапорці стану, які можуть змінюватися після виконання машинних команд і відображають особливості результату виконання арифметичних або логічних операцій, що дає можливість аналізувати стан обчислювального процесу та реагувати на нього за допомогою команд умовних переходів і викликів підпрограм;
- керуючий прапорець DF (Directory Flag), який знаходиться в 10-му біті регістра RFAGS і використовується для рядкових операцій; значення прапорця DF визначає напрям поелементної обробки в цих операціях: від початку рядка до його кінця або навпаки, від кінця рядка до його початку (DF = 1); для роботи з прапорцем DF існують спеціальні команди: cld (зняти прапорець DF) та std (встановити прапорець DF), застосування яких дозволяє привести прапорець DF у

відповідність із алгоритмом і забезпечити автоматичне збільшення або зменшення лічильників при виконанні операцій із рядками;

- системні прапорці, які потрібні для керування введенням/виведенням, маскованими перериваннями, відлагоджуванням, перемиканнями між завданнями і віртуальним режимом; прикладним програмам не рекомендується модифікувати без необхідності ці прапорці, оскільки в більшості випадків це призведе до переривання роботи програми.

Сегментні регістри. Існує шість сегментних регістрів: CS, SS, DS, ES, GD, FS. Їх існування обумовлено специфікою організації і використання оперативної пам'яті МП (16- і 32-розрядними) корпорації Intel. Вона полягає в тому, що МП апаратно підтримує структурну організацію програми у вигляді трьох частин, які називають сегментами. Відповідно, така організація пам'яті називається **сегментною.** Для того, щоб вказати на сегменти, до яких програма має доступ у конкретний момент часу, призначені **сегментні регістри.** У цих регістрах містяться адреси пам’яті, з яких починаються відповідні сегменти. Логіка обробки машинної команди побудована так, що при вибірці команди чи доступі до даних програми або до стека неявно використовуються адреси в цілком певних сегментних регістрах.

МП підтримує наступні типи сегментів:

1. Сегмент коду. Містить команди програми. Для доступу до цього сегменту використовується регістр CS (code segment register) – сегментний регістр коду. Він містить адресу сегменту з машинними командами, до якого має доступ МП (тобто ці команди завантажуються до конвеєра МП).
2. Сегмент даних. Містить оброблювані програмою дані. Для доступу до цього сегменту використовується регістр DS (data segment register) – сегментний регістр даних, який зберігає адресу сегменту даних поточної програми.
3. Сегмент стека. Цей сегмент є областю пам’яті, яку називають стеком. Роботу зі стеком МП організовує за наступним принципом: останній записаний до цієї області елемент вибирається першим. Для доступу до цього сегменту використовується регістр SS (stack segment register) — сегментний регістр стека, який містить адресу сегменту стека.
4. Додатковий сегмент даних. Неявно алгоритми виконання більшості машинних команд припускають, що оброблювані ними дані розташовані в сегменті даних, адреса якого знаходиться в сегментному регістрі DS. Якщо програмі недостатньо одного сегменту даних, то вона має можливість використати ще три додаткові сегменти даних. Але на відміну від основного сегменту даних, адреса якого міститься в сегментному регістрі DS, при використанні додаткових сегментів даних їх адреси потрібно вказувати явно, за допомогою спеціальних префіксів перевизначення сегментів у команді. Адреси додаткових сегментів даних повинні міститися в регістрах додаткового сегменту даних (extension data segment registers) ES, GS, FS.

2.4 Пам’ять комп’ютера

Пам’ять комп’ютера буває внутрішньою (регістрова, ОЗП, ПЗП, кеш-пам’ять) та зовнішньою.

Оперативна пам’ять (ОЗП, RAM – Random Access Memory – пам’ять з довільним доступом) – енергозалежна пам’ять для тимчасового зберігання команд і даних, які використовуються під час роботи комп’ютера. Забезпечує оперативний доступ до потрібної інформації процесору, відеокарті тощо і тимчасове збереження результатів їх роботи.

На даний момент найбільш розповсюдженими є два **типи оперативної пам’яті** – статична і динамічна.

Динамічна пам’ять (DRAM – Dynamic Random Access Memory) – енергозалежна напівпровідникова пам’ять із довільним доступом, в якій кожен двійковий розряд (біт) зберігається в конденсаторі, і яка вимагає постійної регенерації для збереження інформації.

На даний момент динамічна пам’ять є основним типом оперативної пам’яті сучасних комп’ютерів. Ця пам’ять є відносно дешевою і досить швидкою, але серйозно програє у швидкості дорогій кеш-пам’яті.

Динамічна пам’ять будується у вигляді матриці, елементи якої називають **комірками.** Кожна комірка динамічної пам’яті складається з **одного конденсатора і декількох транзисторів**. Конденсатор зберігає один біт інформації, а транзистори виконують роль ключів, які затримують заряд у конденсаторі і дозволяють доступ до конденсатора при зчитуванні або записі даних і регенерації.

Утім, конденсатор малих розмірів, який використовується в ОЗП, не здатний довго тримати заряд (більше декількох мікросекунд), крім цього, розряджання конденсатора відбувається у процесі зчитування. Отже, **конденсатор доводиться дозаряджати** десятки разів на секунду.

Переваги і недоліки динамічної пам’яті:

- переваги:
  - відносно низька собівартість;
  - високий ступінь пакування, що дозволяє створювати мікросхеми пам’яті великого об’єму.
- недоліки:
  - відносно невисока швидкодія внаслідок того, що процес зарядки/розрядки конденсатора займає набагато більше часу, ніж переключення тригера (у статичній пам’яті);
  - необхідність регенерації заряду конденсатора внаслідок його швидкого саморозряджання, причиною чого є мікроскопічні розміри самого конденсатора.

Статична пам’ять (SRAM – Static Random Access Memory) – енергозалежна напівпровідникова пам’ять із довільним доступом, в якій кожен розряд (біт) зберігається у тригері, який дозволяє підтримувати стан розряду без постійного перезапису.

Кожна комірка статичної пам’яті складається з **одного тригера і трьох або більшої кількості транзисторів.** Для організації зчитування та запису з комірки пам’яті додатково використовуються три або більше транзисторів.
Робота статичної пам’яті схожа на роботу динамічної, але процеси запису і зчитування відбуваються значно швидше, оскільки не витрачається час на заряджання / розряджання конденсаторів і не потрібна регенерація комірок.
Переваги і недоліки статичної пам’яті:
- переваги:
  - висока швидкість роботи;
  - немає потреби в регенерації.
- недоліки:
  - висока ціна;
  - низька щільність пакування;
  - невеликий об’єм;
  - велике енергоспоживання.

Враховуючи сказане, область застосування статичної пам’яті обмежується її використанням в якості кеш-пам’яті.
Кеш-пам’ять є буфером між основною оперативною пам’яттю і її «клієнтами» – МП та іншими абонентами системної шини. Вона будується на основі SRAM.
Основне призначення кеш-пам’яті – зберігати копії блоків даних областей ОЗП (зокрема ті, до яких відбувалися останні звернення), тобто її призначення – компенсувати різницю у швидкості обробки інформації МП і дещо менш швидкодіючою оперативною пам’яттю. Кеш-пам’яттю керує спеціальний пристрій – контролер, який аналізуючи виконувану програму, намагається передбачити, які дані та команди ймовірніше за все знадобляться в найближчий час процесору, і підкачує їх до кеш-пам’яті. При цьому можливі як «потрапляння», так і «промахи». У випадку «потрапляння», тобто, якщо до кеша підкачані потрібні дані, вибирання їх із пам’яті проходить без затримки. Якщо ж необхідна інформація в кеші відсутня, то МП зчитує її безпосередньо з оперативної пам’яті. Співвідношення кількості «потраплянь» і «промахів» визначає ефективність кешування.
Кеш ділиться на декілька рівнів. У сучасних МП зазвичай цих рівнів три. Кеш більш високого рівня завжди більша за розміром, але повільніша за швидкістю порівняно із кеш більш низького рівня.
Найшвидша і найменша (зазвичай декілька сотень мегабайт) кеш-пам’ять – **кеш першого рівня**, яка як правило працює на частоті МП. Може бути єдиною, а може розділятися на пам’ять команд і даних.
Кеш другого рівня дещо повільніша (час доступу 8-12 тактів процесора), але має дещо більший розмір.
Кеш третього рівня повільніша за перші дві і має найбільший об’єм.
У багатоядерних МП останній рівень кеш-пам’яті зазвичай роблять спільним для всіх ядер, що дозволяє динамічно збільшувати її при збільшенні навантаження на якесь окреме ядро. Кеш нижчих рівнів зазвичай окремі для кожного ядра.

Постійний запам'ятовуючий пристрій (ПЗП) або Read Only Memory (ROM) – мікросхеми, з яких можна тільки читати дані (змінювати неможна). У кожному персональному комп'ютері є декілька мікросхем ПЗП. Перш за все, до постійної пам'яті записують програму керування роботою самого МП. У ПЗП знаходяться програми управління дисплеєм, клавіатурою, принтером, зовнішньою пам'яттю, програми завантаження і зупинки комп'ютера, тестування пристроїв.

Постійна пам'ять знаходиться в модулі на материнській платі. Зазвичай, ПЗП має розмір 1-2 Мбайти. Важлива мікросхема постійної пам'яті – модуль ROM BIOS.

BIOS (Basic Input / Output System – базова система введення-виведення) – сукупність програм, призначених для

- автоматичного тестування пристроїв після увімкнення живлення комп'ютера;
- завантаження операційної системи із зовнішнього запам'ятовуючого пристрою (вінчестера, дискети, компакт-диска) до оперативної пам'яті.

Постійна пам'ять ROM BIOS доповнюється невеличкою енергонезалежною оперативною пам'яттю CMOS RAM.

CMOS RAM (Complementary Metal-Oxide Semiconductor RAM) – це пам'ять з невисокою швидкодією і мінімальним енергоспоживанням від батарейки. Використовується для зберігання інформації про конфігурацію та склад устаткування комп'ютера, а також про режими його роботи. Вміст CMOS змінюється спеціальною програмою Setup, яка знаходиться у BIOS.

Сегментування пам'яті. У МП x86 для погодження розрядності шини адрес і шини даних був уведений механізм сегментування. Полягає він у тому, що кожна програма «бачить» не всю пам'ять, а тільки декілька сегментів. Під сегментом розуміється ділянка пам'яті, обсяг якої 65536 байт = 64 кб.

Один сегмент (головний) відводиться під програму, другий – під дані, третій – під стек. У залежності від складності програми, сегменти можна об'єднувати. Наприклад, якщо заздалегідь відомо, що програма разом із даними та стеком займає менше 64 кб, то використовують всього один сегмент. Таким чином, користувач має можливість керувати процесом сегментування (обирати модель пам'яті).

Початкова адреса сегмента завжди починається з номера, кратного числу 16. Такий шістнадцятковий п'ятизначний номер закінчується нулем, який при записі до сегментних регістрів опускається. Початкова адреса сегмента зберігається в сегментних регістрах (розрядність яких 16 біт):

в регістрі CS – початкова адреса сегмента програми (коду),
в регістрі DS – початкова адреса сегмента даних,
в регістрі SS – початкова адреса сегмента стеку,
в регістрі ES – початкова адреса додаткового сегмента даних.

Нумерують байти в сегменті від 0000 до FFFF. Цей номер називають зміщенням (відносно початкової адреси сегмента).

Розділ 3. ПРОГРАМУВАННЯ МІКРОПРОЦЕСОРІВ

3.1 Мова мікропроцесора та асемблер

Кожен МП має свою конкретну **машинну мову**, тобто двійкові коди команд та інших умовних позначень (регістрів, змінних, чисел). Наприклад, двійкові коди програми для 16-розрядного МП корпорації Intel, які знаходяться в чотирьох сусідніх байтах:

B 0 0 5 0 4 0 D

означають:
– занести константу 5 до регістра AL,
– виконати операцію додавання двох чисел; перше число знаходиться в регістрі AL; друге є константою, величина якої 13.
Людині незручно створювати програми машинною мовою. У зв'язку з цим була створена велика кількість різноманітних зручних для людини і абстрактних щодо машинної мови мов програмування. Очевидно, що такі мови є безпосередньо «не зрозумілими» для процесорів, тому для виконання програм, написаних на них, використовуються спеціальні програми – **транслятори.**
Створені абстрактні мови мають різний рівень абстрагування. **Мови високого рівня** практично повністю абстраговані від машинної мови і орієнтуються виключно на зручність людини. **Мови низького рівня** є компромісним варіантом – вони забезпечують мінімально необхідну зручність для людини, але водночас – за змістом і принципом роботи – дуже нагадують машинну мову, відрізняючись від неї тільки формою запису програм.
Мова низького рівня Assembler є символьним записом машинної мови, наведені вище команди записуються в ній так:

mov al,5
add al,13

Отже, форма запису інша, а порядок дій той самий: перша дія – занести число до регістра, друга дія - виконати додавання.
Аналог цих самих дій у мові високого рівня є близький до людського розуміння, але далекий від машинної мови, наприклад, у мові Паскаль:

A:=5; B:= A+13;

На питання, куди поміщуватимуться операнди і де буде отримуватися результат, а отже, як швидко будуть виконані зазначені команди, відповідає транслятор.
Таким чином, мова Assembler відображує зміст машинної мови і водночас є достатньо зрозумілою та зручною для людини.
Транслятори, які перетворюють програму на мові Assembler на машинну мову, називаються **асемблерами**. Існують транслятори, що здатні виконувати зворотне

перетворення. **Дизасемблер** - транслятор, який перетворює машинний код, об'єктний файл або бібліотечні модулі на текст програми на мові Assembler.

3.2 Система команд мікропроцесора

Усі можливості МП як перетворювача та оброблювача інформації закладені до його **системи команд**. Залежно від архітектури МП, система команд може включати від декількох десятків (архітектура MISC – minimal instruction set computer) до декількох сотень (архітектура CISC) команд.

Система (набір) команд МП – угода про засоби програмування, які надаються архітектурою, а саме:

- типи даних;
- інструкції;
- системи регістрів;
- методи адресації;
- моделі пам'яті;
- способи обробки переривань та виключень;
- методи введення та виведення.

Базовою системою команд (базовими командами) МП називають команди, які найчастіше використовуються і присутні в більшості МП, незалежно від архітектури. Крім базових команд існують команди розширень.

Базові команди МП:

- передавання (копіювання, пересилання): mov, lea, in, out, xchg;
- арифметичні: add, sub, mul, div, inc, dec;
- логічні: and, or, xor, not, cmp;
- переходів (умовних та безумовного): jc, js, jnc, jns, loop та jmp;
- перетворення: cbw, cwd, neg, xlat;
- спеціальні: int, cli, sti, push, pop.

Команди розширень:

- X87 – розширення, що містить команди математичного співпроцесора (для роботи з дійсними числами);
- MMX – розширення, яке містить команди для кодування / декодування потокових аудіо- та відеоданих;
- SSE – розширення, що містить набір інструкцій для роботи зі скалярними та упакованими типами даних;
- SSE2 – модифікація, яка містить інструкції для потокової обробки цілочисельних даних, що робить це розширення кращим за MMX, яке історично з'явилося значно раніше;
- SSE3, SSE4 – додаткові інструкції розширення SSE.

3.3 Асемблер NASM у середовищі SASM

Діалекти мови Assembler. Існує чимало різних асемблерів:
MASM, GAS, fasm, NASM, RosASM, TASM, Yasm, HLASM, GoAsm.
Розглянемо найвідоміші з них.

MASM (Macro Assembler) – продукт Microsoft Corporation для МП сім'ї x86 під операційні системи DOS та Windows; містить багато макрозасобів, а також конструкції високого рівня для повторень, викликів процедур і чергувань, що робить його асемблером високого рівня. У результаті конкуренції із TASM та NASM, став некомерційним продуктом.

TASM (Turbo Assembler) – суто комерційний продукт американської компанії Borland (ліквідована 2009 року шляхом поглинання британською компанією Micro Focus; зараз права на TASM належать компанії Embarcadero Technologies), призначений для розробки програм на мові асемблера для архітектури x86. Має режим сумісності з MASM, що дозволяє транслятору TASM транслювати початковий код, написаний на MASM. Як і всі інші продукти лінійки Turbo, TASM на даний момент не підтримується Embarcadero Technologies.

NASM (Netwide Assembler) – вільний (GNU LGPL та ліцензія BSD) асемблер архітектури Intel x86, який підтримує написання 8-, 16-, 32- та 64-розрядних програм. Створений Саймоном Тетхемом разом із Юліаном Холлом. На поточний момент підтримується та розвивається невеличкою групою розробників на SourceForge.net.

Деякі відмінності NASM від інших асемблерів:

– регістрочутливість для таких міток, як foo, Foo та FOO;

– NASM вимагає квадратні дужки для посилань на пам'ять;

– NASM не підтримує директиву ASSUME (програміст сам відповідає за те, що він поміщує до сегментних регістрів);

– NASM не підтримує моделі пам'яті (програміст повинен самостійно слідкувати, які функції передбачається викликати ближнім RETN (або RET), а які дальнім (RETF) викликом;

– особливості обробки чисел із плаваючою крапкою (NASM посилається на регістри співпроцесора за іменами st0, st1, ... на відміну від ST(0), ST(1), ... , як це робиться в MASM);

– за історичними причинами NASM використовує ключове слово TWORD там, де MASM і сумісні з ним асемблери використовують TBYTE;

– NASM не підтримує синтаксис резервування неініціалізованого простору типу «DW ?», як у TASM / MASM.

Існує **два види синтаксису** асемблерів:

Intel-синтаксис – використовується в документації Intel, в асемблерах для DOS та Windows (MASM, TASM, FASM, NASM тощо). Особливості:

– приймач знаходиться зліва від джерела;

– назви регістрів зарезервовані (їх неможна використовувати в якості операндів).

AT&T-синтаксис – використовується в GAS.

Особливості:

– приймач записується після джерела;

– назви регістрів пишуться через префікс % (%eax, %ebx,...), що дозволяє використовувати eax, ebx тощо як операнди.

Загальний синтаксис команди NASM. Кожний рядок NASM (якщо це не макрос, препроцесорна або асемблерна директива), може містити комбінацію чотирьох полів:

мітка : інструкція операнди ; коментар

Як правило, більшість цих полів є необов'язковими.

Для забезпечення зручності роботи програміста створюють інтегровані середовища розробки.

IDE (Integrated Development Environment; інтегроване середовище розробки) – комплекс програмних засобів, призначений для розробки програмного забезпечення. IDE містить у собі:

- текстовий редактор;
- транслятор (компілятор та / або інтерпретатор);
- засоби автоматизації роботи;
- відлагоджувач.

Відлагоджувач – програма, яка дозволяє переглядати регістри, пам'ять, покроково виконувати програму тощо.

Для роботи з асемблером у даному курсі використовується IDE SASM (рис. 3.1).

SASM (Simple Assembler) – просте кросплатформне (Windows, Linux) середовище розробки для мов асемблера NASM, MASM, GAS, FASM із підсвічуванням синтаксису та відлагоджувачем. Поширюється за вільною ліцензією GNU GPL v3.0. Розробник – програміст Дмитро Манушин. SASM містить бібліотеку макросів для NASM, яка робить його зручним середовищем для тих, хто тільки починає вивчати Assembler.

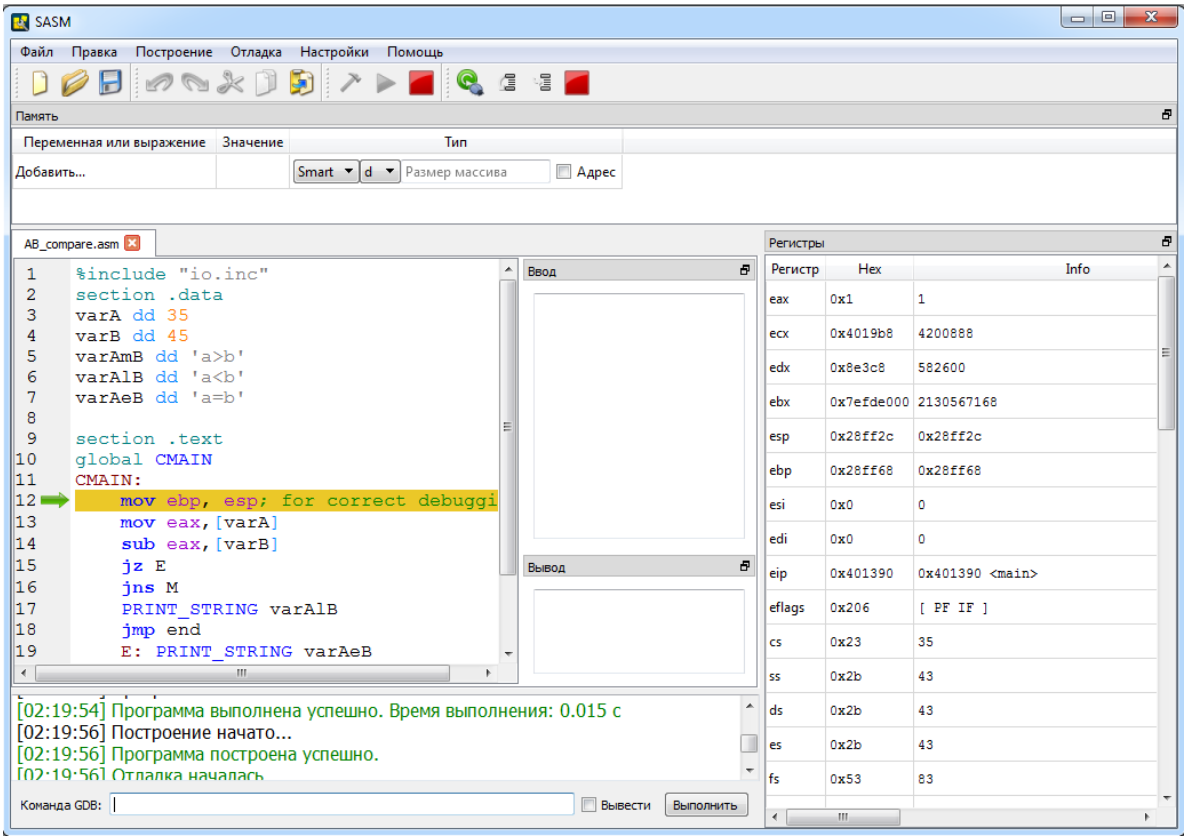


Рис. 3.1. Середовище розробки SASM

Приклад програми обчислення суми чисел a та b на Асемблері NASM:

%include "io.inc"

```
section .data
   msg db 'a + b =', 0
section .text
   global CMAIN
CMAIN:
   PRINT_STRING msg
   mov al,2
   mov bl,3
   add al,bl
   PRINT_DEC 1, al
   ret
```

Типи даних. Для асемблера тип даних (integer, string, char, ...) – поняття умовне. Більш коректним буде казати про розмір даних, а що з виділеним розміром даних робити – справа програміста.

У NASM розрізняють такі дані:

db (define byte) – 1 байт;

dw (define word) – 2 байти (слово);

dd (define double word) – 4 байти (подвійне слово);

dq (define quad word) – 8 байт (почетверене слово);

dt (define ten bytes) – 10 байт (для нецілочисельних даних).

```
1   %include "io.inc"
2   section .data
3   var1 db 24
4   var2 dw 400
5   var3 dd 70000
6   var4 dq 1234567891234567891
7   message db 'Hello world'
8   cnst equ 12
9
10   section .bss
11   buffer: resb 64 ;резервування 64 байтів
12   bytevar: resb 1 ;резервування 1 байта
13   wordvar: resw 1 ;резервування слова
14   doubleword: resd 1 ;резервування подвійного слова
15   quadword: resq 1 ;резервування почетвереного слова
16
17  section .text
18  global CMAIN
19  CMAIN:
20      mov eax,4 ;безпосередній запис константи до регістра
21      mov ebx,cnst ;запис визначеної заздалегідь константи до регістра
22      mov eax,ebx ;запис вмісту одного регістра до іншого регістра
23      mov eax,[var3] ;запис змінної до регістра
24
25      mov [var1],al; запис вмісту регістра до пам'яті
26      mov byte[bytevar], 5; безпосередній запис константи до пам'яті
27      mov dword[wordvar],cnst; запис визначеної заздалегідь константи до пам'яті
28      ret
```

Рис. 3.2. Змінні та константи у NASM

Сегменти (секції, області). Директива section вказує, в яку секцію кінцевого файлу буде асембльований написаний код. Об’єктні формати Unix та bin підтримують стандартизовані імена секцій: .text, .data, .bss. Область коду доступна

тільки для читання, секції .data та .bss доступні також і для запису. Області ініціалізованих даних (section .data), неініціалізованих даних (section .bss) та коду (section .text) можна розташовувати в довільному порядку одна відносно одної.
Приклади створення змінних різних типів, констант і команд переміщення наведені на рис. 3.2.
Команда переміщення даних. Загальний синтаксис пересилання даних:

mov приймач,джерело

В якості **приймача** може виступати регістр або ділянка пам'яті. В якості **джерела** – регістр, ділянка пам'яті або константа. При цьому повинна дотримуватися розрядність. Для **вказування розміру змінної** при записі до неї константи зарезервовані наступні ключові слова: byte, word, dword, qword. Розмір qword дозволений тільки в режимі x64. **Заборонено пересилання «mov змінна,змінна»**, тобто запис однієї створеної у пам'яті змінної до іншої.
Арифметичні команди:

- **Додавання та віднімання**:

add операнд1,операнд2
sub операнд1,операнд2

Результат додавання / віднімання поміщується до першого операнду (операнд1). Операнди повинні мати **однакову розрядність.** Додавати і віднімати можна як беззнакові, так і знакові числа. **Заборонено додавання / віднімання «add змінна,змінна» / «sub змінна,змінна»**, тобто обидва операнди в команді add / sub не можуть бути змінними одночасно. Приклади команд додавання і віднімання:

```
%include "io.inc"
section .text
global CMAIN
CMAIN:
    mov al,2
    mov bl,3
    add al,bl
    ret
```

```
%include "io.inc"
section .data
var db 45
section .text
global CMAIN
CMAIN:
    mov bl,3
    add [var],bl
    ret
```

```
%include "io.inc"
section .data
var dd 45
section .text
global CMAIN
CMAIN:
    mov ebx,3
    sub [var],ebx
    ret
```

```
%include "io.inc"
section .data
var1 dw 400
var2 dw 300
cnst equ 7
section .text
global CMAIN
CMAIN:
    add word[var1],cnst
    sub word[var2],7
    ret
```

- **Інкремент** (збільшення на 1) **та декремент** (зменшення на 1) операнду (вмісту регістра або змінної у пам’яті):

inc al
dec bx

Зміна знаку операнду:

neg cx

Інкремент, декремент та зміна знаку змінних у пам’яті:

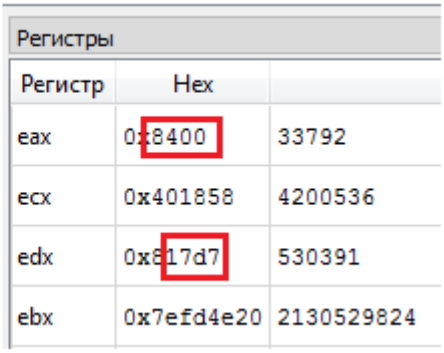
inc word[var1]
dec word[var2]
neg byte[var3]

- **Множення.** Синтаксис команди беззнакового та знакового множення:

mul операнд
imul операнд

Операнд, який вказаний у команді, є одним зі співмножників (уміст регістра або змінна у пам’яті). Якщо цей співмножник є змінною, то треба вказати її розмір (byte, word, dword), наприклад, word[var4].
Місцезнаходження другого співмножника та добутку фіксоване і в команді не вказується:
– якщо операнд команди mul має розмір db (байт), то другий співмножник береться з регістра al, а результат поміщується до регістра ax;
– якщо операнд команди mul має розмір dw (слово), то другий співмножник береться з регістра ax, а результат поміщується до регістрової пари dx:ax;
– якщо операнд команди mul має розмір dd (подвійне слово), то другий співмножник береться з регістра eax, а результат поміщується до регістрової пари edx:eax.
Один зі співмножників міститься в команді, другий повинен міститися в регістрі eax/ax/al, а результат поміщується до регістрів edx:eax/dx:ax/ax.
Приклад множення числа 20000 на число 20000:

```
%include "io.inc"
section .text
global CMAIN
CMAIN:
    mov ax,20000
    mov bx,20000
    mul bx
    ret
```

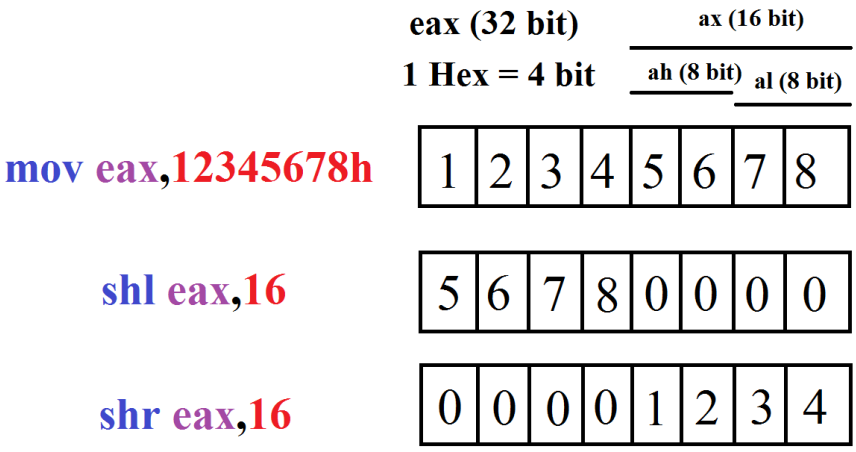


Результат (20000*20000=400000000=**17d78400h**):

Як скористатися результатом операції, якщо він розкиданий по декількох регістрах? Із частиною, що була поміщена до eax, просто – вона на своєму місці, у частині молодших бітів регістра eax (у регістрі ax), і відповідає молодшим розрядам числа. Для використання старших розрядів нашого результату, а також, для звернення до старших бітів регістрів, використовують **команду логічного зсуву бітів**:

SHL <операнд>, <кількість бітів> ; Логічний зсув уліво
SHR <операнд>, <кількість бітів> ; Логічний зсув управо

Наприклад,



Зрозуміло, що команди можна написати і в іншому вигляді, наприклад, шістнадцятковому: shl eax,10h *або двійковому* shl eax,10000b.
При логічному зсуві «вивільнені» біти заповнюються нулями.
Існують також інші зсуви (арифметичний, циклічний, розширений), які ми розглядати не будемо (від логічного вони відрізняються наповнювачем «вивільнених» бітів). У всіх зазначених зсувах останній вивільнений біт зберігається у прапорці CF.
Повернемося до прикладу із множенням:

```
1   %include "io.inc"
2   section .text
3   global CMAIN
4   CMAIN:
5       mov ebp, esp;
6       mov ax,20000
7       mov bx,20000
8       mul bx
9       shl edx,16
10      add eax,edx
11      ret
12
```

Регистр	Hex	Int
eax	0x17d78400	400000000
ecx	0x40185c	4200540
edx	0x17d70000	399966208
ebx	0x7efd4e20	2130529824
esp	0x28ff2c	0x28ff2c
ebp	0x28ff2c	0x28ff2c
esi	0x0	0
edi	0x0	0

Отже, результат множення переміщений до регістра eax і готовий до подальшого використання.

- **Ділення.** Синтаксис команди беззнакового та знакового ділення:

div операнд
idiv операнд

Цілочисельне ділення є діленням із остачею. Операнд, який вказаний у команді, є дільником (вміст регістра). Місцезнаходження подільного та результату фіксоване і в команді не вказується:
– якщо дільник має розмір db (байт), то подільне береться з регістра ax, частка поміщується до регістра al, остача – до ah;
– якщо дільник має розмір dw (слово), то подільне береться з регістрової пари dx:ax, частка поміщується до регістра ax, остача – до dx;
– якщо дільник має розмір dd (подвійне слово), то подільне береться з регістрової пари edx:eax, частка поміщується до регістра eax, остача – до edx.

```
%include "io.inc"
section .text
global CMAIN
CMAIN:
    mov ax,2000
    mov dx,0
    mov bx,200
    div bx
    ret
```

Регистр	Hex	
eax	0xa	10
ecx	0x40185c	4200540
edx	0x0000	524288
ebx	0x7efd00c8	2130510024
esp	0x28ff2c	0x28ff2c
ebp	0x28ff2c	0x28ff2c

Враховуючи зазначене вище, якщо дільник 32-/16-/8-розрядний, то подільне є відповідно 64-/32-/16-розрядним. Якщо реально подільне має розрядність дільника, то регістри відповідно edx/dx/ah потрібно заповнити нулями командою **mov reg,0** (у прикладі вище, **mov dx,0**), або командою **xor reg,reg**, де reg – ім'я регістра, який заповнюється нулями.

Логічні побітові (порозрядні, булівські) команди:
and, or, not, xor.
Синтаксис:
not операнд
and операнд1,операнд2
or операнд1,операнд2
xor операнд1,операнд2
*Приклад***:**
xor eax,eax
Команда **xor eax,eax** виконує тут ж саму дію, що й команда mov eax,0, утім займає меншу кількість пам'яті.

3.4 Програмування обчислювальних процесів

Обчислювальний (алгоритмічний) процес – реалізація алгоритму на комп'ютері. Виділяють три основних типи обчислювальних процесів: лінійні, розгалужені та циклічні.

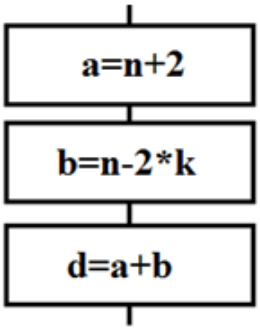
Для запису алгоритму можуть використовуватися різні форми його представлення. Однією з цих форм є **вербальна** – запис алгоритму у вигляді послідовності пронумерованих інструкцій. Ця форма запису є досить громіздкою і не дуже наглядною.

Іншою формою запису алгоритму є його запис у вигляді так званого **псевдокоду,** тобто за допомогою стандартного набору синтаксичних конструкцій. Псевдокод нагадує формальну мову програмування.

На поточний момент найбільш популярною у професійному середовищі формою запису алгоритму є запис його у вигляді **блок-схем**. Ця форма представлення реалізується у вигляді набору геометричних елементів (блоків). Кожний блок є **кроком** (окремою дією) **алгоритму.**

Лінійним (обчислювальним процесом лінійної алгоритмічної структури) називають обчислювальний процес, в якому операції виконуються послідовно у порядку їх запису. Кожна операція є самостійною і не залежить від будь-яких умов.

Приклад. Лінійний обчислювальний процес із трьох операцій:




```
1   %include "io.inc"
2   section .data
3   n equ 15 ;константа n
4   k equ 4 ;константа k
5   section .bss
6   a: resd 1 ;резервування змінної a
7   b: resd 1 ;резервування змінної b
8   d: resd 1 ;резервування змінної d
9   section .text
10  global CMAIN
11  CMAIN:
12      mov eax,n ;поміщуємо n до eax
13      add eax,2 ;n+2
14      mov [a],eax ;a=n+2
15      mov eax,k ;поміщуємо k до eax
16      mov ebx,2 ;поміщуємо 2 до ebx
17      mul ebx ;eax = 2*k
18      neg eax ;eax = -2*k
19      add eax,n ;eax = n-2*k
20      mov [b],eax ;b = n-2*k
21      mov eax,[a] ;поміщуємо a до eax
22      add eax,[b] ;eax = a+b
23      mov [d],eax ;d = a+b
24      ret
```

Лінійні процеси мають місце, наприклад, при обчисленні значень арифметичних виразів, коли наявні конкретні числові дані і над ними виконуються відповідні до умови задачі дії.
У мовах високого рівня для запису лінійних процесів використовують оператори присвоювання та арифметичні оператори.
У асемблері аналогом цьому є команда передавання даних (mov), арифметичні команди (add, sub, neg, mul, imul, div, idiv, cmp, inc, dec тощо), команди зсуву (shr, shl), команди співпроцесора для виконання дій над дійсними числами тощо.
Розгалужений процес (блок-схема зображена на рис. 3.3) – обчислювальний процес, в якому порядок виконання дій залежить від початкових умов або проміжних результатів. Кожний напрям обчислень у такому процесі є **гілкою обчислень**. Обчислювальний процес виконується тільки по одній гілці, вибір якої визначається перевіркою логічної умови.

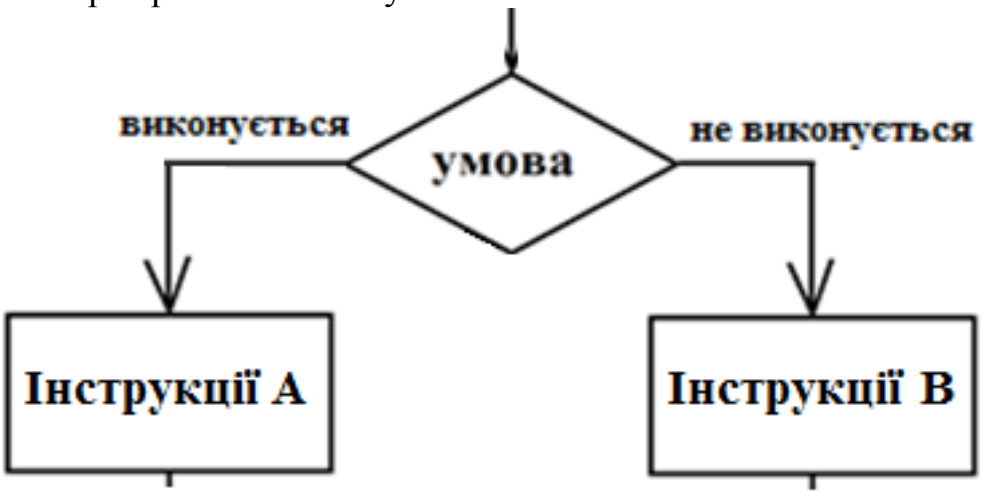


Рис. 3.3. Загальна блок-схема розгалуженого процесу

Реалізація умовних переходів. Усі логічні умови незалежно від їх складності **можна звести до трьох** найпростіших: **A>B, A<B та A=B**, або, враховуючи те, що універсальним способом порівняння величин у математиці є віднімання, – до A-B>0, A-B<0 та A-B=0. Наприклад, умова A≠B рівносильна умовам A>B та A<B, записаним у вигляді системи нерівностей. На цих властивостях, а також на використанні змін значень **бітів регістра прапорців** (встановлення / скидання), побудована робота із розгалуженнями в мові Assembler.
Виходячи із практичних міркувань зручності, три найпростіші умови розширюють до шістьох простих співвідношень: A>=B, A>B, A<=B, A<B, A=B та A≠B (інший запис A<>B).
Для реалізації умовних переходів у асемблері використовуються команди sub (або cmp) і команди умовних переходів, список яких буде наведений нижче.
Чим відрізняється команда sub від cmp? При виконанні будь-якої із команд sub або cmp виконується віднімання другого операнду від першого. Відмінність між цими командами полягає тільки в тому, що після виконання команди sub встановлюються / скидаються відповідні прапорці і до першого операнду записується різниця, а при виконанні команди cmp тільки встановлюються відповідні прапорці, а вміст першого операнду не змінюється.
Усі команди умовних переходів наведені в таблиці:

Команда	Перехід, якщо	Умова переходу
JZ/JE	нуль або дорівнює	ZF=1
JNZ/JNE	не нуль або не дорівнює	ZF=0
JC/JNAE/JB	є переповнення/не вище і не дорівнює/нижче	CF=1
JNC/JAE/JNB	немає переповнення/вище або дорівнює/не нижче	CF=0
JP	число одиничних біт парне	PF=1
JNP	число одиничних біт непарне	PF=0
JS	знак дорівнює 1	SF=1
JNS	знак дорівнює 0	SF=0
JO	є переповнення	OF=1
JNO	немає переповнення	OF=0
JA/JNBE	вище/не нижче і не дорівнює	CF=0 та ZF=0
JNA/JBE	не вище/нижче або дорівнює	CF=1 або ZF=1
JG/JNLE	більше/не менше і не дорівнює	ZF=0 и SF=OF
JGE/JNL	більше або дорівнює/не менше	SF=OF
JL/JNGE	менше/не більше і не дорівнює	SF≠OF
JLE/JNG	менше або дорівнює/не більше	ZF=1 або SF≠OF
JCXZ	уміст CX дорівнює нулю	CX=0

ПЕРЕВІРКА ПРОСТИХ СПІВВІДНОШЕНЬ:

1) Нестрога нерівність A>=B (або A-B>=0). Якщо A>=B, то виконується перехід на мітку M, якщо A<B – дія не потрібна. Використовуються **команди віднімання sub (або порівняння cmp) та умовного переходу jns (прапорець SF=0).**

Приклад:

```
1   %include "io.inc"
2   section .data
3   varA dd 12
4   varB dd 10
5   section .text
6   global CMAIN
7   CMAIN:
8       M: mov eax,[varA]
9       sub eax,[varB]
10      jns M
11      ret
```

АБО

```
1   %include "io.inc"
2   section .data
3   varA dd 12
4   varB dd 10
5   section .text
6   global CMAIN
7   CMAIN:
8       M: mov eax,[varA]
9       cmp eax,[varB]
10      jns M
11      ret
```

Коментар: якщо varA>=varB (як і є в наведеному прикладі), то після виконання 10-го рядка програми відбудеться перехід на рядок із міткою M (це 8-й рядок); якщо ж зробити операнд varA меншим за varB, то переходу не відбудеться і буде виконаний 11-й рядок.

2) Строга нерівність A>B (або A-B>0). Умова A>B перетворюється на B<A і використовуються **команди sub / cmp та умовного переходу js (прапорець SF=1).**

Приклад:

```
 8      M: mov eax,[varB]
 9      sub eax,[varA]
10      js M
11      ret
```

АБО

```
 8      M: mov eax,[varB]
 9      cmp eax,[varA]
10      js M
11      ret
```

Коментар: якщо varA>varB, то після виконання 10-го рядка програми відбудеться перехід на рядок із міткою M (це 8-й рядок); якщо varA<=varB, то перехід не відбудеться.

3) Нестрога нерівність A<=B (або A-B<=0). Реалізується аналогічно A>=B.

Приклад:

```
 8      M: mov eax,[varB]
 9      sub eax,[varA]
10      jns M
11      ret
```

АБО

```
 8      M: mov eax,[varB]
 9      cmp eax,[varA]
10      jns M
11      ret
```

4) Строга нерівність A<B (або A-B<0). Реалізується аналогічно A>B.

Приклад:

```
 8      M: mov eax,[varA]
 9      sub eax,[varB]
10      js M
11      ret
```

АБО

```
 8      M: mov eax,[varA]
 9      cmp eax,[varB]
10      js M
11      ret
```

5) Рівність A=B (або A-B=0). Використовуються **команди віднімання sub та умовного переходу jz (прапорець ZF=0).**

Приклад:

```
M: mov eax,[varA]
sub eax,[varB]
jz M
ret
```

АБО

```
M: mov eax,[varA]
cmp eax,[varB]
jz M
ret
```

(якщо операнди varA і varB поміняти місцями, результат не зміниться).

6) Нерівність $A \neq B$ (інший запис A<>B). Повністю аналогічно A=B, але використовується команда *jnz (прапорець ZF=1).*

Команда безумовного переходу jmp.

Синтаксис команди безумовного переходу:

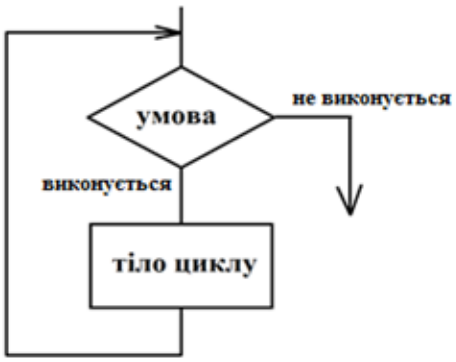
jmp операнд

Операнд є адресою переходу. Залежно від способу вказування цієї адреси, розрізняють прямий і непрямий безумовний переходи.

Прямий безумовний перехід (у команді вказується мітка, на яку треба перейти):

```
jmp M
...
```

M: ret
Непрямий безумовний перехід (у команді вказується регістр або комірка пам’яті, де знаходиться адреса переходу):
jmp ebx
Циклічний процес – обчислювальний процес, при якому група дій багатократно повторюється доки не виконається вказана умова. Згадана група дій називається циклом.
Отже, **цикл** – фрагмент коду, який виконує певні дії декілька разів. Кожне виконання послідовності інструкцій (**тіла циклу**) називається **ітерацією**.
Існує певна класифікація циклічних обчислювальних процесів, зокрема, цикли з передумовою, цикли з післяумовою, цикли з лічильником та комбіновані цикли (різні варіації вищенаведених циклів).
Цикл з передумовою. Цикл такого вигляду виконується доти, доки є вірною умова, яка перевіряється кожний раз перед початком виконання тіла циклу. Якщо ця умова буде хибною із самого початку, то тіло циклу не буде виконане жодного разу.
Приклад. Програма, яка виводить у циклі на екран числа від 1 до 10:




```
1   %include "io.inc"
2   section .data
3   n dd 10
4   section .text
5   global CMAIN
6   CMAIN:
7       mov edi,0; очищення edi
8       cycle_begin:
9       ;перевірка умови:
10      mov esi,edi
11      sub esi,[n]
12      jns cycle_end
13      ;тіло циклу:
14      inc edi
15      PRINT_DEC 4,edi
16      NEWLINE
17      jmp cycle_begin
18      cycle_end:
19      ret
```

```
Ввод

Вывод
1
2
3
4
5
6
7
8
9
10
```

Нескладно зрозуміти, що змінюючи змінну n, можна виводити на екран довільну кількість натуральних чисел від 1 до n, де n>=1.
Цикл з післяумовою. У циклі такого вигляду умова перевіряється після виконання тіла циклу. Отже, цикл з післяумовою буде завжди виконаний хоча б один раз незалежно від умови.
У різних мовах програмування високого рівня вихід із циклу може відбуватися за різного результату перевірки умови, тобто в одних мовах вихід із циклу може

бути за істинності умови, в інших – за хибності; у мовах низького рівня, зокрема, в асемблері, можна зробити як завгодно.


```
1   %include "io.inc"
2   section .data
3   n dd 10
4   section .text
5   global CMAIN
6   CMAIN:
7       mov edi,0; очищення edi
8       cycle_begin:
9       ;тіло циклу:
10      inc edi
11      PRINT_DEC 4,edi
12      NEWLINE
13      ;перевірка умови:
14      mov esi,edi
15      sub esi,[n]
16      js cycle_begin
17      ret
18
```

Ввод

Вывод

```
1
2
3
4
5
6
7
8
9
10
```

Цикл з лічильником можна організувати так само, як і попередні два цикли, використовуючи умовні та безумовні переходи, але існують спеціальні команди для цього, а саме, loop / loope / loopne. Ці команди самостійно зменшують його значення після кожної ітерації циклу і порівнюють після зменшення з нулем. Команди loope / loopne дозволяють вийти з циклу за додатковою умовою.

```
1   %include "io.inc"
2   section .data
3   n dd 10
4   section .text
5   global CMAIN
6   CMAIN:
7       mov ecx,[n]
8       mov edi,0
9       cycle_begin:
10      inc edi
11      PRINT_DEC 4,edi
12      loop cycle_begin
13      ret
14
```

Ввод

Вывод

```
12345678910
```

Команда **loop мітка** виконує такі дії:
– зменшує регістр ecx;
– порівнює вміст регістра ecx із нулем і, якщо він >0, то відбувається перехід на зазначену мітку.
Команда loop реалізує цикл з післяумовою. Перехід loop має тип short. Якщо здійснюється дуже далекий перехід, то з’явиться помилка «short jump is out of range». У цьому випадку потрібно скористатися іншим способом створення

циклу, без команди loop. Узагалі, вбудована **команда loop є далеко не найкращою** для програмування циклів.

3.5 Масиви

Масив – це упорядкований набір даних, доступ до яких здійснюється за допомогою **ключа**, який однозначно ідентифікує елементи всередині масиву (один масив не може містити елементів із однаковими ключами). Тобто, з математичної точки зору, масив є *відображенням*, яке встановлює взаємно однозначну відповідність між множиною значень і множиною ключів.

Масиви корисні при роботі із великою кількістю даних (наприклад, обробці результатів запитів до баз даних), або із групами значень, пов'язаних між собою.

Кожне окреме значення в масиві називається **елементом масиву**. Масиви можуть містити довільну **кількість значень**, зокрема, не містити жодного. Кількість елементів у масиві називають **довжиною масиву**.

Одновимірні масиви також називають **векторами**, двовимірні масиви – **матрицями**.

Масив у асемблері представлений як **багатозначна змінна**, тобто звичайна змінна у пам'яті, у відповідність якій поставлене не одне, а декілька значень.

Оскільки типізація даних в асемблері є досить умовною і визначається, головним чином, їх розміром, а не якісною складовою, то як такого **поділу масивів на числові, рядкові тощо не існує**.

Нескладно зрозуміти, що для того, щоб працювати із масивами, потрібно по-перше, їх визначити (описати), по-друге – мати змогу до них звертатися.

Створення одновимірних масивів. Описуються (створюються) масиви за директивами визначення даних:

– за допомогою багатозначної змінної звичайним способом:

```
1   %include "io.inc"
2   section .data
3   array1 dw 4,'string',12,81 ;визначення масиву шляхом перелічення
4                              ;його елементів, кожен з яких має розмір
5                              ;слова, а ідентифікатором масиву є
6                              ;багатозначна змінна array1
7
8   section .bss
9   array2: resd 10 ;створення масиву із 10 неініціалізованих
10                  ;подвійних слів
```

– за допомогою префіксу «times», який змушує інструкцію асемблюватися декілька разів:

```
1   %include "io.inc"
2   section .data
3   array3 times 7 dd 3 ;створення ініціалізованого масиву, який складається із 7-х
4                       ;елементів, значення кожного з яких дорівнює 3,
5                       ;розмір кожного - подвійне слово
6
7   section .bss
8   array4: times 10 resd 1 ;створення масиву із 10 неініціалізованих
9                           ;подвійних слів
```

Принципової різниці між інструкціями «times 10 resd 1» та «resd 10» не має.

Звернення до елементів одновимірних масивів. У якості *ключа* в асемблері виступає не індекс, як зазвичай у мовах програмування високого рівня, а **адреса елемента**. *Звернення до елемента (виклик елемента) масиву* здійснюється шляхом вказування його *адреси* (яка є *адресою початку* масиву) та *зміщення* елемента в масиві.

Адреса елемента масиву записується за наступною формулою:

array_address (i) = [array_name + array_type*i]

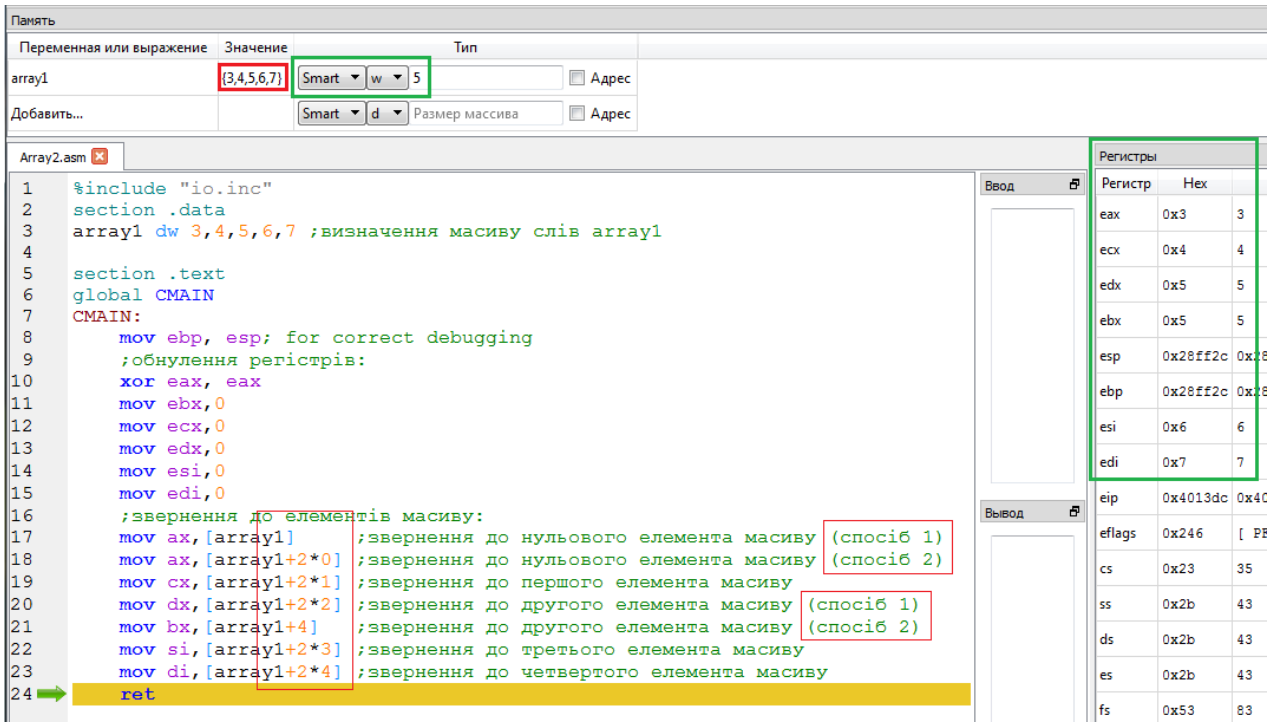
У цій формулі:

array_name – ім'я змінної, якій присвоєний масив;

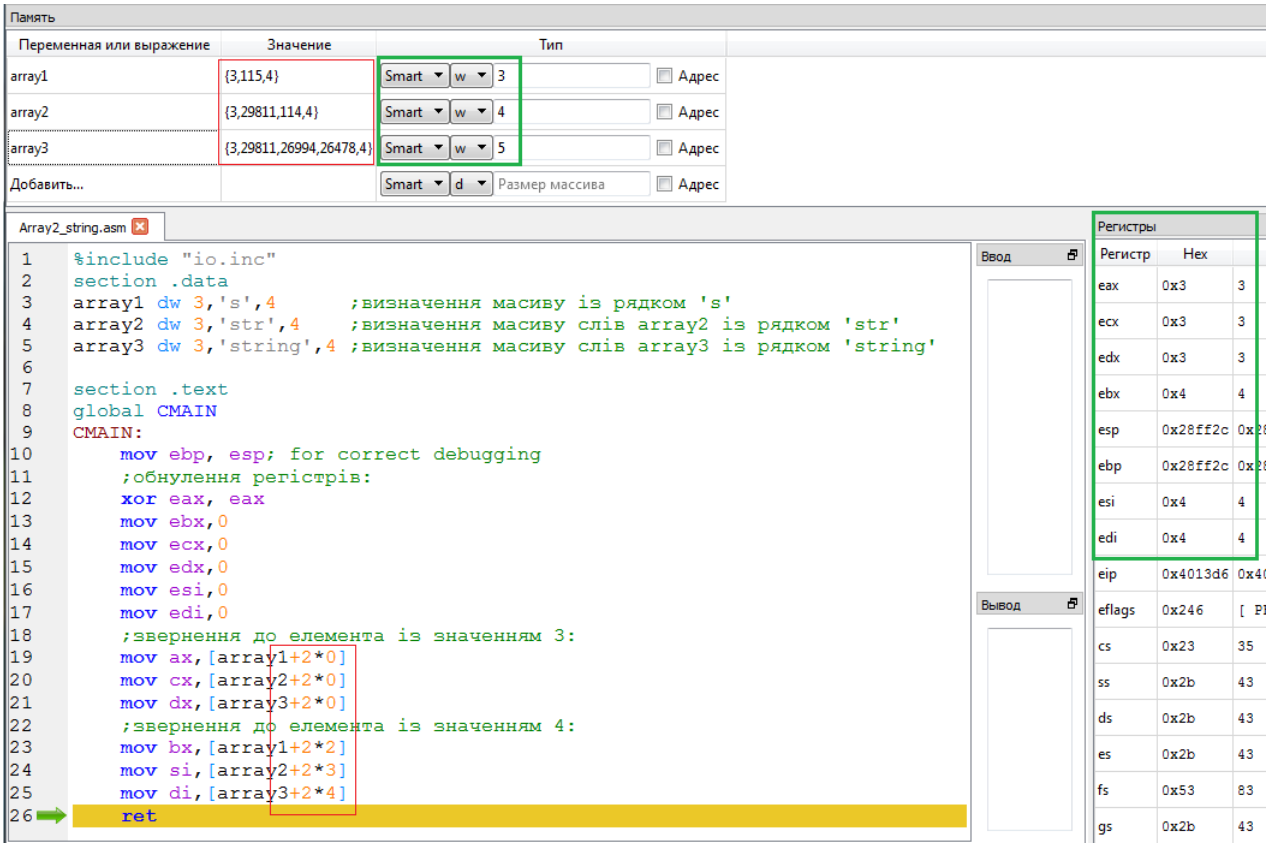
array_type – стала, яка відповідає типу даних і дорівнює: 1 – якщо змінна «array_name» має розмір байт (db), 2 – якщо розмір «array_name» дорівнює слову (dw), 4 – якщо розмір «array_name» дорівнює подвійному слову (dd), 8 – якщо розмір «array_name» дорівнює почетвереному слову (dq);

i – порядковий номер елемента масиву, починається з нуля.

Приклад обчислення адрес масивів:



Якщо масив містить тільки числові значення, то розрахунок порядкового номера є простим, як і свідчить попередній приклад. Якщо в масиві містяться рядкові значення, то задача звернення до потрібного елемента ускладнюється (потрібно знати, скількома елементами масиву записується рядок), наприклад:



Створення двовимірного масиву. Оскільки реальна пам’ять комп’ютера є (за своєю суттю) одновимірним масивом, то масиви розміщуються в «неперервній» області пам’яті, а для звернення до елемента масиву, який має декілька індексів, призначають відповідність між послідовними комірками пам’яті та елементами масиву, тобто створюють взаємно однозначне відображення.

Для створення двовимірного масиву потрібно виділити пам’ять так само, як і для одновимірного, – за допомогою **багатозначної змінної**, наприклад, створимо матрицю розмірності 5*3:

```
%include "io.inc"
section .data
array1 dw 1,2,3
       dw 4,5,6
       dw 7,8,9
       dw 10,11,12
       dw 13,14,15
```

АБО

```
%include "io.inc"
section .data
array1 dw 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
```

Обидва записи повністю еквівалентні з точки зору комп’ютера – вони визначають виділення однієї і тієї ж самої пам’яті, – але перший є більш наглядним і зрозумілим для людини.

Звернення до елементів двовимірного масиву. Вище було сказано про взаємно однозначне відображення між множиною елементів масиву та послідовних комірок пам'яті. Для роботи із двовимірними масивами нам знадобиться функція цього відображення для випадку двовимірного масиву.

Розглянемо **прямокутну матрицю розміру (m*n)**, у якої **(i,j)-й елемент дорівнює** a_{ij}**:**

$$\begin{pmatrix} a_{00} & \cdots & a_{o(n-1)} \\ \vdots & \ddots & \vdots \\ a_{(m-1)0} & \cdots & a_{(m-1)(n-1)} \end{pmatrix}$$

Тоді **адреса елемента двовимірного масиву** записується за наступною формулою:

array_address (i,j) = [array_name + array_type*(n*i + j)]

У цій формулі array_name та array_type визначаються аналогічно до відповідної формули адреси елементів одновимірних масивів.

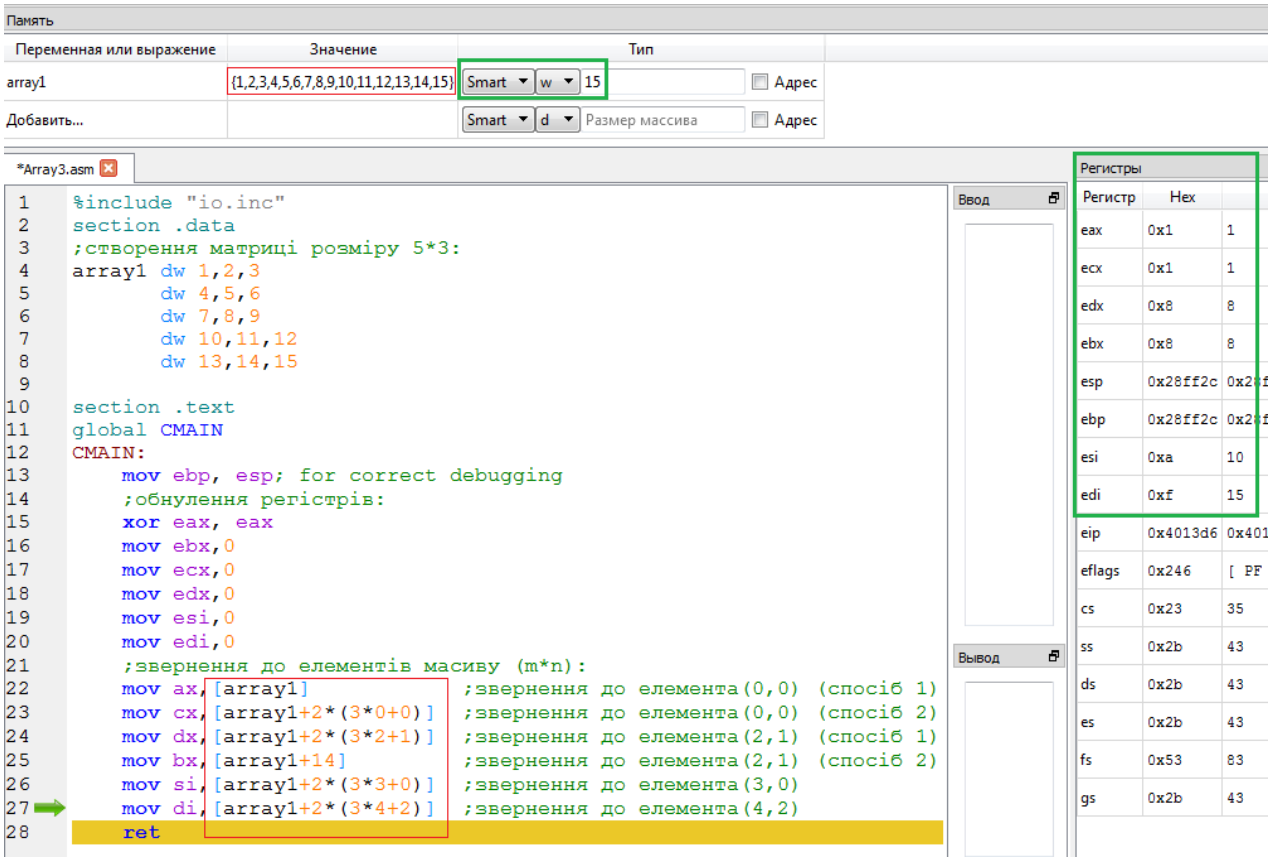
Приклад. Розглянемо матрицю розмірності 5*3:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{pmatrix}$$

Нехай кожен елемент цієї матриці має розмір слова. При цьому, нумерація елементів є наступною:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \\ a_{40} & a_{41} & a_{42} \end{pmatrix}$$

Тоді адреси масивів обчислюються наступним чином:



ПРИКЛАД. ПРОГРАМА, ЯКА ПІДНОСИТЬ ЦІЛІ ЧИСЛА ДО НАТУРАЛЬНОГО СТЕПЕНЯ, ОБЧИСЛЮЄ ЧАСТИННІ СУМИ РЯДУ

$$\sum_{k=1}^{n} a^n$$

ТА ВИВОДИТЬ РЕЗУЛЬТАТИ ОБЧИСЛЕНЬ У ЦИКЛІ ЗА ДОПОМОГОЮ МАСИВІВ.

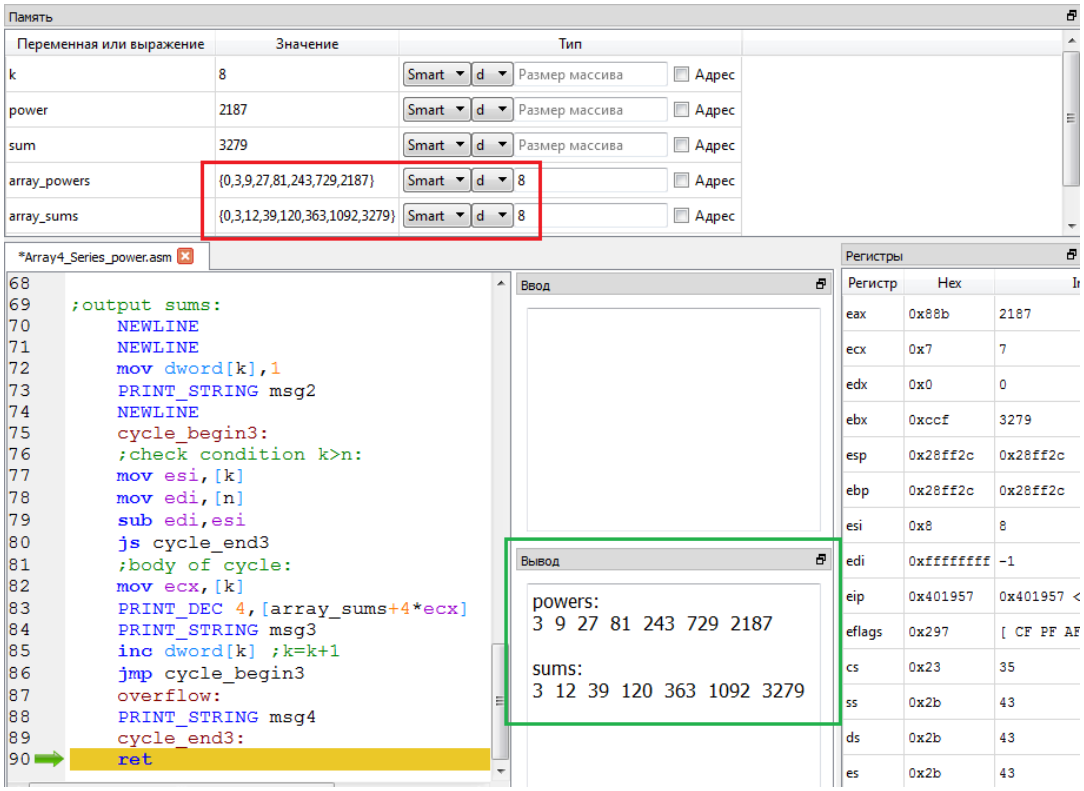
а) Лістинг:

```
%include "io.inc"
section .data
a dd 3
n dd 7
k dd 1
power dd 1
sum dd 0
msg1 dd 'powers:',0
msg2 dd 'sums:',0
msg3 dd '  ',0
msg4 dd 'overflow power or sum!'

section .bss
array_powers resd 100
array_sums resd 100

section .text
global CMAIN
CMAIN:
    xor eax,eax ;clear of eax
    mov ebx,0 ;clear of ebx
    mov ecx,0 ;clear of ecx
    mov edi,0 ;clear of edi
    mov esi,0 ;clear of esi

;calculating and create arrays:
    cycle_begin1:
    ;check condition k>n:
    mov esi,[k]
    mov edi,[n]
    sub edi,esi
    js cycle_end1
    ;body of cycle:
    mov eax,[power]
    mul dword[a]
    js overflow ;owerflow contol (variable power)
    jo overflow ;owerflow contol (variable power)
    mov [power],eax ;power=power*a
    mov ebx,[sum]
    add ebx,eax
    jo overflow ;owerflow contol (variable sum)
    mov [sum],ebx ;sum=sum+power
    mov ecx,[k]
    mov [array_powers+4*ecx],eax ;save array_powers
    mov [array_sums+4*ecx],ebx   ;save array_sums
    inc dword[k] ;k=k+1
    jmp cycle_begin1
    cycle_end1:
```

```
50  ;output powers:
51      mov dword[k],1
52      PRINT_STRING msg1
53      NEWLINE
54      cycle_begin2:
55      ;check condition k>n:
56      mov esi,[k]
57      mov edi,[n]
58      sub edi,esi
59      js cycle_end2
60      ;body of cycle:
61      mov ecx,[k]
62      PRINT_DEC 4,[array_powers+4*ecx]
63      PRINT_STRING msg3
64      inc dword[k] ;k=k+1
65      jmp cycle_begin2
66      cycle_end2:
67
```

```
68  ;output sums:
69      NEWLINE
70      NEWLINE
71      mov dword[k],1
72      PRINT_STRING msg2
73      NEWLINE
74      cycle_begin3:
75      ;check condition k>n:
76      mov esi,[k]
77      mov edi,[n]
78      sub edi,esi
79      js cycle_end3
80      ;body of cycle:
81      mov ecx,[k]
82      PRINT_DEC 4,[array_sums+4*ecx]
83      PRINT_STRING msg3
84      inc dword[k] ;k=k+1
85      jmp cycle_begin3
86      overflow:
87      PRINT_STRING msg4
88      cycle_end3:
89      ret
```

б) Результати та відлагодження:



3.6 Апаратний стек

Для деяких операцій, наприклад, для роботи з підпрограмами, апаратними перериваннями тощо, МП використовує стек. Розглянемо, що це таке?

Узагалі, розрізняють апаратний і програмний стеки. **Програмний стек** є користувальницькою моделлю даних.

Надалі під словом «стек» ми будемо розуміти виключно апаратний стек.

Апаратний стек – це область оперативної пам'яті, призначена для тимчасового зберігання даних. Стек адресується за допомогою пари регістрів SS:ESP. Регістр SS (Stack Segment Selector) містить селектор сегмента стека при роботі МП в захищеному режимі, ESP (Stack Pointer, вказівник стека) – зміщення відносно базової адреси сегмента стека.

Стек працює за принципом «неправильної черги» **LIFO (Last Input – First Output)**: «останнім прийшов – першим вийшов». Дані, які були уштовхнуті до стека останніми, будуть виштовхнуті першими. «Правильною чергою» є FIFO (First Input — First Output; «першим прийшов – першим вийшов»).
МП має дві команди для роботи зі стеком – **push** (уштовхування) та **pop** (виштовхування).

Рис. 3.4. Апаратний стек

Стек росте у пам’яті «вниз», тобто нова порція даних записується за меншою адресою. Утім, для програмування точна адреса даних всередині стека не має значення, оскільки будь-яка операція над стеком має справу з його вершиною, на яку завжди вказує регістр SP (ESP). Запис і зчитування відбуваються словами (по 2 байти).
Номери адрес при записі до стека зростають у бік молодших адрес ОЗП, при зчитуванні – навпаки (рис. 3.4):
– при записі адреса вершини стека зменшується на 2;
– при зчитуванні адреса вершини стека збільшується на 2.

3.7 Підпрограми

Підпрограма – окрема функціонально незалежна частина програми, яка можна багаторазово використовуватися протягом виконання основної програми. По суті, підпрограма – це одна програма всередині іншої програми.
За способом створення підпрограми бувають двох видів – **стандартні** підпрограми та **підпрограми, створені користувачем (користувальницькі)**.
Підпрограми використовують для:

- уникнення потреби багаторазово повторювати однакові фрагменти в тексті програми;
- покращення структури програми та полегшення її розуміння;
- підвищення стійкості до помилок програмування при модифікації програми;
- економії пам'яті, оскільки пам'ять для збереження змінних, які використовуються у підпрограмах, виділяється тільки на час роботи підпрограми (звичайно, якщо ці змінні не оголошені, як глобальні).

Підпрограми існують у різних мовах програмування, але мають відмінності у назвах, синтаксисі тощо, зокрема, в асемблері їх, як правило, називають **підпрограмами**, в C – **функціями**, в мові Pascal існують два види підпрограм – **функції** та **процедури**.
Приклад підпрограми, яка реалізує додавання двох чисел (результат поміщується до регістра eax):

```
1   %include "io.inc"
2   section .data
3   b dd 7
4   subpr:
5   add eax,[b]
6   ret
7   section .text
8   global CMAIN
9   CMAIN:
10      mov ecx,5
11      call subpr
12      ret
```

Синтаксис підпрограм у NASM має наступний вигляд:

- (звичайна) мітка
- команди підпрограми
- інструкція retn (ret; return near - ближнє повернення) або retf (return far – дальнє повернення).

Викликаються підпрограми за допомогою інструкції call.
Зрозуміло, що перед виконанням підпрограми потрібно передати їй операнди з основної програми (звичайно, якщо такі існують).
При виконанні команди **call** відбуваються наступні дії:
– поміщується до стека адреса повернення на наступну після call команду;

– поміщується до регістра IP/EIP/RIP або до регістрової пари CS:IP адреса підпрограми і після цього виконується безумовний перехід на цю адресу. Потім виконуються команди підпрограми до команди ret (retn, retf).
При виконанні команди **ret (retn, retf)** відбуваються такі дії:
– поміщується з вершини стека адреса повернення до регістра IP/EIP/RIP або до регістрової пари CS:IP;
– здійснюється безумовний перехід на встановлену в IP/EIP/RIP адресу, з якої продовжується виконання основної програми.
Розміщення підпрограм у коді програми. В асемблері підпрограми можуть розміщуватися в різних місцях програми, зокрема, перед сегментом коду, після сегмента коду, всередині сегмента коду.
Головною вимогою розміщення підпрограми є **виключення можливості передачі на неї керування** без використання інструкції виклику call. Крім цього, **підпрограму не можна розміщувати в області неініціалізованих даних (section .bss).**
Приклади розміщення підпрограми в тексті основної програми:

перед кодом

```
1   %include "io.inc"
2
3   section .bss
4   c: resd 1
5
6   section .data
7   b dd 7
8
9   subpr:
10  add eax,[b]
11  mov [c],eax
12  ret
13
14  section .text
15  global CMAIN
16  CMAIN:
17      xor eax,eax
18      mov eax,5
19      call subpr
20      ret
```

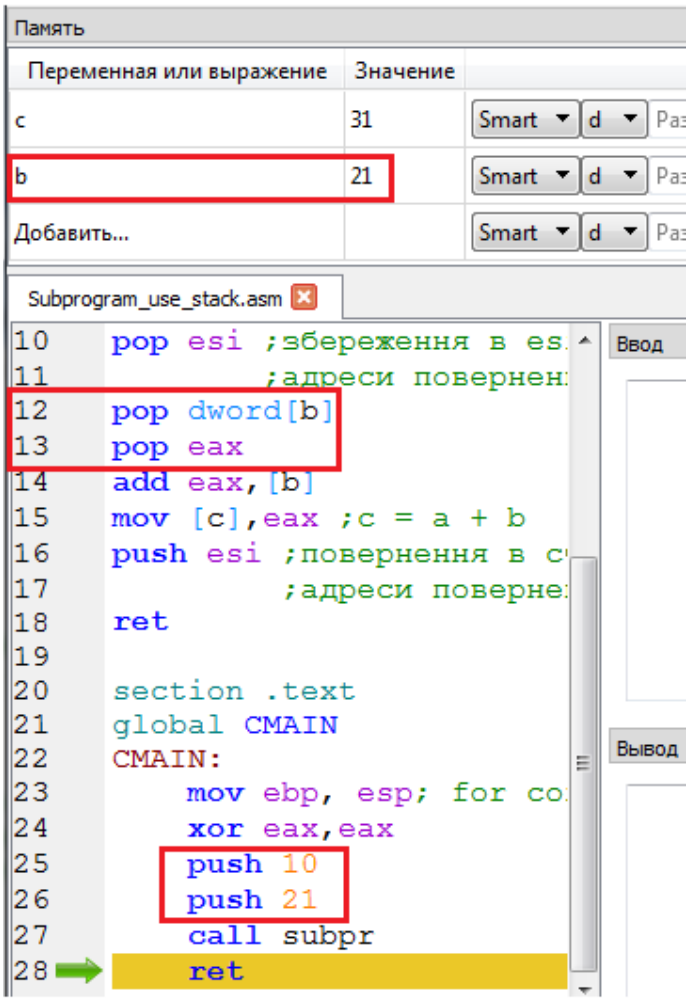
після коду

```
1   %include "io.inc"
2
3   section .bss
4   c: resd 1
5
6   section .data
7   b dd 7
8
9   section .text
10  global CMAIN
11  CMAIN:
12      xor eax,eax
13      mov eax,5
14      call subpr
15      ret
16
17      subpr:
18      add eax,[b]
19      mov [c],eax
20      ret
```

усередині коду

```
1   %include "io.inc"
2
3   section .bss
4   c: resd 1
5
6   section .data
7   b dd 7
8
9   section .text
10  global CMAIN
11  CMAIN:
12      xor eax,eax
13      mov eax,5
14      jmp subprogram
15          subpr:
16      add eax,[b]
17      mov [c],eax
18      ret
19      subprogram:
20      call subpr
21      ret
```

Передача параметрів підпрограмі. Параметри підпрограмі можна передавати через регістри загального призначення командою mov, але регістрів мало, а параметрів може бути багато. У цьому випадку можна передавати параметри через стек. Розглянемо приклад:



Перша команда pop (pop dword[b]) видалить зі стека адресу повернення, записану туди командою call, тому її потрібно зберегти, а потім повернути, для чого й потрібні команди: pop esi та push esi.

Виймаються параметри зі стека в оберненому порядку, тобто якщо виконувалися команди push 10, push 21, то після виконання у підпрограмі pop dword[b], pop eax, буде: b = 21; eax = 10.

Збереження вмісту регістрів. Крім адресування та передачі параметрів підпрограмі, стек може використовуватися й для інших цілей. Під час виконання підпрограма може змінювати значення регістрів, або змінних у пам'яті для збереження своїх проміжних результатів. Зазвичай або всі, або частина цих результатів не потрібні, а потрібні ті значення, які були до виконання підпрограми. Отже, останні потрібно перед виконанням підпрограми зберегти, а після виконання – відновити. Для цього використовується стек.

У NASM сегмент стека визначати не потрібно.

Для уштовхування / виштовхування до стека одночасно всіх 16-розрядних регістрів загального призначення використовується команда **pusha / popa**; для 32-х розрядних – **pushad / popad.** Для уштовхування / виштовхування до стека одночасно всіх 16-розрядних регістрів прапорців використовується команда **pushf / popf**; для 32-х розрядних – **pushfd / popfd.** У 64-розрядному режимі команди переміщення всіх регістрів до стека усунені.

Приклад:

```
%include "io.inc"
section .bss
c: resd 1
section .data
b dd 7
subpr:
push eax
add eax,[b]
mov [c],eax
pop eax
ret
section .text
global CMAIN
CMAIN:
    xor eax,eax
    mov eax,5
    call subpr
    ret
```

У наведеному прикладі регістр eax як перед виконанням підпрограми subpr, так і після її виконання буде містити значення 5. Результат додавання буде поміщений до змінної c.
Команда mov eax,5 реалізує передачу аргументів підпрограмі.
Змінна c визначена перед змінною b внаслідок двопрохідності асемблера NASM (конструкція resd є критичною); її також можна визначити після CMAIN.

3.8 Макроси, їх відмінності від підпрограм

Наскільки важливу роль відіграють в асемблері макроси, помітити зовсім не складно. Коли при роботі із SASM ми вибираємо «Создать новый проект», то відкривається вікно:

```
%include "io.inc"

section .text
global CMAIN
CMAIN:
    ;write your code here
    xor eax, eax
    ret
```

в якому присутні одразу чотири макроси:
- директива %include асемблера NASM;
- директива section асемблера NASM;
- директива global асемблера NASM;
- макрос CMAIN середовища SASM.

Макровизначення (макрос) – символічне ім'я, яке замінює декілька команд асемблера. Макроси можуть містити в собі інструкції, мітки, директиви, виклики інших макросів.

Макроси та підпрограми. Незважаючи на те, що і підпрограми, і макроси використовуються для тих самих цілей – заміни повторюваного декілька разів коду, – між ними є суттєві відмінності.

Підпрограма на протязі виконання програми компілюється один раз, її еквівалент у машинному коді тільки один раз з'являється в готовій програмі. За наявності **макросу,** транслятор підставляє в місце кожного входження макросу до програми, відповідний йому блок коду, отже, макрос буде компілюватися стільки разів, скільки він присутній у програмі. Відповідно, розмір кінцевого файлу збільшуватиметься від кожного наступного входження макросу.

Таким чином, **при використанні макросів ми, як правило, маємо виграш у швидкодії,** – за рахунок відсутності потреби у виклику (команда call) та поверненні (команда ret), а також відсутності «провантаження» стека на 2 байти, оскільки під час виклику підпрограми, до стека поміщується адреса повернення на наступну після call команду. **При використанні підпрограм – зазвичай, виграш у меншому розмірі кінцевого файлу**, за виключенням того випадку, коли замінюваний макросом повторюваний код містить лише невеличку кількість команд (три-чотири), що компенсує збільшення коду файлу від використання call, ret і стека у випадку застосування підпрограми.

Підпрограми, на відміну від макросів, економлять використання **пам'яті для збереження змінних**, що використовуються в них, оскільки зазначена пам'ять виділяється тільки на час роботи підпрограми (звичайно, якщо змінні не оголошені, як глобальні).

Здебільшого історично, склалося так, що у мовах високого рівня широко поширеним є використання підпрограм, у мовах низького рівня – макросів.

Особливості компіляції макросів. Препроцесор макромови (макропроцесор, макрогенератор) – комп'ютерна програма, яка готує початковий код для компіляції, тобто перетворює його до **пре**процесорної форми (**pre**processor form), придатної для обробки (to process) компілятором. Зокрема, препроцесор виконує **макропідстановку (макророзширення** початкового коду), тобто замінює макроси на символічні конструкції мови програмування. Входження імені макросу до початкової програми називають **макровикликом.**

Наприклад, препроцесор NASM замінює макрос:

```
PRINT_DEC 4, eax
```

наступним кодом:

```
section .data
format db "%d", 0
section .text
push eax
push format
call printf
add esp, 8
```

Оголошення макросу починається зі знака процента (%). У загальному випадку, імена макросів чутливі до регістра символів. Для створення нечутливих до регістра макросів, директиви повинні починатися з «%i», наприклад, %idefine замість %define.
Макроси можуть оголошуватися в будь-якому місці програми, в будь-якому її сегменті (області), але до їх виклику. Також макроси (у вигляді бібліотек макросів) можуть бути описаними в зовнішньому файлі. Зокрема, до складу SASM входять дві стандартні бібліотеки макросів io.inc та io64.inc (відповідно для 32-х і 64-х розрядного режиму роботи), розташовані за адресою ...\SASM\include. У цих бібліотеках, зокрема, містяться кросплатформні команди введення / виведення і макроси: CMAIN – точка входу та CEXTERN для доступу до зовнішніх команд на мові С.
Макроси бувають однорядкові і багаторядкові.
Однорядкові макроси визначаються за допомогою директиви препроцесора **%define**. Приклад створення та виклику макросу, який обчислює середнє арифметичне двох цілих чисел:

```
1   %include "io.inc"
2
3   %define average(a,b) (((a)+(b))/2)
4
5   section .text
6   global CMAIN
7   CMAIN:
8       mov eax,average(14,12)
9       ret
```

Примітка: якщо результат не є цілим числом, то до eax поміщується його ціла частина.
За допомогою директиви %define можна також **визначати константи**:

```
1   %include "io.inc"
2
3   %define average(a,b) (((a)+(b))/2)
4   %define cnst 60
5
6   section .text
7   global CMAIN
8   CMAIN:
9       mov eax,average(14,cnst)
10      ret
```

Директива %assign є ще одним способом оголосити однорядковий макрос. Цей макрос не повинен мати параметрів, а результатом його виконання повинно бути число. Директива %assign зазвичай використовується у складних макросах.
Приклад. Збільшення макросом incr числового значення іншого макросу на 1:

```
%include "io.inc"

%define cnst 60
%assign incr (cnst+1)

section .bss
var resd 1

section .text
global CMAIN
CMAIN:
    mov dword[var],incr
    ret
```

Багаторядкові макроси. Для визначення макросів, які складаються з декількох рядків, використовуються директиви **%macro** та **%endmacro**. Синтаксис директиви %macro наступний:

%macro <ім'я_макросу> <кількість_параметрів>

Приклад. Програма, яка обчислює дискримінант квадратного рівняння $ax^2 + bx + c = 0$ за його параметрами a, b, c, яка використовує макрос:

```
%include "io.inc"

%macro quad_equ 3
  section .data
    msg_noq dd 'equation is not quadratic (a = 0)',0
    msg_d dd 'D=',0

  section .text
    mov eax,%1
    sub eax,0
    jz endm_noq
    mov eax,(%2*%2-4*%1*%3)
    PRINT_STRING msg_d
    PRINT_DEC 4, eax
    jmp endm
    endm_noq:
    PRINT_STRING msg_noq
    endm:
%endmacro


section .text
global CMAIN
CMAIN:
    quad_equ 1,2,3
    ret
```

Загальна структура багаторядкового макросу NASM:

%macro <ім'я_макросу> <кількість_параметрів>

<тіло_макросу>
%endmacro

У багаторядкових макросах (внаслідок того, що NASM є двопрохідним асемблером) не може використовуватися **префікс times**, оскільки він обробляється після повного розгортання цих макросів.
Недоліком визначення макросу із наведеного вище прикладу є те, що використати в одній програмі його можна тільки один раз, оскільки змінні та мітки в ньому не є локальними, а отже, після макропідстановки виявиться, що відбулося їх дублювання, що призведе до помилки.
Локальні мітки та змінні для макросу створюються за допомогою префіксу %%. Наприклад, для того, щоб використати два або більше разів один і той самий макрос, визначений у попередньому прикладі, його слід переписати наступним чином:

```
1   %include "io.inc"
2
3   %macro quad_equ 3
4   section .data
5       %%msg_noq dd 'equation is not quadratic (a = 0)',0
6       %%msg_d dd 'D=',0
7
8     section .text
9       mov eax,%1
10      sub eax,0
11      jz %%endm_noq
12      mov eax,(%2*%2-4*%1*%3)
13      PRINT_STRING %%msg_d
14      PRINT_DEC 4, eax
15      jmp %%endm
16      %%endm_noq:
17      PRINT_STRING %%msg_noq
18      %%endm:
19  %endmacro
20
21  section .text
22  global CMAIN
23  CMAIN:
24      quad_equ 1,2,3
25      NEWLINE
26      quad_equ 5,7,9
27      ret
```

```
Ввод

Вывод
D=-8
D=-131
```

Використання макросів. Умовна компіляція. Під **умовною компіляцією (умовним асемблюванням)** розуміють те, що залежно від умови буде відкомпільований (асембльований) той чи інший блок коду:

%if<умова_1>
; код, який асемлюватиметься тільки за виконання <умови_1>
%elif<умова_2>
; код, який асемлюватиметься тільки якщо умова_1 хибна, а умова_2 істинна
%else
; код, який асемлюватиметься якщо обидві вказані умови хибні
%endif

Директиви **%elif** та **%else** необов'язкові.

Перевірка оголошення макросу.

Для перевірки оголошення макросу використовуються директиви **%ifdef, %ifndef**. Інструкції, які входять до блоку **%ifdef**, будуть виконані за умови існування макросу.

Інструкції, які входять до блоку **%ifndef**, будуть виконані за умови, якщо макросу не існує.

```
%include "io.inc"
section .text
global CMAIN
CMAIN:
    %define macro
    %ifdef macro
        mov eax,7
    %endif
    ret
```

Використання макросів із зовнішніх файлів. Вставлення макросів, записаних у зовнішньому файлі, здійснюється за допомогою директиви **%include** (ім'я зовнішнього файлу пишеться у подвійних лапках).

Приклад, **%include "io.inc"**.

Директиви NASM (наприклад, section, global, extern) – це **макроси, визначені самим компілятором**.

До макросів, визначених компілятором належать і **псевдо-інструкції**, які не є реальними інструкціями МП, а є засобами, що полегшують роботу з асемблером.

Приклади псевдо-інструкцій: db, dw, dd, dq, resb, resw, resd, resq, rest, equ, префікс times.

Порівняно із TASM та MASM, асемблер NASM має відносно небагато директив.

3.9 Кодування дійсних чисел

Для роботи з дійсними числами у комп'ютерах використовується співпроцесор, який є модулем, що працює із числами з плаваючою крапкою.

Довільне дійсне число можна представити у вигляді числа з плаваючою крапкою, яке можна потім записати у двійковому коді, використовуючи стандарт **IEEE 754**, розроблений організацією IEEE (Institute of Electrical and Electronics Engineers).

Отже, перш ніж вивчати співпроцесор, слід розглянути спосіб представлення дійсних чисел у пам'яті комп'ютера у вигляді чисел з плаваючою крапкою згідно зі стандартом IEEE 754.

Нормалізованим експоненціальним виглядом числа (стандартним виглядом числа) є представлення його у формі:

$$a = \pm m * q^p,$$

де a – початкове число;

m – мантиса, 1 <= m <q;
q – основа СЧ;
p – порядок числа.
Знак числа у подальшому будемо позначати символом **s.**
Наприклад, **десяткові** числа 121,34; -100000 та 0, 0072 у стандартному вигляді в десятковій СЧ записуються відповідно так:
$1{,}2134*10^2$; $-1{,}0*10^5$; $7{,}2*10^{-3}$.
Зокрема, для числа $1{,}2134*10^2$ (в десятковому вигляді) маємо:
s = +; m = 1,2134; p = 2.
Формати дійсних чисел у двійковому вигляді:
КД - коротке дійсне (одинарна точність; 32 розряди);
ДД - довге дійсне (подвійна точність; 64 розряди);
ТД - тимчасове дійсне (розширений формат, дозволяє зберігати ненормалізовані числа; 80 розрядів).
Розподіл розрядів для наведених форматів:

формат	s	p	m
КД	1	8	23
ДД	1	11	52
ТД	1	15	64

Пояснення:
1) знак числа: s = 0, якщо число додатне; s = 1, якщо число від'ємне;
p та m тут зображуються двійковими числами у прямому коді;
2) наприклад, при записі у форматі КД один розряд виділяється під знак, 8 – під порядок і 23 – під мантису.
При записі чисел згідно зі стандартом IEEE 754 для уникнення від'ємного порядку використовується так званий **зміщений порядок**, суть якого ілюструє наступна таблиця (в якій наведено перетворення звичайного порядку на зміщений для формату КД):

істинний порядок	зміщений порядок у десятковому вигляді	зміщений порядок у двійковому вигляді
-127	0	0 0 0 0 0 0 0 0
-126	1	0 0 0 0 0 0 0 1
.....	.....	.....
.....	.....	.....
0	127	0 1 1 1 1 1 1 1
1	128	1 0 0 0 0 0 0 0
.....	.....	.....
.....	.....	.....
128	255	1 1 1 1 1 1 1 1

Для розрахунку зміщеного порядку можна користуватися наступною формулою:

$$\mathbf{ЗП = ІП + 2^{k-1} – 1}$$

де ЗП – зміщений порядок;
ІП – істинний порядок;

k – кількість розрядів, які виділені для запису порядку.
Наприклад (у десятковому вигляді), якщо ІП = -127, то ЗП = -127 + 2^{8-1} - 1 = 0;
якщо ІП = 128, то ЗП = 128 + 2^{8-1} - 1 = 255.
Приклад. Запишемо у пам'ять комп'ютера число -1435.75_{10} у форматі коротке дійсне.
Для цілої частини маємо (в результаті цілочисельного ділення та виписування остач): $1435_{10} = 10110011011_2$.
Для переведення дробової частини у двійкову СЧ її потрібно множини на основу СЧ і виписувати цілі частини доки дробова частина частки не стане дорівнювати нулю, або доки не буде досягнуто потрібну точність (кількість розрядів):
0.75 * 2 = 1.5 => ціла частина добутку = 1;
0.5 * 2 = 1.0 => ціла частина добутку = 1.
Дробова частина частки = 0 => множення припиняється.
Отже, $0.75_{10} = 0.11_2$.
Таким чином, $-1435.75_{10} = -10110011011.11_2$.
Перетворюємо отриманий результат до нормалізованого експоненціального вигляду (у двійковій СЧ):

$$-10110011011.11 = -1.011001101111*2^{1010}.$$

Виділена фоном одиниця називається **прихованою** і у представленні числа не записується, але враховується при обчисленнях.
Ми перенесли крапку на десять розрядів, тобто істинний порядок ІП = $10_{10}=1010_{10}$.
Обчислюємо зміщений порядок числа:

$$ЗП = 10 + 2^{8-1} - 1 = 137_{10} = 10001001_2.$$

Зміщений порядок числа можна обчислювати й одразу у двійковій формі:

```
  0 1 1 1 1 1 1 1   умовний нуль
+       1 0 1 0     істинний порядок
-----------------
  1 0 0 0 1 0 0 1   зміщений порядок
```

Отже, p = 10001001.
Оскільки число від'ємне, то s = 1.
Представлення мантиси у форматі КД містить 23 двійкові розряди, в отриманому числі 011001101111 розрядів тільки 12-ть, тому потрібно буде дописати справа (оскільки ми маємо справу із дробовою частиною) 11 нулів.
Таким чином, для числа -1435.75_{10} у форматі коротке дійсне ми отримаємо:

s	p	m
1	1 0 0 0 1 0 0 1	0 1 1 0 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0

Приклад. Запишемо у пам'ять комп'ютера число 625.01_{10} у форматі КД.
Для цілої частини маємо (в результаті цілочисельного ділення та виписування остач): $625_{10} = 1001110001_2$.
Переводимо дробову частину у двійкову СЧ:
0.01*2 = 0.02 => ціла частина добутку = 0;
0.02*2 = 0.04 => ціла частина добутку = 0;
0.04*2 = 0.08 => ціла частина добутку = 0;
0.08*2 = 0.16 => ціла частина добутку = 0;
0.16*2 = 0.32 => ціла частина добутку = 0;

0.32*2 = 0.64 => ціла частина добутку = 0;
0.64*2 = 1.28 => ціла частина добутку = 1;
0.28*2 = 0.56 => ціла частина добутку = 0;
0.56*2 = 1.12 => ціла частина добутку = 1;
0. 12*2 = 0.24 => ціла частина добутку = 0;
0. 24*2 = 0.48 => ціла частина добутку = 0;
0. 48*2 = 0.96 => ціла частина добутку = 0;
0. 96*2 = 1.92 => ціла частина добутку = 1;
0. 92*2 = 1.84 => ціла частина добутку = 1;
(14-ть дій множення на 2; – див. пояснення нижче).

Пояснення. У цьому випадку ми отримали, що точно представити зазначене число у форматі КД за стандартом IEEE 754 нам не вдасться. Тому ми представимо його з потрібною точністю, тобто з точністю 23 двійкові знаки. **Оскільки в цілій частині числа 10 знаків, то враховуючи приховану одиницю, маємо 9-ть знаків мантиси, а всього їх 23, тому ми і виконували 23 - 9 = 14 дій множення на 2 при обчисленні дробової частини.**

Враховуючи вищесказане, із зазначеною точністю дробова частина у двійковому вигляді дорівнює:
00000010100011.
Отже, $0.\ 01_{10} \approx 0.\ 00000010100011_2$.
Таким чином, $625.01_{10} = 1001110001.\ 00000010100011_2$.
Перетворюємо отриманий результат до нормалізованого експоненціального вигляду (у двійковій СЧ):

$$1001110001.\ 00000010100011 = 1.00111000100000010100011$$

Істинний порядок ІП = $9_{10}=1001_{10}$.
Обчислюємо зміщений порядок:

$$ЗП = 9 + 2^{8-1} - 1 = 136_{10} = 10001000_2.$$

Отже, p = 10001000.
Оскільки число додатне, то s = 0.
Таким чином, для число 625.01_{10} у форматі коротке дійсне має такий вигляд:

s	p	m
0	1 0 0 0 1 0 0 0	0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 1 1

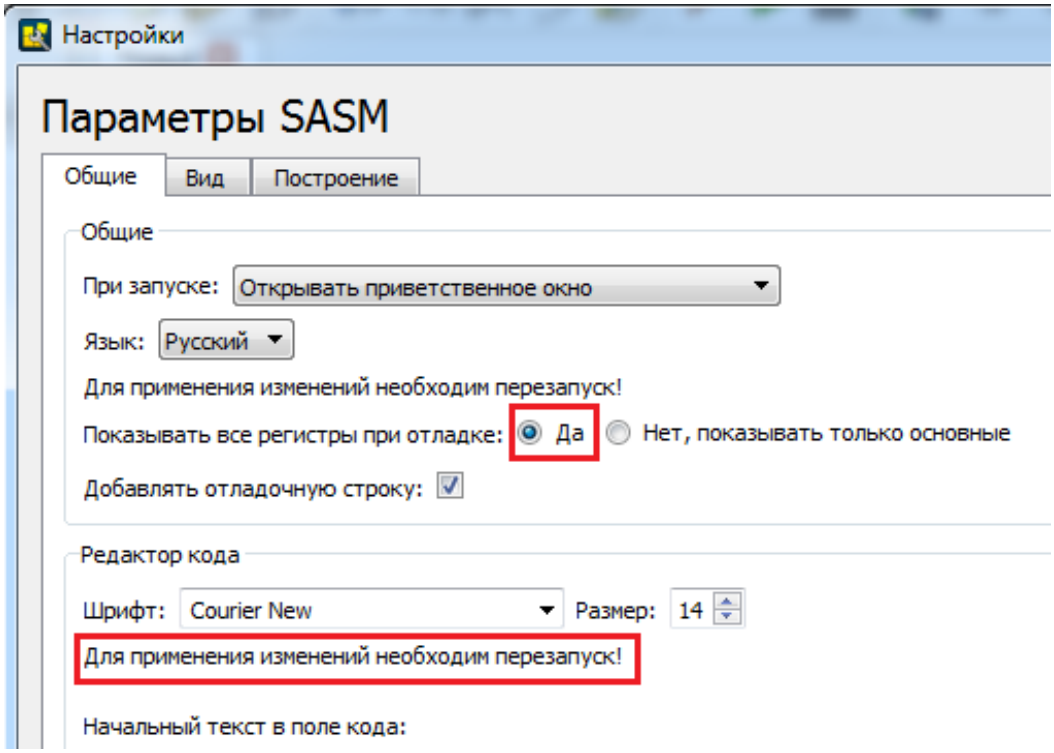
Ураховуючи вищенаведене, можна записати **загальний алгоритм представлення дійсних чисел у пам'яті комп'ютера:**
1) перевести число до двійкової СЧ;
2) записати отримане двійкове число в нормалізованому експоненціальному вигляді;
3) розрахувати зміщений порядок числа;
4) помістити знак, порядок і мантису до відповідних розрядів.

Примітка. Цікавою прикладною ілюстрацією наведеного матеріалу є те, що коли ви використовуєте комп'ютер для обчислень, то наприклад:
0,2 + 0,2 ≠ 0,4;
0,1 + 0, 4 ≠ 0,5

і т.д. для всіх дійсних чисел, які не точно представляються за стандартом IEEE 754, тобто для тих дійсних чисел, які не є степенями двійки, або сумою її степенів, як, наприклад, $0.75 = 0.5 + 0.25 = 2^{-1} + 2^{-2}$.

3.10 Арифметичний співпроцесор

Співпроцесор (FPU, Floating Point Unit) – пристрій для роботи із числами з плаваючою крапкою.
У сучасних комп'ютерах, як правило вбудований всередину основного процесора. Співпроцесор може працювати із 16-80 розрядними числами. Операції з цілими числами співпроцесор виконує значно повільніше, ніж основний процесор.
Регістри співпроцесора мають назви st0, st1, ... , st7.
Для **ініціалізації співпроцесора** призначена команда finit, яка втім не є обов'язковою. Ця команда очищує регістри st0 та st1.
Основним регістром є st0 – тільки до нього можна завантажувати числа зі змінних і до змінних результат. Для **завантаження змінних до співпроцесора** призначені команди: fld (для чисел із плаваючою крапкою), fild (для цілих чисел), fbld (для десяткових чисел).
Для того, щоб побачити роботу співпроцесора при відлагоджуванні у середовищі SASM, потрібне наступне налаштування (для його застосування необхідно перезавантажити SASM):



Для десяткових чисел слід використовувати префікс tword (від слова «ten»).
При завантаженні чисел відбувається наступне переміщення вмісту регістрів співпроцесора:
st(k) => st(k+1).
Проілюструємо це в середовищі SASM:

```
1   %include "io.inc"
2   section .data
3   f1 dd 0.1
4   f2 dd 0.01
5   f3 dd -135.75
6   section .text
7   global CMAIN
8   CMAIN:
9       mov ebp, esp; for correct debugging
10      fld dword[f1]
11      fld dword[f2]
12      fld dword[f3]
13      ret
```

Ввод

Вывод

Регистр	Hex
gs	0x2b
st0	0
st1	0
st2	0
st3	0
st4	0
st5	0
st6	0
st7	0

```
1   %include "io.inc"
2   section .data
3   f1 dd 0.1
4   f2 dd 0.01
5   f3 dd -135.75
6   section .text
7   global CMAIN
8   CMAIN:
9       mov ebp, esp; for correct debugging
10      fld dword[f1]
11      fld dword[f2]
12      fld dword[f3]
13      ret
```

Ввод

Вывод

Регистр	Hex
gs	0x2b
st0	0.10000000149011612
st1	0
st2	0
st3	0
st4	0
st5	0
st6	0
st7	0

```
1   %include "io.inc"
2   section .data
3   f1 dd 0.1
4   f2 dd 0.01
5   f3 dd -135.75
6   section .text
7   global CMAIN
8   CMAIN:
9       mov ebp, esp; for correct debugging
10      fld dword[f1]
11      fld dword[f2]
12      fld dword[f3]
13      ret
```

Ввод

Вывод

Регистр	Hex
gs	0x2b
st0	0.0099999997764825821
st1	0.10000000149011612
st2	0
st3	0
st4	0
st5	0
st6	0
st7	0

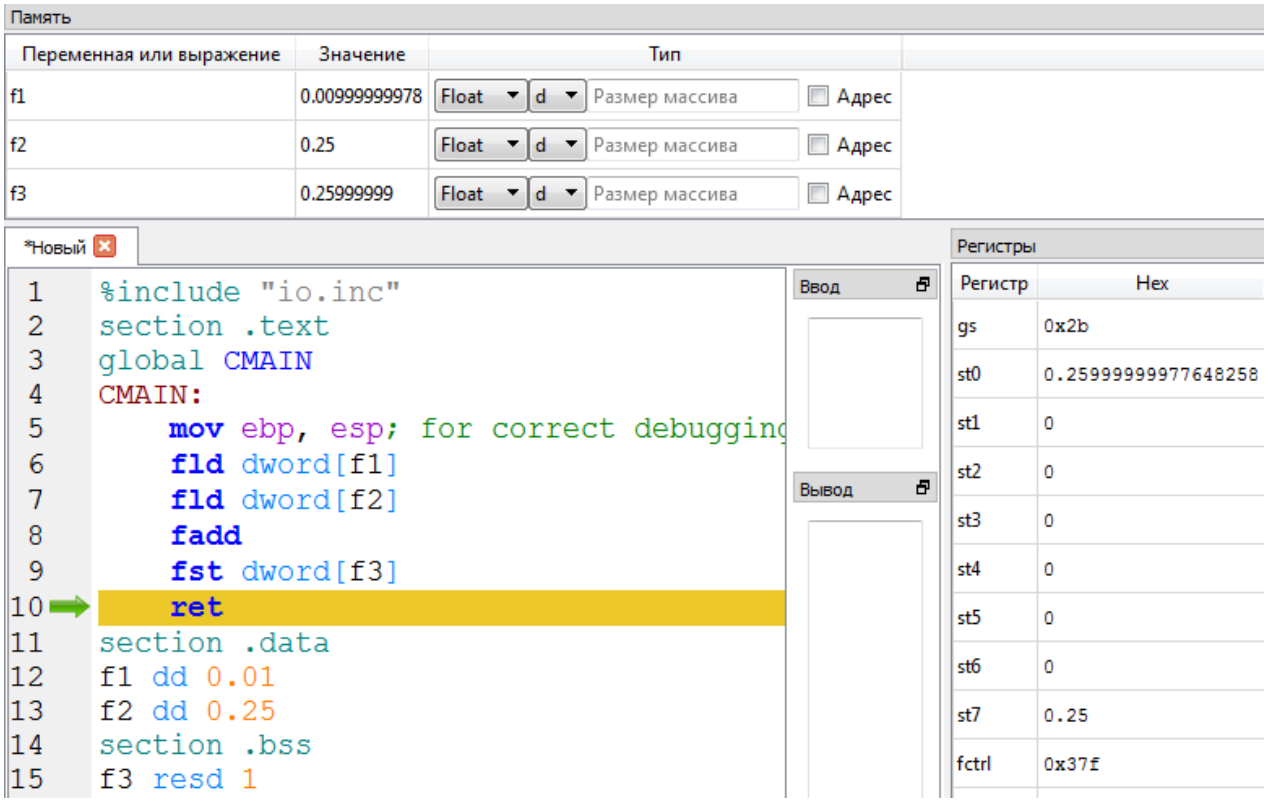
```
1   %include "io.inc"
2   section .data
3   f1 dd 0.1
4   f2 dd 0.01
5   f3 dd -135.75
6   section .text
7   global CMAIN
8   CMAIN:
9       mov ebp, esp; for correct debugging
10      fld dword[f1]
11      fld dword[f2]
12      fld dword[f3]
13      ret
```

Ввод

Вывод

Регистр	Hex
gs	0x2b
st0	-135.75
st1	0.0099999997764825821
st2	0.10000000149011612
st3	0
st4	0
st5	0
st6	0
st7	0

Додавання. Команда fadd додає числа, які знаходяться в регістрах st0 та st1. Існують різні варіанти її запису:
fadd ; st0 := st0 + st1
fadd st1 ; це ж саме
fadd st0,st1 ; це ж саме
fadd st1,st0 ; st1 := st1 + st0
fadd to st1 ; це ж саме.
Для **завантаження результату** виконання операцій **до змінних** у пам'яті, використовуються команди: fst (для чисел з плаваючою крапкою), fist (для цілих чисел) та fbstp (для десяткових чисел).
Команда fxch міняє місцями значення регістрів st0 та st1.
Приклад:



Команди порівняння чисел у співпроцесорі порівнюють значення регістрів співпроцесора і встановлюють біти C0, C2, C3 регістра swr. На практиці цікавими є команди FCOMI (FUCOMI) та FCOMIP (FUCOMIP), які можуть встановлювати біти регістра EFLAGS, що дозволяє робити умовні переходи при роботі зі співпроцесором.

Умова	ZF	PF	CF	Перехід
st0>st(i)	**0**	**0**	**0**	**ja**
st0<st(i)	**0**	**0**	**1**	**jb**
st0=st(i)	**1**	**0**	**0**	**je**

st0>=st(i)	*****	**0**	**0**	**jae**
st0=<st(i)	*****	**0**	*****	**jbe**
Недопустима операція (#IA)	**1**	**1**	**1**	

* – означає, що значення відповідного прапорця не відіграє ролі для умовного переходу.

Якщо аргумент у команді порівняння не вказаний, то порівнюються регістри st0 та st1.

Список деяких інших команд співпроцесора:

FABS Отримання модуля числа
FADD Додавання дійсних чисел
FADDP Додавання дійсних чисел із виштовхуванням зі стека
FCHS Зміна знаку числа
FCOM Порівняння дійсних чисел
FCOMP Порівняння дійсних чисел із виштовхуванням зі стека
FCOMPP Порівняння дійсних чисел із подвійним виштовхуванням зі стека
FCOS Обчислення косинуса
FDIV Ділення дійсних чисел a/b (a = st1, b = st0)
FDIVP Ділення дійсних чисел з виштовхуванням зі стека
FFREE Звільнення регістра st(i)
FIADD Додавання цілих чисел
FICOM Порівняння цілих чисел
FICOMP Порівняння цілих чисел з виштовхуванням зі стека
FIDIV Ділення цілих чисел
FILD Завантаження цілого числа
FIMUL Множення цілих чисел
FINCSTP Інкремент умісту вказівника стека
FINIT Ініціалізація FPU
FISUB Віднімання цілих чисел
FLD Завантаження дійсного числа
FLDL2E Завантаження $\log_2 e$
FLDL2T Завантаження $\log_2 10$
FLDLG2 Завантаження lg2
FLDLN2 Завантаження ln2
FLDPI Завантаження числа PI = 3.14...
FMUL Множення дійсних чисел
FMULP Множення дійсних чисел з виштовхуванням зі стека
FNOP Відсутність операції
FRNDINT Округлення до цілого значення
FSIN Обчислення синуса
FSINCOS Обчислення синуса та косинуса
FSQRT Обчислення квадратного кореня

FST Запис до пам’яті дійсного числа
FSTP Запис до пам’яті дійсного числа с виштовхуванням зі стека
FSUB Віднімання дійсних чисел
FSUBP Віднімання дійсних чисел із виштовхуванням зі стека
FTST Порівняння з нулем
FXTRACT Виділення мантиси та порядку
F2XM1 Обчислення функції $(2^x -1)$, де x = st0
FYL2X Обчислення функції $y*\log_2 x$, де x = st0, y = st1
FWAIT Очікування готовності FPU

Спосіб піднесення дійсних чисел до дійсного степеня за допомогою співпроцесора. Враховуючи властивості показникової функції, справедливою є рівність

$$A^b = 2^x \text{ (де } A > 0, A \neq 1)$$

Логарифмуючи обидві частини рівності за основою 2, дістанемо:

$$x = b*\log_2 A.$$

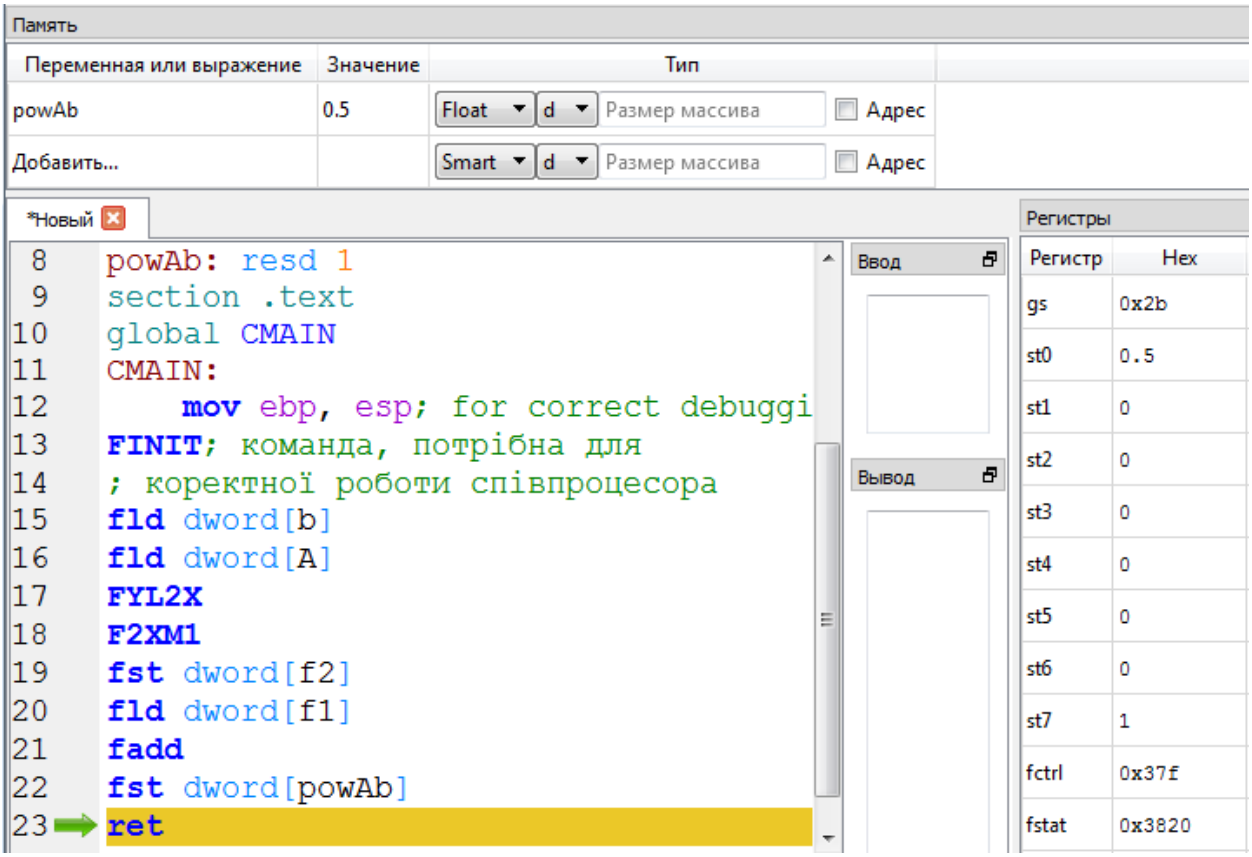
Таким чином, $A^b = 2^{b*\log_2 A}$.
Отже, використовуючи команди:

F2XM1 Обчислення функції $(2^x -1)$, де x = st0
FYL2X Обчислення функції $y*\log_2 x$, де x = st0, y = st1

можна підносити додатні дійсні числа, не рівні одиниці, до довільного дійсного степеня.
Приклад. Піднесемо число 0.25 до степеня 0.5.
Програма:

```
%include "io.inc"
section .data
A dd 0.25
b dd 0.5
f1 dd 1.0
section .bss
f2: resd 1
powAb: resd 1
section .text
global CMAIN
CMAIN:
FINIT; команда, потрібна для
; коректної роботи співпроцесора
fld dword[b]
fld dword[A]
FYL2X
F2XM1
fst dword[f2]
fld dword[f1]
fadd
```

fst dword[powAb]
ret



БІБЛІОГРАФІЧНИЙ СПИСОК

1. Арифметические основы вычислительной техники **[Електронний ресурс] : (курс лекций для студентов компьютерных специальностей)** / А. Алешин. – Режим доступу: http://vestikinc.narod.ru/AB. – Назва з екрана.
2. Інформатика та комп'ютерна техніка. Частина І. Електронно-обчислювальні машини: навч. посібник. / Н.П. Кухарська, Т.Є. Рак, Р.О. Григорчук, О.Б. Зачко, О. О. Смотр. – Львів: ЛДУ БЖД, 2011. – 120 с.
3. IEEE 754 - стандарт двоичной арифметики с плавающей точкой. **[Електронний ресурс] : (стаття)** / В. Яшкардин – Режим доступу: http://www.softelectro.ru/ieee754.html. – Назва з екрана.
4. Устройство персонального компьютера. **[Електронний ресурс] : (информационный сайт о высоких технологиях).** – Режим доступу: http://www.all-ht.ru/about.html.
5. Марек Р. Ассемблер на примерах. Базовый курс. / Р. Марек. – СПб : Наука и техника, 2005. – 240 с.
6. Расширенный ассемблер: NASM. **[Електронний ресурс] : (документация по ассемблеру NASM).** – Режим доступу: http://biguniverse.narod.ru/nasm.pdf.
7. Электронный учебник - Assembler. **[Електронний ресурс] : (учебник по программированию)** – Режим доступу: http://www.cyberforum.ru/assembler-articles/thread1005284.html. – Назва з екрана.
8. Калашников О.А. Ассемблер – это просто. Учимся программировать / О.А. Калашников. – СПб. : БХВ, 2011. – 336 с.
9. Система команд сопроцессора. **[Електронний ресурс] : (электронный учебник)** – Режим доступу: https://prog-cpp.ru/asm-coprocessor-command. – Назва з екрана.
10. Программирование на языке ассемблера. **[Електронний ресурс] : (электронный курс)** – Режим доступу: http://natalia.appmat.ru/c&c++/assembler.html. – Назва з екрана.