Міністерство освіти і науки України
ДВНЗ “Прикарпатський національний університет імені Василя Стефаника”
Кафедра комп’ютерної інженерії та електроніки

Методичний посібник
до виконання лабораторних робіт з дисципліни
“СИСТЕМНЕ ПРОГРАМУВАННЯ”

Затверджено
на засіданні кафедри радіофізики і електроніки
протокол № 4 від 8 листопада 2017 р

Розроблено:
доц. Голота В.І.

Івано-Франківськ 2017

Зміст

1. Програмна модель процесора, типи даних, директиви і адресація пам’яті 3
2. Синтаксис і формати команд асемблерів NASM і GAS.....19
3. Команди mov, lea, цілочисельні арифметичні.....42
4. Безумовні та умовні переходи, цикли, логічні побітові команди, бітові операції, зсуви ..54
5. Ланцюжкові команди.....64
6. Директиви препроцесора.....72
7. Виклики підпрограм 84
8. Взаємодія з ОС. Переривання.....99
9. Структури даних 118
10. Модульне програмування 128
Список використаних джерел.....134

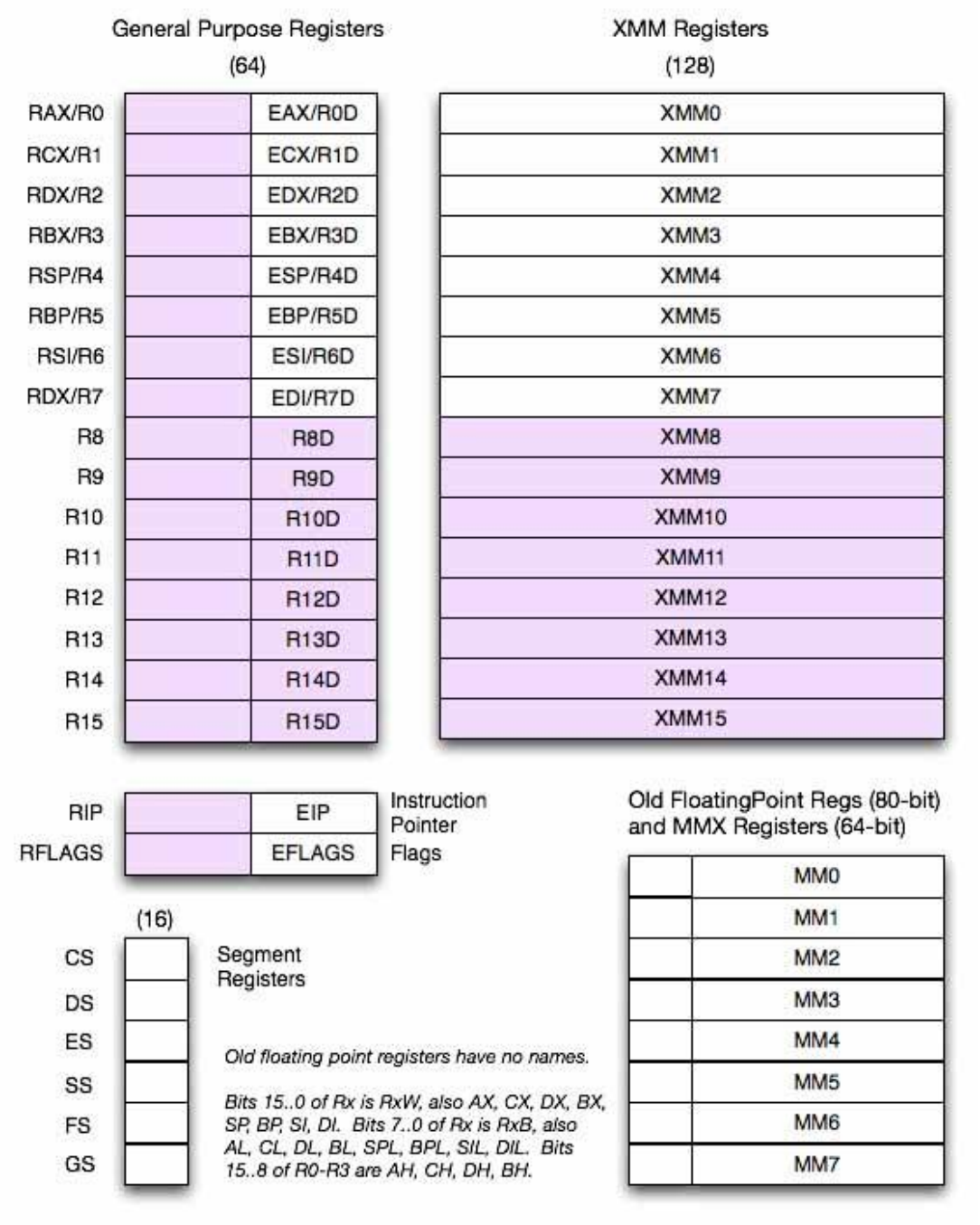
Лабораторна робота № 1.

1. Програмна модель процесора, типи даних, директиви і адресація пам'яті

Мета роботи: вивчення програмної моделі мікропроцесорів IA-32 і x86-64, типів даних і директив асемблера.

1. Програмна модель МП IA-32 і x86-64

Мікропроцесори фірми Intel IA32 (8086, 80186, 80286 – 16 біт і 80386, 80486 – 32 біт) і x86_64 мають наступні набори регістрів:



Примітка. Розовим кольором позначені регістри які є тільки в МП Intel x86_64.

2. Регістри загального призначення

Регістри загального призначення використовуються для зберігання:
- операндів арифметичних і логічних виразів;
- компонентів адрес;
- вказівників на комірки пам'яті.

МП ІА-32 має наступні цілочисельні 32-розрядні регістри загального призначення:

Регістри даних **eax** (Accumulator register) – акумулятор, використовується для зберігання даних проміжкових обчислень;	**Регістри вказівники для роботи зі стеком** **esp** (Stack pointer register) – регістр вказівник стеку. Містить адрес верхівки стеку.
ebx (Base register) – базовий регістр, використовується для зберігання базової адреси деякого об'єкта в пам'яті;	**ebp** (Base pointer register) – регістр вказівник бази кадру стека. Призначений для організації довільного доступу до даних всередині стеку.
ecx (Counter register) – регістр лічильник, неявно використовується в деяких командах для організації циклів;	**Індексні регістри для підтримки ланцюжкових операцій** **esi** (Source index register) – індекс джерела, в ланцюгових командах містить поточну адресу елемента джерела;
edx (Data register) – регістр даних, використовується для зберігання результатів проміжкових обчислень і введення-виведення;	**edi (**Destination index register) – індекс приймача, в ланцюгових командах містить поточну адресу елемента приймача.

До цих регістрів можна звертатися по "частинах". Наприклад до молодших 16-біт регістра `eax` можна звернутися як до `ax.ax`. Вони в свою чергу містять дві однобайтні половинки, до яких можна звернутися як до `ah` (старшого), `al` (молодшого) байту.

Інструкції, в яких використовуються регістри з префіксом "`r`" дають доступ до байтових регістрів. Для цього старші частини байтів `ah, dh, ch, bh` замінюються молодшими частинами наступних регістрів `sil, dil, spl, bpl`.

В МП x86-64 регістри 64-розрядні і відповідно позначаються з префіксом "r": `rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi`. Додатково добавлено вісім нових 64-розрядних регістрів `r8, r9,…, r15` в яких префікс "`r`" вказує, що в не 64-розрядних інструкціях довжина інструкції може мати різні розміри:

`r8b` – байт (8-бітів)
`r8w` – слово (16-бітів)
`r8d` – подвійне слово (32- біти)
`r8` – почотирне слово (64-біти)

3. Сегментні регістри

Сегментні регістри містять адреси пам'яті, з яких починаються відповідні сегменти:
- **cs** (Code segment) – сегмент коду, містить адресу сегменту з машинними командами.
- **ds** (Data segment) – сегмент даних, містить адресу сегменту даних, які обробляє програма.
- **ss** (Stack segment) – сегмент стеку, містить адресу області пам'яті, яку використовує стек.

Додаткові сегменти даних. Якщо програмі недостатньо одного сегменту даних, то вона може використати ще три додаткових сегменти даних, адреси яких зберігаються в регістрах `es, fs, gs.`

• **es** (Extra segment) – додатковий сегмент, який використовується неявно в командах роботи з символьними рядками як сегмент отримувач.

• **fs** (F segment) – додатковий сегментний регістр без спеціального призначення.

• **gs** (G segment) – додатковий сегментний регістр без спеціального призначення.

4. Регістри спеціального призначення МП 80x86

Регістр вказівник команд **eip/ip** (Instruction Pointer register) - містить зміщення відносно вмісту сегментного регістра `cs` (`cs+offset` задають адресу) наступної команди, яка буде виконуватися.

Регістр прапорів **eflags/flags** містить інформацію про стан як самого МП, так і виконуваної програми. Найбільш важливі прапори:

• **cf** (*Carry flag*) – прапор перенесення:

1 – під час арифметичної операції було перенесення із старшого біту (7-го, 15-го, 31-го, 63-го для 8, 16, 32, 64 розрядних операндів) результату;

0 – перенесення не було.

• **pf** (*Parity flag*) – прапор паритету:

0 – 8-м молодших розрядів результату містить непарне число одиниць;

1 – 8-м молодших розрядів результату містить парне число одиниць.

• **af** (*Auxiliary carry flag*), фіксація факту перенесення (позики) в (з) молодшої тетради при роботі з BCD-числами.

• **zf** (*Zero flag*) – прапор нуля:

0 – результат останньої операції не нульовий.

1 – результат останньої операції нульовий;

• **sf** (*Sign flag*) – прапор знаку:

0 – старший біт результату дорівнює 0;

1 – старший біт результату дорівнює 1;

• **tf** (*Trap flag*) – дозволяє переведення інструкцій процесора в однокроковий (DEBUG) режим:

0 – звичайний режим;

1 – однокроковий режим.

• **if** (Interrup flag) – визначає чи зовнішні переривання ігноруються чи обробляються:

0 – ігноруються;

1 – обробляються.

• **df** (*Direction flag*) – прапор напрямок символьних рядків:

1 – напрямок “назад”, від старших адресів до молодших;

0 – напрямок “вперед”, від молодших адресів до старших.

• **of** (*Overflow flag*) – прапор втрати значущого біта при арифметичних операціях:

1 - під час арифметичної операції було перенесення (позика) в (з) старшого (знакового) біта результату;

0 – не було перенесення/позики із старшого (знакового) біта.

5. Службові регістри

5.1. Керуючі регістри

До керуючих відносяться вісім (32-бітові) регістри CR0, CR1, CR2, CR3, CR4, CR5, CR6, CR7.

Регістр CR0:

• 0-біт, дозвіл захисту. Переводить процесор у захищений режим;

• 1-біт, моніторинг співпроцесора (MP). Викликає виняток 7 для кожної команди wait;

• 2-біт, емуляція співпроцесора (EM). Викликає виняток 7 для кожної команди співпроцесора;

• 3-біт, перемикання задач (TS). Дозволяє визначити, чи відноситься даний контекст співпроцесора до поточної задачі чи ні. Викликає виняток 7 при виконанні наступної команди співпроцесора;

• 4-біт, індикатор підтримки інструкцій співпроцесора (ET);

• 5-біт, дозвіл стандартного механізму повідомлень про помилку співпроцесора (NE);

• 6-15 біти, не використовуються;

• 16-біт, дозвіл захисту від запису на рівні привілеїв супервізора (WB);

• 17-біт, не використовується;

• 18-біт, дозвіл контролю вирівнювання (AM);

• 19-28 біти, не використовуються;

• 29-біт, заборона наскрізного запису кешу і циклів анулювання (NW);

• 30-біт, заборона використання кешу (CD);

• 31-біт, включення механізму сторінкової переадресації.

Регістр CR1 зарезервований.

Регістр CR2 зберігає 32-бітну лінійну адресу, для якої була отримана остання відмова сторінки пам'яті.

Регістр CR3:

• 3-біт, кешування сторінок із наскрізним записом (PWT);

• 4-біт, заборона кешування сторінки (PCD);

• 11-31, 20 старших бітів фізичної адреси таблиць каталогу сторінок при умові, що 5-й біт регістра CR4 дорівнює 1.

Регістр CR4:

• 0-біт, дозвіл використання віртуального прапора переривань в режимі V8086 (VME);

• 1-біт, дозвіл використання віртуального прапора переривань в захищеному режимі (PVI);

• 2-біт, перетворення інструкції RDTSC в привілейовану (TSD);

• 3-біт, дозвіл точок зупинки при зверненні до портів введення-виведення (DE);

• 4-біт, включає режим адресації з 4 мегабітними сторінками (PSE);

• 5-біт, включає 36-бітний фізичний адресний простір (PAE);

• 6-біт, дозвіл винятку MC (MCE);

• 7-біт, дозвіл глобальної сторінки (PGE);

• 8-біт, дозвіл виконання команди RDPMC (PMC);

• 9-біт, дозволяє команди швидкого збереження/відновлення стану співпроцесора (FSR).

5.2. Системні адресні регістри

До системних адресних регістрів відносяться чотири (16-бітові) регістри таблиць GDTR, IDTR, LDTR, TR.

• GDTR – 6-байтовий регістр, в якому міститься лінійна адреса глобальної дескрипторної таблиці;

• IDTR – 6-байтовий регістр, який містить 32-бітову лінійну адресу таблиці дескрипторів обробників переривань;

• LDTR – 10-байтовий регістр, який містить 16-бітовий селектор (індекс) для GDT і 8-байтовий дескриптор;

• TR – 10-байтовий регістр, який містить 16-бітовий селектор (індекс) для GDT і 8-байтовий дескриптор з GDT, який описує TSS поточної задачі.

5.3. Регістри налагодження

• DR0-DR3 – зберігають 32-бітові лінійні адреси точок зупинки.
• DR6 (еквівалентно DR4) – відображає стан контрольних точок.
• DR7 (еквівалентно DR5) – керує встановленням контрольних точок.

6. Розміри даних

В програмах на асемблері використовуються наступні позначення для різних за розміром ділянок пам'яті:

`b` (byte) – 1 байт, 8-бітів
`w`(word) – 2 байти, 16-бітів
`d`, `l` (long, double word) – 4 байти, 32-біти
`q` (quad, quadword) – 8 байтів, 64-біти
`t` (ten bytes) – 10 байтів
Специфікатори типу даних: **`byte, word, dword, qword, tword.`**
В Linux для вирівнювання і адресації пам'яті використовуються:
`paragraph` (double quadword) – 16 байтів
`page` – 256 байт
`segment` – 65536 байт

7. Константні типи даних

Асемблер NASM підтримує наступні типи констант: числові, символьні, стрічкові, з плаваючою крапкою, стрічки UNICODE, запаковані BCD числа.

7.1. Числа

Числові константи є просто числа. Числа в 2-й, 8-1, 10-й і 16-й системах можна задати з використанням префіксів або суфіксів:

`0b11110001, 0b111_1000, 0y111_1000, 111110001b, 1111_10001b,` – двійкове ціле
`0o1234567, 01234567o, 01234567q` – вісімкове ціле
`200, 0200, 0200d, 0d200` – десяткове ціле
`0x1A2B3C4E5F, 1A2B3C4E5Fh, $A2B3C4E5F` – шістнадцяткове ціле (при використанні символу `$` потрібно слідкувати, щоб зразу після нього стояла цифра, а не буква, наприклад, `$0f5` замість `$f5`. Аналогічно необхідно слідкувати і за першим символом при використанні символу `h`, наприклад, `0a51h`, а не `a51h`). Приклади пересилання різних чисел у регістр ax:

```
mov ax,200   ; десяткове
mov ax,0200  ; десяткове
mov ax,0200d ; явне десяткове
mov ax,0d200 ; явне десяткове
mov ax,0c8h  ; шістнадцяткове
mov ax,$0c8  ; шістнадцяткове: перед буквою потрібний 0
mov ax,0xc8  ; шістнадцяткове
mov ax,0hc8  ; шістнадцяткове
mov ax,310q  ; вісімкове
mov ax,310o  ; вісімкове
mov ax,0o310 ; вісімкове
mov ax,0q310 ; вісімкове
mov ax,11001000b   ; двійкове
mov ax,1100_1000b  ; двійкове
mov ax,1100_1000y  ; двійкове
mov ax,0b1100_1000 ; двійкове
mov ax,0y1100_1000 ; двійкове
```

7.2. Символьні стрічки

Символьні стрічки містять до 4-х (i386) або до 8-ми (x86-64) символів взятих у одинарні (‘…’), подвійні (“…”) або лівонахилені апострофи (`…`).

У стрічки, які знаходяться у лівонахилених апострофах, можна поміщати esc-послідовності у С-стилі:

```
\’ одинарні лапки (’)
\" подвійні лапки (")
\‘ одинарні лівонахилені апострофи (‘)
\\ зворотній slash (\)
\? знак запитання (?)
\a BEL (ASCII 7)
\b BS  (ASCII 8)
\t TAB (ASCII 9)
\n LF  (ASCII 10)
\v VT  (ASCII 11)
\f FF  (ASCII 12)
\r CR  (ASCII 13)
\e ESC (ASCII 27)
\377 до 3-х вісімкових цифр – літеральний byte
\xFF до 2-х шістнадцяткових цифр – літеральний byte
\u1234  4-ри шістнадцяткові цифри – символи Unicode
\U12345678 8-м шістнадцяткових цифр – символи Unicode
```

Unicode символи задані як `\U`, `\u` конвертуються у UTF-8. Наприклад наступні рядки є еквівалентними (символ посмішки).

```
db ‘\u263a‘         ; UTF-8
db ‘\xe2\x98\xba‘   ; UTF-8
db 0E2h, 098h, 0BAh ; UTF-8
```

7.3. Символьні константи (character constant)

Символьна константа розглядається як стрічка довжиною до 8 байт. Символьна константа довжиною більшою як один байт розміщується у регістрі і пам’яті з порядком байтів “little-endian”:

```
mov eax,’abcd’  ; 0x61626364 => 0x64636261,
```

7.4. Стрічкові константи (string constant)

Стрічкові константи є стрічками символів, які використовуються у деяких директивах (db) і `incbin` (як імена файлів):

```
db ’hello’             ; стрічкова константа
db ’h’,’e’,’l’,’l’,’o’ ; еквівалентна стрічкова константа
dd ’ninechars’         ; стрічкова константа doubleword
dd ’nine’,’char’,’s’   ; стрічкова константа з трьох doublewords, але
db ’ninechars’,0,0,0   ; яка так реально розміщується в пам’яті
```

7.5. Константи з плаваючою крапкою

Константи з плаваючою крапкою сприймаються тільки як аргументи директив `DB`, `DW`, `DD`, `DQ`, `DT` і `DO`:

```
db -0.2 ; "1/4 точність"
dw -0.5 ; IEEE 754r/SSE5 1/2 точність
dd 1.2 ; 1-на точність
dd 1.222_222_222 ; '_' дозволено розбивати на групи
dd 0x1p+2 ; 1.0x2^2 = 4.0
```

```
dq 0x1p+32 ; 1.0x2^32 = 4 294 967 296.0
dq 1.e10 ; 10 000 000 000.0
dq 1.e+10 ; синонім до 1.e10
dq 1.e-10 ; 0.000 000 000 1
dt 3.141592653589793238462 ; pi
do 1.e+4000 ; IEEE 754r почотирна точність
```

7.6. Запаковані BCD константи

Запаковані BCD константи можуть використовуватися як 80-бітові числа з плаваючою крапкою. Вони вказуються префіксом `0p` або суфіксом `p`, і можуть містити до 18 десяткових цифр:

```
dt  12_345_678_901_245_678p
dt -12_345_678_901_245_678p
dt +0p33
dt 33p
```

8. Порядок розміщення байтів у пам'яті

У різних машинах використовується різна послідовність байтів для зберігання чисел. Термін **little-endian** використовується для позначення послідовності байтів в якій молодші розрядові байти розміщуються в молодших адресах пам'яті. Термін **big-endian** позначає послідовності байтів у яких молодші розрядові байти розміщуються в старших адресах пам'яті. В Intel процесорах використовується little-endian послідовність зберігання байтів (II), а в процесорах фірми Motorola – big-endian послідовність (MM). Так для послідовностей чисел

```
m1 db 0,0,0x43,0x21
```

порядок розміщення байтів у пам'яті може буде наступним:

	little-endian				**big-endian**			
Адреси пам'яті	0	1	2	3	0	1	2	3
Розрядові байти	21	43	00	00	43	21	00	00

```
m1 dw 0,0,0x43,0x21
```

	little-endian				**big-endian**			
Адреси пам'яті	0	1	2	3	0	1	2	3
Розрядові байти	00	21	00	43	43	00	21	00

9. Псевдоінструкції і директиви

Псевдоінструкції і директиви не виконуються процесором. Вони самі виконують якусь дію, яка не траслюється в машинний код або інформує процесор. Псевдоінструкції і директиви використовуються для:

- визначення констант;
- визначення і резервування пам'яті для зберігання даних;
- розбивки пам'яті на сегменти;
- включення початкових файлів за умовою;
- включення інших файлів.

9.1. Директиви

До директив відносяться BITS, DEFAULT, SECTION або SEGMENT, ABSOLUTE, EXTERN, GLOBAL, COMMON, CPU, FLOAT.

BITS xx, де xx=16, 32, 64 задає розрядність коду, який генерує асемблер.

DEFAULTS – змінює значення параметрів асемблера за замовчування.

ABSOLUTE – є альтернативною директивою до SECTION, створюючи сегмент, який починається з абсолютної адреси:

```
absolute 0x1A
kbuf_chr resw 1
kbuf_free resw 1
kbuf resw 16
```

EXTERN – імпортує символи з інших модулів:

```
extern _printf
extern _sscanf,_fscanf
```

GLOBAL – експортує символи в інші модулі:

```
global _main
_main:
; деякий код
```

COMMON – оголошує загальну область пам'яті для декількох модулів:

```
common intvar 4
```

CPU xx, де xx=8086, 186, 286, 386, 486, 586, 686, P2, P3, P4, X64, IA64 – обмежує асемблер інструкціями вказаного процесора.

FLOAT – задає оброблення констант з плаваючою крапкою.

SECTION або SEGMENT – задає тип сегментів у пам'яті.

Адреси сегментів пам'яті різного призначення зберігаються у сегментних регістрах.

Синтаксис NASM. Типи сегментів:

.text – сегмент коду

.data – сегмент ініціалізованих даних

.bss – сегмент неініціалізованих даних (в ньому розміщуються буфери)

.stack – сегмент стеку

Директиви задання сегмента

```
segment <.ім'я сегменту>
section <.ім'я сегменту>
```

Синтаксис GAS. Типи сегментів:

.text – сегмент коду

.data – сегмент ініціалізованих даних

.rodata – сегмент даних тільки для читання

.stack – сегмент стеку

.bss - сегмент неініціалізованих даних (blank static storage)

Синтаксис TASM, MASM

segment .code – сегмент коду

.data – сегмент ініціалізованих даних. Також використовується для даних типу near.

.const – сегмент констант

.data? - сегмент неініціалізованих даних. Також використовується для даних типу near.

.stack [size] – сегмент стеку розміром size

.fardata – початок або продовження сегменту ініціалізованих даних типу far.

.fardata? – початок або продовження сегменту неініціалізованих даних типу far.

Директиви задання сегмента

.section <.ім'я сегменту> – директива задання сегмента

9.2. Псевдоінструкції оголошення, ініціалізації і резервування пам'яті

До псевдоінструкцій відносяться:

- `DB, DW, DD, DQ, DT` для оголошення і ініціалізації пам'яті в секції `.data`;
- `RESB, RESW, RESD, RESQ, REST` для резервування пам'яті в секції `.bss`;
- `INCBIN` створення області пам'яті заповненої даними із зовнішнього (графічного або акустичного) файлу;
- `EQU` визначає символ для заданого константного значення;
- префікс `TIMES` повторює інструкцію, яка асемблюється, задане число разів.

Синтаксис NASM.

Псевдоінструкції оголошення ініціалізованих даних не просто резервують пам'ять, а вказують, які значення в цій пам'яті повинні бути до моменту запуску програми. Псевдоінструкції **`db, dw, dd, dq, dt`** виділяють і ініціалізують області пам'яті розміром байт, слово, подвійне слово, почотирне слове, десять байтів. Прийнято позначати виділені області пам'яті позначками, які будуть асоційовані з адресою першого байту.

```
section    .data
L1 db 0       ; byte позначений L1 і ініціалізований значенням 0
L2 db 0, 1, 2, 3 ; визначено 4 байти із значеннями 0, 1, 2, 3
L3 db 110101b ; byte ініціалізований бінарним значенням 110101 (53₁₀)
L4 db 12h     ; byte ініціалізований шістнадцятковим значенням 12 (18₁₀)
L5 db 17o     ; byte ініціалізований вісімковим значенням 17 (15₁₀)
L6 dw 0x1234  ; word 0x34 0x12
L7 dw 'a'     ; word 0x61 0x00
L8 dw 'ab'    ; word 0x61 0x62 (character константа)
L9  dd 0x12345678  ; 0x78 0x56 0x34 0x12
L10 dd 1.234567e20 ; floating-point константа
L12 dd 1A92h   ; double word ініціалізований шістнадцятковим значенням 1A92
L13 dq 0x123456789abcdef0 ; 8-ми байтна константа
L14 dq 1.234567e20 ; double-точності float
L15 dt 1.234567e20 ; extended-точності float
```

Для задання символу, його потрібно взяти в одинарні або подвійні лапки. Аналогічно можна задати символьний рядок. Всередині подвійних лапок, одинарні лапки розглядуються як звичайний символ. Те саме можна сказати і про подвійні лапки всередині одинарних.

```
section    .data
L1 db 'A' ; byte ініціалізований ASCII кодом для A (65)
L2 db "A" ; byte ініціалізований ASCII кодом для A (65)
L3 db 'Hello word'
L4 db "Hello word"
L5 db 'So I say: "Don', "'", 't panic"'
L6 db "w", "o", "r", 'd', 0 ; визначено символьну стрічку C = "word"
L7 db 'word', 0 ; так само як L6

hexstr  db "01 02 03 04 05 06 0A",10
hexlen  equ $-hexstr       ; $ - поточна адреса
digits  db "0123456789"
ClearTern db 27  ;  <ESC>
```

Псевдоінструкція `dd` може визначати як цілі числа, так і константи звичайної точності з плаваючою крапкою. Псевдоінструкція `dq` визначає тільки константи подвійної точності з плаваючою крапкою.

Псевдоінструкція `equ` визначає символ для заданого константного значення, яке не може бути змінене пізніше:

```
message db 'hello, world'
msglen equ $-message  ; msglen є константою із значенням 12
```

Префікс `times` повторює інструкцію при асемблюванні задане число разів:

```
zerobuf times 64 db 0
L12 times 100 db 0 ; виділення 100 байтів ініціалізованих нулями
```

Причому `times` не є константою, а виразом:

```
buffer db 'hello, world'
times 64-$+buffer db ' '   ; виділення буфера довжиною 64 байти
```

Псевдоінструкції резервування пам'яті повідомляють асемблеру про потреби в оперативній пам'яті. Вони поділяються на два види: псевдоінструкції резервування неініціалізованої пам'яті і псевдоінструкції задання початкових даних.

Псевдоінструкції резервування неініціалізованої пам'яті повідомляють асемблеру, що потрібно зарезервувати задану кількість комірок пам'яті. Псевдоінструкції **`resb, resw, resd, resq, rest`** резервують неініціалізовані комірки пам'яті розміром байт, слово, подвійне слово, почотирне слово, десять байт в секції `.bss`. Після псевдоінструкцій вказується число, яке задає кількість комірок пам'яті. Перед псевдоінструкцією ставиться позначка.

```
section   .bss
bufflen    equ 16    ; задання розміру буфера
buff resb, bufflen  ; за адресою buff буде розміщено буфера розміром 16 байт
L13       resw 100  ; за адресою L13 буде виділено 100 слів
x         resd      ; за адресою x буде розміщене подвійне слово
buffer    resb 64   ; резервувати 64 bytes
wordvar   resw 1    ; резервувати word
realarray resq 10   ; масив 10 reals
ymmval    resy 1    ; один YMM регістр
```

Синтаксис GAS.

Псевдоінструкції виділення і ініціалізації даних в пам'яті:

```
.section     .data
.byte 0x10, 0112, 0x102  - розміщує кожний вираз як 1 байт
.short -123, 456 – розміщує кожний вираз як 2 байти
.long – 0xaabbaabb - розміщує кожний вираз як 4 байти
.quard – розміщує кожний вираз як 8 байтів
.octa 0x123456787898  - розміщує велике число
.float 0f-3141.5926E-10  - розміщує число з плаваючою крапкою
.ascii "str" – розміщує символьний рядок str, не добавляючи нульових байтів
.string "str" – розміщує символьний рядок str, добавляючи нульовий байт
.asciiz "str" – розміщує символьний рядок str, добавляючи нульовий байт
```

. – поточна адреса, наприклад:

```
label:  .long   ; визначення позначки label, яка містить свою власну адресу
```

```
.section .data
L1:  .byte  0        # байт ініціалізований нулем
L2:  .short  1000    # слово ініціалізоване 1000
L3:  .byte  b110101  # байт ініціалізований бітами
L4:  .byte  0x12     # байт ініціалізований шістнадцятковим число
L5:  .byte  o17      # байт ініціалізований вісімковим числом
L6:  .long 0x1A92    # подвійне слово ініціалізоване 0x1a92
L7:  .byte  'A'      # байт ініціалізований кодом "A"
L8:  .byte  0,1,2,3  # визначення 4-х байтів
L9:  .byte 'w','o','r','d',0  # визначення С рядка "word"
```

Якщо дані не передбачається змінювати, їх поміщають в спеціальну секцію (read only)

```
.section    .rodata
.string "Program version 1.1"
```

Якщо дані потрібно вирівнювати на границю адресів кратну степені двійки, то використовують псевдоінструкцію

```
.p2align  степінь двійки, заповнювач
.data
.string "Hello world"
```

```
.p2align 2      # вирівнюється на границю 4 байти для наступного .long
.long 123456
```

Для створення нових символів використовується псевдоінструкцію `.set`

```
.set  символ, вираз
.set abc, 45
hello_str:
.string ”Hello world\n”
.set  hello_str_length,  hello_str -1     # довжина рядка
```

Попередні псевдоінструкції потребують ініціалізуючого значення для розміщуваних змінних. Для неініціалізованих даних, наприклад буферів, використовується псевдоінструкція `.space` у секції `bss`.

```
.space кількість байтів
.space кількість байтів, заповнювач
.bss
buffer:
            .space  1024
.structura:
            .space 32
```

9.3. Інші псевдо інструкції і директиви препроцесора NASM

Псевдоінструкція `INCBIN` включає бінарний (графічний або звуковий) файл у вихідний файл:

```
incbin "file.dat"       ; включити увесь файл
incbin "file.dat",1024 ; пропустити перші 1024 байти
incbin "file.dat",1024,512 ;пропустити перші 1024 і включити наступні 512
байт
```

10. Використання позначок в програмі NASM

Позначки використовуються в секції коду для галуження програми. Позначка `label:` є вказівником і задає адресу в пам’яті. Позначка `.label:` є локальною. Якщо позначка взята в квадратні дужки `[label]`, вона задає значення за адресою `label`.

Деякі асемблери розрізняють позначки з двокрапками і без. NASM такі позначки не розрізняє. Звичайно програмісти ставлять двокрапку після позначок, якими позначені машинні команди, на які можна передати керування, але не ставлять двокрапку після позначок, які позначають дані у пам’яті (змінні).

Приклад використання позначок в 32-бітному режимі:

```
mov al, [L1]  ; копіювати в регістр al дані (байт) за адресою L1
mov eax, L1   ; копіювати в регістр EAX адресу байта з позначкою L1
mov [L1], ah  ; копіювати регістр AH у комірку з адресою L1
mov eax, [L6] ; копіювати подвійне слово за адресою L6 у регістр EAX
add eax, [L6] ; EAX = EAX + подвійне слово за адресою L6
add [L6], eax ; до подвійного слова за адресою L6 додати регістр EAX
mov al, [L6];копіювати перший байт подвійного слова за адресою L6 у регістр
AL
```

При записуванні безпосереднього даного (числа) у пам’ять без вказання його розміру асемблер видає помилку:

```
mov [L6], 5 ; записати число 5 у пам’ять за адресою L6 – помилка. Не вказано
як записати число 5 – як byte, word, чи double word
```

Для вказування розміру записуваного безпосереднього числа потрібно вказати один із специфікаторів `byte, word, dword, qword, tword`.

```
mov dword [L6], 5 ; записати 5, як подвійне слово, за адресою L6
mov [L6], dword 5 ; або так
```

Спеціальна псевдопозначка **$** – містить поточну адресу.

11. Способи адресації пам'яті

Простір пам'яті призначений для зберігання кодів команд і даних, для доступу до яких використовуються різні способи адресації. Операнди можуть знаходитися у внутрішніх регістрах процесора (найбільш зручний і швидкий варіант) або в системній пам'яті (самий поширений варіант). Дані можуть також знаходиться у пристроях введення/виведення. Місцезнаходження операндів задається кодом команди. Для кожної команди методи адресації визначають звідки взяти вхідний і куди помістити вихідний операнд.

11.1 Безпосередня адресація

Команда містить безпосередньо сам операнд (безпосередній операнд):

```
value db 5      ; виділення і ініціалізація байту
mov ax, value+2 ; скопіювати адресу value+2 у регістр
mov eax, 45h    ; скопіювати безпосереднє значення 45h у регістр ax
```

Безпосередня адресація дозволяє підвищити швидкість виконання операції, так як в цьому випадку вся команда, включно з операндом, зчитується з пам'яті одночасно і на час виконання команди зберігається в процесорі в спеціальному регістрі команд (РК). При використанні безпосередньої адресації появляється залежність кодів команд від даних, що потребує зміни програми при кожній зміні безпосереднього операнда.

11.2. Регістрова адресація

При *регістровій адресації* операнд знаходиться у регістрі (регістровий операнд):

```
mov  eax, ecx    ; скопіювати значення з регістра ecx в регістр eax
mov dx, tax_rate ; скопіювати адресу комірки пам'яті у регістр
mov count, cx    ; скопіювати значення з регістра у пам'ять
```

11.3. Пряма і непряма адресація

При *прямій адресації* операнд містить адресу (зміщення) комірки пам'яті. Таке зміщення називається *ефективною* адресою, так як вона відраховується від початку сегменту, адреса якого знаходиться у регістрі `ds`. Так як для обчислення ефективної адреси необхідна і адреса сегменту, тому така адресація є повільною.

```
section .data
addr1 dd 0
section .text
mov eax,addr1   ; скопіювати в регістр eax адресу addr1
add eax,2       ; збільшити адресу на 2
```

При *прямій адресації із зміщенням* використовуються арифметичні операції для модифікації адреси. Приклад копіювання даних з таблиці у регістри:

```
byte_table db 14, 15, 22, 45       ; таблиця байтів
word_table dw 134, 345, 564, 123   ; таблиця слів
mov cl, byte_table+2   ; скопіювати адресу 3-го елементу з byte_table
mov cx, word_table+3   ; скопіювати адресу 4-го елементу з word_table
```

Непряма адресація пам'яті використовує базові (EBX, BX, EBP, BP) і індексні регістри (DI, SI) для адресації комірок пам'яті. Для доступу до значень комірок регістри записуються в квадратних дужка, наприклад **[EBX].**

Операнд команди може містити адресу комірки пам’яті або регістр з адресою комірки пам’яті. Адреса комірки пам’яті задається змінною (позначкою). Для доступу до значень комірки пам’яті змінну записують у квадратних дужках, наприклад **`,[num].`**

```
segment .data
num dw 1,2,3,4,5
segment .text
global _start
_start:
mov eax,[num]     ; завантажити в регістр eax значення за адресою num
mov ebx,[num+2]   ; nasm - завантажити в регістр ebx значення,
                  ; яке знаходиться за адресою num+2
mov ecx,[eax]     ; завантажити в регістр ecx значення, яке знаходиться
                  ; за адресою eax
```

Звичайно непряма адресація використовується для доступу до масивів. *Стартова адресу масиву зберігається у базовому регістрі* `ebx`**.**

```
my_table times 10 dw 0 ; виділення 10 word з ініціалізацією 0
mov ebx, my_table      ; ефективна адреса my_table в ebx
mov [ebx], dword 1     ; my_table[0] = 1
add ebx, 2             ; адреса ebx = адреса ebx+2
mov [ebx],dword 3      ; my_table[1] = 3
```

Використання непрямої адресації операнду в оперативної пам’яті, при зберіганні його адреси в регістровій пам’яті, суттєво скорочує довжину поля адреси, одночасно зберігаючи можливість використання для фізичної адреси повної розрядності регістра. Недолік цього способу – необхідність додаткового часу для читання адреси операнду. Разом з тим він суттєво підвищує гнучкість програмування. Змінюючи вміст комірки пам’яті або регістра, через які здійснюється адресація, можна, не міняючи команди в програмі, обробляти операнди, які зберігаються в різних адресах.

12. Обчислення адреси під час виконання програми

Існують і інші більш складні способи обчислення виконавчої адреси під час виконання, наприклад відносна, адресація по базі із зміщенням, адресація по базі з індексуванням і адресація з масштабуванням.

Відносна адресація використовується тоді, коли пам’ять логічно розбивається на сегментами. В цьому випадку адреса комірки пам’яті складається з двох частин: адреси початку сегмента (базова адреса), яка зберігається в регістрі, і зміщення, яке визначає положення комірки відносно початку сегмента. Адреса комірки пам’яті визначається як сума адреси сегменту і зміщення. Головний недолік відносної адресації – великий час обчислення виконавчої (фізичної) адреси операнда. Але суттєвою перевагою цього способу адресації є можливість створення “переміщуваних” програм, які можна розмістити в різних частинах пам’яті без зміни команд програми.

При адресації *по базі із зміщенням*, *із індексуванням* і *масштабуванням* виконавча адреса обчислюється під час виконання програми.

Асемблери NASM, MASM, TASM. *Виконавча* (фізична) адреса у пам’яті обчислюється за виразом:

```
[  SELECTOR: BASE + INDEX*SCALE + OFFSET ]
SELECTOR = { CS, DS, ES, SS, FS, GS }
BASE =  { EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP}
INDEX = { EAX, EBX, ECX, EDX, ESI, EDI, EBP }
SCALE = { 1, 2, 4, 8 }
OFFSET = CONSTANT
```

де `SELECTOR` – один з шести сегментних регістрів `cs, ds, es, ss, fs, gs`.

`BASE` – один з регістрів загального призначення `eax/R0D, ecx/R1D, edx/R2D, ebx/R3D, esp/R4D, ebp/R5D, esi/R6D, edi/R7D, ebp, esp`.

INDEX – один з регістрів загального призначення, за винятком esp/R4D.

SCALE – масштабний множник, 1, 2, 4, 8.

OFFSET – любе 32-бітне число.

Асемблер GAS. Адресація пам'яті має наступний вигляд:
OFFSET(%BASE, %INDEX, SCALE)

Виконавча (фізична) адреса обчислюється за виразом:

```
FINAL  ADDRESS =  %BASE + %INDEX * SCALE + OFFSET
```

12.1. Команда lea

lea — мнемоніка від англ. Load Effective Address. Синтаксис:

```
lea  отримувач, джерело
```

Обчислити виконавчу адресу *без звертання* до пам'яті можна командою **lea**:

```
lea eax, [1000+ebx+8*ecx]
```

Команда помножить значення регістра ecx на 8, додасть до добутку значення з регістра ebx та число 1000 і отриманий результат помістить у регістр eax.

Запитання.

1. Програмна модель МП 80x86 і x86-64.
2. Регістри загального призначення. Сегментні регістри.
3. Регістри спеціального призначення eip, eflags.
4. Типи і розміри даних асемблера.
5. Виділення пам'яті для цілих чисел, чисел з плаваючою крапкою і BCD чисел.
6. Виділення пам'яті для символьних стрічок, символьних і стрічкових констант.
7. Порядок розміщення байтів у пам'яті.
8. Директиви розбивки пам'яті на сегменти.
9. Псевдоінструкції оголошення, ініціалізації і резервування пам'яті.
10. Використання позначок в асемблері.
11. Способи адресації пам'яті.
12. Розрахунок виконавчої адреси. Команда lea.

Завдання.

1. Написати програму, яка ініціалізує область пам'яті DATA1 типу byte значеннями 1,2,3,4,5, а потім пересилає їх в область неініціалізованих даних DATA2 (сегмент .bss).

2. Написати програму яка виводить в ОС довжину привітання "Hello student".

3. Написати програму, яка послідовно пересилає символьний рядок "Good bay" з області даних STR1 і символьний рядок "bambino" з області даних STR2 в область даних STR3, яка розміщена в сегмент .bss.

4. Написати програму на С, яка визначає порядок розміщення байтів у пам'яті.

13. Приклади для самостійної роботи

1. 1_1.asm - значення повернення і виклик ядра (32-бітна програма)

```
; nasm –f elf32 1_1.asm –o 1_1.o   - завдання на асемблювання
; ld –m elf_i386 1_1.o –o 1_1
; Сегмент коду
segment .text
global _start
; Точка входу в програму
   _start:
; номер системного виклику
      mov  eax,1
```

```
; параметр системного виклику (код повернення)
      mov  ebx,2
; системний виклик ядра
      int   $0x80
```

Запуск на виконання:

```
>./1_1
>echo $?    # виведення коду повернення
2
```

2. 2_1.c - генерація asm програми із main C-програми

```
/*  gcc -S 2_1.c */
int main(){
  int i,j=0;
  for(i=0;i<8;i++) j=j+i;
  return 0;
}
```

```
>gcc -S 2_1.c
```

/* 2_2.c - генерація asm програми із C-функції */

```
/*   gcc -S 2_2.c  */
unsigned gcd(unsigned x, unsigned y) {
    while (y > 0) {
        unsigned old_x = x;
        x = y;
        y = old_x % y;
    }
    return x;
}
```

```
>gcc -S 2_2.c
```

3. 1_3.asm – виведення повідомлення на екран

```
; Використання переривання Linux INT 80H для виведення тексту.
; Створення бінарного коду:
; завдання на асемблювання для роботи из відладчиком
; nasm -f elf32 -g -F stabs 1_3.asm -o 1_3.o
; завдання на асемблювання без включення відлагоджувальної інформації
; nasm -f elf32 1_3.asm -o 1_3.o
; компонування бінарного виконуваного файлу
; ld -m elf_i386 1_3.o -o 1_3
;
section .data  ; Секція містить ініціалізовані дані
EatMsg  db "MySyscall!!!",10   ; 10 -> '\n'
EatLen  equ $-EatMsg  ; $ - поточна адреса

segment .bss    ; Секція для неініціалізованих даних
segment .text   ; Секція коду
global _start   ; Позначка початку модуля для редактора (linker'a) !
_start:
nop             ; Пуста команда для зневадника gdb
mov eax,4       ; номер системного виклику для sys_write
mov ebx,1       ; параметр - файловий дескриптор 1: Standard Output

mov ecx,EatMsg  ; завантажити адрес повідомлення
mov edx,EatLen  ; завантажити довжину повідомлення
int 80H         ; системний виклик ядра

mov eax,1       ; номер системного виклику для _exit
mov ebx,0       ; завантажити 0-й код повернення програми
int 80H         ; системний виклик ядра
```

Запуск на виконання:

```
>./1_3
MySyscall!!!
```

Лабораторна робота № 2.

2. Синтаксис і формати команд асемблерів NASM і GAS

Мета роботи: вивчення синтаксису і форматів команд асемблерів NASM і GAS.

1. Короткі теоретичні відомості

Асемблери NASM і MASM використовують в командах синтаксис фірми Intel, а GAS – синтаксис фірми AT&T. В ОС Linux і Unix переважно використовуються асемблери NASM, GAS (AT&T), а у ОС Windows – MASM (Intel).

```
>nasm –h   ; отримання довідки про ключі програми
>nasm –hf  ; отримання довідки про формати вихідних файлів
```

1.1. Асемблювання і компонування програм на асемблері

Асемблювати програми для 80x86, x86_64 архітектур можна за допомогою різних асемблерів NASM, GAS (GNU AS є back-end в компілятора gcc), as86, MASM, TASM, FASM. Для програмування критичних частин ядра ОС Linux використовується вбудований в GNU gcc асемблер “as”.

Асемблювання програм NASM

```
>nasm –f <format> <filename> [–o <output>]
>nasm –f elf program.asm  ; асемблювання у об’єктний файл формату ELF
>nasm –f elf32 program.asm –o program.o (64-розрядна ОС, об’єктний файл 32-роз.)
>nasm –f elf –g –F stabs  program.asm ( -g вставити налагоджувальну інформацію для
налагоджувача, -F stabs формат генерування налагоджувальної інформації)
>nasm –f bin program.asm –o myfile.com ; асемблювання у ”сирий” бінарний файл
>nasm –f coff myfile.asm –l myfile.lst ; отримання лістингу з hex-кодом
```

Формат відлагоджувальної інформації

```
nasm -f ELF -y
valid debug formats for 'elf32' output format are ('*' denotes default):
    dwarf     ELF32 (i386) dwarf debug format for Linux/Unix
  * stabs     ELF32 (i386) stabs debug format for Linux/Unix
```

Асемблювання програм GNU AS

```
>as program.s -o program.o
де program.s – файл з початковою програмою
program.o – об’єктний файл
```

Компілювати програми GAS асемблера можна також компілятором gcc

```
> gcc –c program.s
```

Або можна компілювати і компонувати за один крок

```
>gcc –nostdlib program.s –o program  (результат program)
```

Компонування програм

```
>ld program.o -o program   (32-розрядна ОС, виконуваний файл 32-розрядний)
>ld –m elf_i386 program.o -o program   (64-розрядна ОС, виконуваний файл 32-роз.)
де program.o – об’єктний файл
program – виконуваний (бінарний) файл формату ELF
```

Запуск програми на виконання

```
>./program
```

Bash сценарій отримання бінарного файлу для налагоджувача GDB

Записати наступні два рядки у файл **asm.sh**:

```
nasm –f elf –g –F stabs –l $1.lst $1.asm –o $1.o
ld –m elf –Map -$1.map $1.o –o $1
```

Запуск сценарію на асемблювання і компонування програми `program.asm` (вказується тільки ім'я програми без розширення `.asm`):

```
>./asm.sh program
```

Makefile

Записати наступні рядки з іменем Makefile:

```
# Linux makefile
# запуск на виконання: make –k
.SUFFIXES: .o, .asm, .lst
PROG=myprog
$(PROG) : $(PROG).o
     ld –m elf_i386 $(PROG).o –o $(PROG)
$(PROG).o : $(PROG).asm
     nasm –f elf32 $(PROG).asm –o $(PROG).o –l $(PROG).lst
clean:
     rm $(PROG).o $(PROG).lst $(PROG)
```

Примітки. Абзацні відступи рядків задаються натисканням клавіші TAB (у mc подвійним натискання TAB).

Налагоджувач GDB

Для виявлення логічних помилок у виконуваному файлі асемблера використовується консольний або графічний налагоджувач (`debugger`) GDB. Запуск консольного налагоджувача на виконання `gdb <program>`. Довідка про налагоджувач `gdb --help`, довідка для роботи зі налагоджувачем, вводиться в середовищі налагоджувача `(gdb) --help`.

Деякі команди налагоджувача:

`start` – запуск програми на виконання

`stepi` – покрокове виконання

`info reg` – інформація про регістри

`quit` – вихід із налагоджувача

1.2. Шаблон програми на асемблері для налагоджувача

```
segment .data
segment .text
global _start
_start:
nop
; код програми...
; ...
nop
section .bss
```

1.3. Шаблон звичайної програми на асемблері

```
%include "stud_io.inc"    ; замість директиви вставляється файл
segment .data
; сегмент для ініціалізованих даних
;. . .
; сегмент для неініціалізованих даних
segment .bss
; . . .
; сегмент коду
```

```
segment .text
global _main
_main:
; . . .
; код
;
;  завершення програми і передача коду завершення в ОС
mov  eax,1    ; номер системного виклику
mov  ebx,code ; код повернення для системного виклику
int  0x80     ; системний виклик ядра
```

1.4. Графічний налагоджувач KDbg

Команди для асемблювання і компонування можна оформити як сценарії. Приклад сценаріїв для асемблювання (`debug.sh`) і компонування 32-розрядного бінарного файлу для роботи із графічним налагоджувачем KDbg (load.sh). Після асемблювання створюється об'єктний файл з розширенням *.o. Після компонування створюється бінарний файл test, який можна завантажувати у налагоджувач.

Сценарій асемблювання (`debug.sh`):

```
#!/bin/bash
echo "Програма " $1
nasm -f elf32 -g -F stabs $1
```

Запуск на виконання

```
>./debug.sh 1.asm
```

Сценарій компонування (`load.sh`):

```
#!/bin/bash
ld -m elf_i386 $1 -o test
```

Запуск на виконання

```
>./load.sh 1.o
```

Для налогодження програм потрібно запустити графічне середовище KDbg, рис. 1.

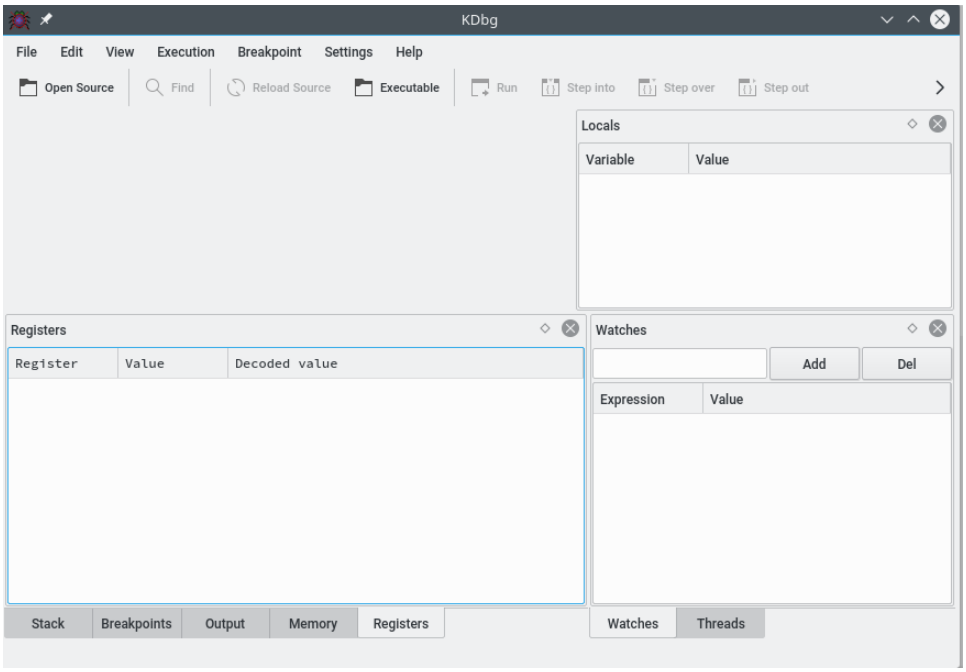


Рисунок 1 – Графічне середовище налагоджувача KDbg

Вибором меню **Open Source** завантажити асемблерну програму і задати в ній точки запинки (Breakpoint/Set/Clear). Вибором меню **Executable** завантажити бінарний файл і запустити його на виконання **Execution/Step into**.

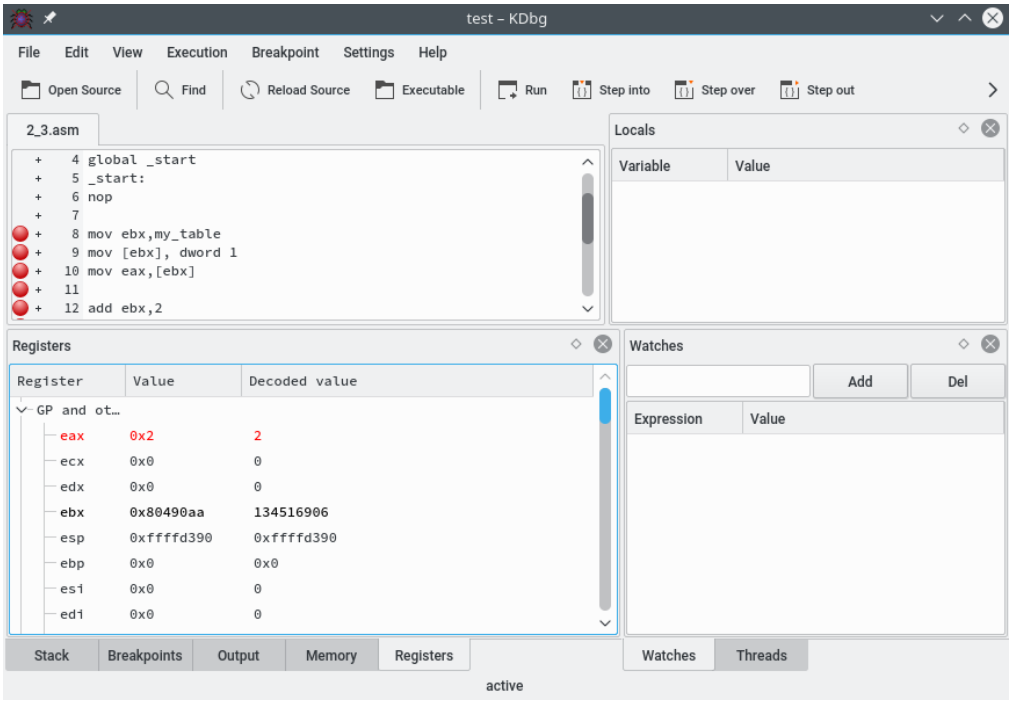


Рисунок 2 – Налагодження програми

1.5. Послідовність створення виконуваних файлів

Виконувані файли з програм асемблера створюються у наступній послідовності:

- початкові файли асемблюються асемблером (`assembler`) в об'єктні файли;
- компонувач (`linker`) збирає об'єктні файли і файли із статичних бібліотек у виконуваний файл з віртуальними (переміщуваними) адресами;
- завантажувач (`loader`) завантажує виконуваний модуль у пам'ять, підключає динамічні бібліотеки і налаштовує фізичні адреси. Отриманий таким чином модуль готовий до виконання.

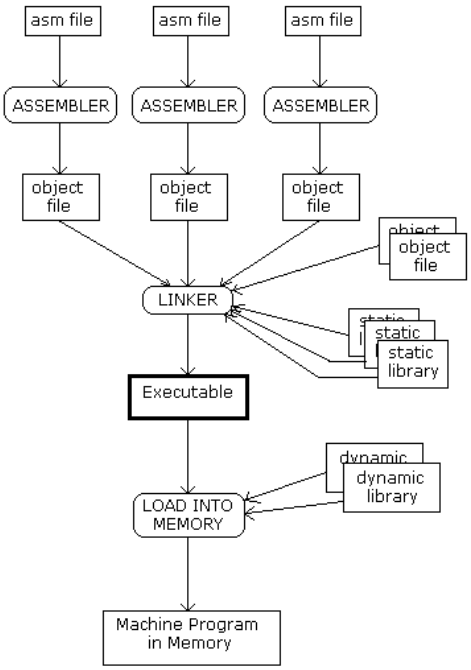


Рисунок 3 – Послідовність створення виконуваних файлів асемблера

2. Огляд команд асемблера

Команди асемблера можна розділити на наступні групи:

• цілочисельної арифметики:
- перетворення типів (`cbw, cwd, cwde, cdq, movsx, movzx`);
- двійкової арифметики (додавання `add, adc, inc`; віднімання `sub, sbb,dec`; множення `mul, imul`; ділення `div, idiv`; зміни знаку `neg`);
- десяткової арифметики (корекції додавання `aaa, daa`; корекції віднімання `aas, das`; корекції множення `aam`; корекції ділення `aad`);
- переставлення байтів (`bswap`);
- інші команди з арифметичним принципом (`cmp, cmpxchg, xadd`);

• логічні команди:
- логічні (`and, or, xor, not, test`);
- сканування бітів `(bsf, bsr);`
- перевірки і модифікації бітів `(bt, btc, btr, bts)`;
- зсуву звичайного (`sar, sal, shl, shr, shld, shrd`);
- зсуву циклічного (`rcl, rcr, rol, ror`);

• логічні операції:
- логічні (`and, or, xor, not`);
- зсуву (`shr, shl`);
- порівняння (`eq, ne, lt, le, gt, ge`);

• команди передачі керування:
- безумовні (`jmp, call, ret`, виклик програмних переривань і повернення з програмних переривань);
- умовні (`jxx`, команди переходів за результатами команди порівняння `cmp`, за станом прапора, за вмістом регістра `cx/ecx`);
- циклів (`loop, loope/loopz, loopne/loopnz`);

• команди для роботи з блоками даних (ланцюжкові):
- пересилання ланцюжків `(movs, movsb, movsw, movsd);`
- порівняння ланцюжків `(cmps, smpsb, cmpsw, cmpsd);`
- сканування ланцюжків `(scas, scasb, scasw, scasd);`
- завантаження елементу з ланцюжка `(lods, lodsb, lodsw, lodsd)`;
- збереження елементу в ланцюжку `(stos, stosb, stosw, stosd);`

2.1. Синтаксис і формати операндів в командах асемблерів NASM, MASM, GAS

Операція	NASM	MASM	GAS
Переслати вміст esi у ebx	mov ebx, esi		movl %esi, %ebx
Переслати вміст si у dx	mov dx, si		movw %si, %dx
Очистити eax регістр	xor eax, eax		xorl %eax, %eax
Переслати безпосереднє значення 10 у регістр al	mov al, 10		movb $10, %al
Переслати вміст за адресою 10 у регістр ecx	mov ecx, [10]	невідомо	movl 10, %ecx
Переслати вміст змінної var у регістр eax	mov eax, [var]	mov eax, dog	movl var, %eax
Переслати адресу змінної var у регістр eax	mov eax, var	невідомо	movl $var, %eax
Переслати безпосереднє байтове значення 10 у пам'ять адресовану регістром edx	mov byte [edx], 10	mov byte ptr [edx], 10	movb $10, (%edx)
Переслати безпосереднє 16-бітове значення 10 у пам'ять адресовану edx	mov word [edx], 10	mov word ptr [edx], 10	movw $10, (%edx)

Переслати безпосереднє 32-бітове значення 10 у пам’ять адресовану edx	mov dword [edx], 10	mov dword ptr [edx], 10	movl $10, (%edx)
Порівняти eax з вмістом комірки пам’яті адресованої як [ebp+8]	cmp eax, [ebp+8]		cmpl $8(%ebp), %eax
Додати до esi значення, адресоване як (eax+ecx*8)	add esi, [eax+ecx*8]		addl (%eax,%ecx,8), %esi
Додати до esi значення, адресоване як (eax+ecx*4+128)	add esi, [eax+ecx*4+128]		addl $128(%eax,%ecx,4), %esi
Додати до esi значення, адресоване як (eax+ecx*4+array)	add esi, [eax+ecx*4+array]		addl array(%eax,%ecx,4), %esi
Додати до esi значення, адресоване як (ecx*2+array)	add esi, [ecx*2+array]		addl array(,%ecx,2), %esi
Переслати безпосереднє значення 4 у комірку пам’яті, яка адресується регістром eax з селектором fs	mov byte [fs:eax], 4	mov byte ptr fs:eax, 4	movb $4, %fs:(%eax)
Перейти в інший сегмент	невідомо	jump far S:O	ljmp $S, $O
Викликати інший сегмент	невідомо	call far S:O	lcall $S, $O
Повернутися з міжсегментного виклику	retf V	ret far V	lret $V
Поширення знаку на увесь регістр ax	cbw		cbtw
Поширення знаку на увесь регістр eax	cwde		cwtl
Поширення знаку на увесь регістр dx:ax	cwd		cwtd
Поширення знаку eax у edx:eax	cdq		cltd
Поширення знаку bh у si	movsx si, bh		movsbw %bh, %si
Поширення знаку bh у esi	movsx esi, bh		movsbl %bh, %esi
Поширення знаку cx у esi	movsx esi, cx		movswl %cx, %esi
Розширення нулем bh у si	movzx si, bh		movzbw %bh, %si
Розширення нулем bh у esi	movzx esi, bh		movzbl %bh, %esi
Розширення нулем cx у esi	movzx esi, cx		movzwl %cx, %esi
100 doublewords, ініціалізованих значенням 8192	times 100 dd 8192	dd 100 dup (8192)	невідомо
Зарезервувати 64 байти пам’яті	resb 64	db 64 dup (?)	невідомо
Ініціалізувати стрічку “Hello World”	db 'Hello, World'		.ascii "Hello, World"
Ініціалізувати стрічку “Hello World” з символом newline і закінченням символом 0	db 'Hello, World', 10, 0		.asciz "Hello, World\n"
Hello world			.string “Hello world”

2.2. Особливості синтаксису і форматів операндів команд асемблера GAS:

- GAS використовує % як префікс перед регістрами
- В командах GAS спочатку іде джерело, а потім отримувач
- GAS вказує розміри операндів в командах (суфіксами b – 8 біт, w – 16 біт, l – 32 біти, q – 64 біти), а не в операндах
- GAS використовує символ $ для безпосередніх значень, а також для адресації змінних
- GAS виводить префікси rep/repe/repne/repz/repnz в окремі рядки перед командами, які він модифікує
- GAS використовує круглі дужки, (%eax) для звернення до комірки пам'яті за вказаною адресою.

В GAS асемблері для позначення типів операндів використовуються суфікси b, w, l, q в командах.

Приклади команд асемблера as.

Переслати адрес 8 в регістр %eax : `movl 8, %eax`

Переслати значення 8 в регістр %eax : `movl $8, %eax`

Переслати вказівник верхівки стеку в регістр %eax : `movl %esp, %eax`

Переслати значення вказівника верхівки стеку в регістр %eax : `movl (%esp), %eax`

Додати 4 до значення вказівника верхівки стеку і значення по цьому адресу переслати в регістр %eax : `movl 4(%esp), %eax`

Зменшити вказівник верхівки стеку на 8 байтів (на два слова) : `subl $8, %esp`

2.3. Синтаксис і формати операндів в командах GAS асемблера

Кожна 32-бітна інструкція має своє 64-бітне розширення, яке вказується суфіксом. Для запису 32-бітних інструкцій використовують суфікс ”l”, а 64-бітних – ”q”.

```
movl  %1,%eax
movq  %1, %rax
```

Виняток з цього правила складають інструкції маніпулювання стеком `push, pop, ret, enter, leave,` які не мають 32-бітових відповідників, але мають 16- і 64-бітові:

```
pushw %ax
pushl %eax  ; error
pushq %rax
pushq %r10
```

Результати 32-бітових інструкцій розширюються нулями у 64-бітових інструкціях, що дозволяє оптимізувати довжину коду:

```
movl $1, %eax    - на 1 байт коротше наступної інструкції
movq $1, %eax
xorg %rax,%rax   - на 1 байт коротше наступної інструкції
mov $0,%rax
andl $5,%eax     - на 1 байт коротше наступної інструкції
andq $5,%erx
```

Безпосередні значення всередині інструкцій залишаються 32-бітовими їх значення розширяються знаком до 64-бітів:

```
addq $1, %rax
addq $0x7fff fffff, %rax      (поширюється біт 0)
addq $0xffff ffff, %rax          -  помилкова інструкція  (поширюється біт 1)
addq $0xffff ffff ffff ffff, %rax
addl $0xffff ffff, %eax
```

Це не стосується до пересилання значень за адресами (зміщень):

```
movl 1, %eax             - 5-ти байтна інструкція
movq 1, %rax             - 7-ми байтна інструкція
movl 0xffff ffff, %eax  - 5-ти байтна інструкція
movq 0xffff ffff, %rax   - 10-ти байтна інструкція
```

Можна використовувати символьні вирази в інструкціях:

```
movl $symbol, %eax  - 5-ти байтна інструкція   # розширення нулем
movl $symbol, %rax  - 7-ми байтна інструкція   # розширення знаком
```

Якщо символьний вираз займає 32-розряди то можна використати 32-бітні інструкції. Якщо потрібно завантажити символьний вираз як 64-бітне значення, то використовується інструкція `movabs`:

```
movabsq $symbol, %rax
```

Переміщення 64-бітових зміщень:

```
movl 0x1, %eax          - завантаження 32-бітового знакового розширення
```

```
movl 0xffff ffff, %eax  - завантаження 64-бітового зміщення
movl symbol, %eax       - завантаження 32-бітового зміщення
movabsl symbol, %eax    - завантаження 64-бітового знакового розширення
```

Завантаження і зберігання 64-бітового зміщення доступне тільки в %eax інструкціях.
Робота з вказівником команд:

```
movl $0x1, 0x10(%rip)   # записати 0x1 в 10-й байт після кінця інструкцій.
movl $0x1, symbol(%rip) # записати 0x1 в за адресом сивола "symbol".
```

2.4. Команди асемблерів GAS і NASM

Gas синтаксис для команди add (додавання)

```
Команда                          Описання
addb constant, destreg8          ;destregn := destregn + constant
addw constant, destreg16         ;destregn, регістр загального призначення (ЗП)
                                 ;n=8, 16, 32 біт
addl constant, destreg32

addb constant, destmem8         ;destmemn := destmemn + constant
addw constant, destmem16        ;destmemn, комірка пам'яті розміром n=8, 16, 32
біт
addl constant, destmem32

addb screg8, destreg8            ;destregn := destregn + srcregn
addw srcreg16, destreg 16        ;destregn, srcregn, регістри загального
призначення n=8, 16, 32 біт
addl srcreg32, destreg 32        ;Регістри мають бути одного розміру

addb srcmem8, destreg8           ;destregn := destregn + srcmemn
addw srcmem16, destreg 16        ;destregn, регістр ЗП n=8, 16, 32 біт
addl srcmem32, destreg 32        ;srcmemn, комірка пам'яті розміром як регістр

addb srcreg8, destmem8           ;destmemn := destmemn + srcregn
addw srcreg16, destmem 16        ;destregn, регістр ЗП n=8, 16, 32 біт
addl srcreg32, destmem 32        ;destmemn, комірка пам'яті розміром як регістр
```

Nasm синтаксис для команди add

```
add destreg, constant            ;destreg := destreg + constant
                                 ;destreg, регістр ЗП n=8, 16, 32 біт

add destmem, constant            ;destmem := destmem + constant
                                 ;destmem, комірка пам'яті розміром n=8, 16, 32
біт

add destreg, srcreg              ;destreg := destreg + srcreg
                                 ;destreg і srcreg, регістр ЗП n=8, 16, 32 біт
                                 ;регістри мають бути одного розміру

add destreg, [srcmem]            ;destreg := destreg + srcmem
                                 ;destreg, регістр ЗП n=8, 16, 32 біт
                                 ;srcmem, комірка пам'яті такого ж розміру

add [destmem], srcreg            ;destmem := destmem + srcreg
                                 ;src, регістр ЗП 8, 16, 32 біт
                                 ;destmem, комірка пам'яті такого ж розміру
```

Gas синтаксис для and

```
andb constant, destreg8         ;destregn := destregn AND constant
andw constant, destreg16        ;destregn, регістр ЗП n=8, 16, 32 біт
```

```
andl constant, destreg32

andb constant, destmem8          ;destmemn := destmemn AND constant
andw constant, destmem16         ;destmemn, комірка пам’яті розміром n=8, 16, 32
біт
andl constant, destmem32

andb srcreg8, destreg8           ;destregn := destregn AND srcregn
andw srcreg16, destreg 16        ;destregn і srcreg, n регістри ЗП n=8, 16, 32 біт
andl srcreg32, destreg 32

andb srcmem8, destreg8           ;destregn := destregn AND srcmemn
andw srcmem16, destreg 16        ;destregn, регістри ЗП n=8, 16, 32 біт
andl srcmem32, destreg 32        ;srcmemn, комірка пам'яті такого ж розміру

andb srcreg8, destmem8           ;destmemn := destmemn AND srcregn
andw srcreg16, destmem 16        ;destregn, регістри ЗП n=8, 16, 32 біт
andl srcreg32, destmem 32        ;destmemn, комірка пам'яті такого ж розміру
```

Nasm синтаксис для and

```
and destreg, constant             ;destreg := destreg AND constant
                                  ;destreg, регістри ЗП 8, 16, 32 біт

and [destmem], type constant      ;destmem := destmem AND constant
                                  ;destmem,  комірка пам'яті 8, 16, 32 біт
                                  ;type := byte, word, dword

and destreg, srcreg               ;destreg := destreg AND srcreg
                                  ;destreg і srcreg, регістри 8, 16, 32 біт
                                  ;Регістри мають бути одного розміру

and destreg, [srcmem]             ;destreg := destreg AND srcmem
                                  ;destreg, регістри ЗП 8, 16, 32 біт
                                  ;srcmem,  комірка пам'яті такого ж розміру

and [destmem], srcreg             ;destmem := destmem AND srcreg
                                  ;srcreg, регістр ЗП 8, 16, 32 біт
                                  ;destmem,  комірка пам'яті такого ж розміру
```

Gas синтаксис для call (виклик підпрограми)

```
call label                        ;виклик підпрограми, яка задана позначкою (label)
                                  ; в програмі
call *reg32                       ;виклик підпрограми за адресою заданою в
                                  ; в 32-бітному регістрі
call *mem32                       ;виклик підпрограми за адресою заданою
                                  ; в 32-бітній комірці пам'яті
```

Nasm синтаксис для call

```
call label                        ;виклик підпрограми, яка задана позначкою (label)
                                  ;в програмі
call reg                          ;виклик підпрограми за адресою заданою в регістрі
                                  ;reg ЗП 8, 16, 32 біт
call mem                          ;виклик підпрограми за адресою заданою в комірці
                                  ;пам'яті mem 8, 16, 32-біт
```

Gas синтаксис для clc

(очистити прапор перенесення в eflag)

```
clc                               ;встановити в 0 прапор carry
cmc                               ;доповнення до 1 (інвертування) прапора carry
```

```
stc                              ;встановити в 1 прапор carry
```

Nasm синтаксис для clc

```
clc                              ;встановити в 0 прапор carry
cmc                              ;доповнення до 1 (інвертування) прапора carry
stc                              ;встановити в 1 прапор carry
```

Gas синтаксис для cmp (порівняння)

```
cmp reg8,constant              ;порівняти регістр regn з константою
cmp reg16,constant             ;regn, регістр ЗП n=8, 16, 32 біт
cmp reg32,constant

cmp byte [mem],constant        ;порівняти memn з константою
cmp word [mem],constant        ;memn, комірка пам'яті розміром n=8, 16, 32 біт
cmp dword [mem],constant

cmp leftreg8, rightreg8        ;порывняти регістри
cmp leftreg16, rightreg16      ;rightregn і leftregn регістри ЗП n=8, 16, 32 біт
cmp leftreg32, rightreg32

cmp reg8,[mem]                 ;порівняти regn з memn
cmp reg16,[mem]                ;regn, регістр ЗП n=8, 16, 32 біт
cmp reg32,[mem32]              ;memn, комірка пам'яті розміром n=8, 16, 32 біт
                                регістр і комірка пам'яті одного розміру

cmp [mem],reg8                 ;порівняти memn з regn
cmp [mem],reg16                ;regn, регістр ЗП n=8, 16, 32 біт
cmp [mem],reg32                ;memn, комірка пам'яті розміром n=8, 16, 32 біт
                               ;комірка пам'яті і регістр одного розміру
```

Nasm синтаксис для cmp

```
cmp regn, constant               ;порівняти reg з константою
                                 ;regn, регістр ЗП n=8, 16, 32 біт

cmp [mem], constant              ;порівняти комірку пам'яті mem з константою
                                 ;mem, комірка пам'яті розміром 8, 16, 32 біт

cmp leftreg, rightreg            ;порівняти регістри leftreg і rightreg .
                                 ;leftreg і rightreg регістр ЗП 8, 16, 32 біт
                                 ;регістри мають бути одного розміру

cmp reg, [mem]                   ;порівняти reg і mem
                                 ;reg, регістр ЗП 8, 16, 32 біт
                                 ;mem, комірка пам'яті розміром 8, 16, 32 біт

cmp [mem], reg                   ;порівняти mem і reg
                                 ;reg, регістр ЗП 8, 16, 32 біт
                                 ;mem, комірка пам'яті розміром 8, 16, 32 біт
```

Gas синтаксис для dec
(зменшення на 1)

```
decb reg8                      ;reg n := regn - 1
decw reg16                     ;regn, регістр ЗП n=8, 16, 32 біт
decl reg32

decb mem8                      ;memn := memn - 1
decw mem16                     ;memn, комірка пам'яті розміром n=8, 16, 32 біт
decl mem32
```

Nasm синтаксис для dec

```
dec reg                         ;reg := reg - 1
                                ;regn, регістр ЗП n=8, 16, 32 біт

dec type [mem]                  ;mem := mem - 1
                                ;memn, комірка пам'яті розміром n=8, 16, 32 біт
                                ;type : = byte, word, dword
```

Gas синтаксис для div (ділення беззнакових чисел),
idiv (ділення знакових чисел)

```
divb reg8           ;al := ax div reg8
                    ;ah := ax mod reg8
                    ;reg8, регістр ЗП 8-біт

divw reg16          ;ax := dx:ax div reg16
                    ;dx := dx:ax mod reg16
                    ;reg16, регістр ЗП 16-біт

divl reg32          ;eax := edx:eax div reg32
                    ;edx := edx:eax mod reg32
                    ;reg32, регістр ЗП 32-біт

divb mem8           ;al := ax div mem8
                    ;ah := ax mod mem8
                    ;mem8, комірка пам'яті 8-біт

divw mem16          ;ax := dx:ax div mem16
                    ;dx := dx:ax mod mem16
                    ;mem16, комірка пам'яті 16-біт

divl mem32          ;eax := edx:eax div mem32
                    ;edx := edx:eax mod mem32
                    ;mem32, комірка пам'яті 32-біт
```

Nasm синтаксис для div (idiv)

```
div reg8            ;al := ax div reg8
                    ;ah := ax mod reg8
                    ;reg8, регістр ЗП 8-біт

div reg16           ;ax := dx:ax div reg16
                    ;dx := dx:ax mod reg16
                    ;reg16, регістр ЗП 16-біт

div reg32           ;eax := edx:eax div reg32
                    ;edx := edx:eax mod reg32
                    ;reg32, регістр ЗП 32-біт

div byte [mem8]     ;al := ax div mem8
                    ;ah := ax mod mem8
                    ;mem8, комірка пам'яті 8-біт

div word [mem16]    ;ax := dx:ax div mem16
                    ;dx := dx:ax mod mem16
                    ;mem16, комірка пам'яті 16-біт

div dword [mem32]   ;eax := edx:eax div mem32
                    ;edx := edx:eax mod mem32
                    ;mem32, комірка пам'яті 32-біт
```

Gas синтаксис для enter (вхід в підпрограму)

```
enter 0, locals16, 0            ;push ebp
                                ;sub ESP, locals
```

Nasm синтаксис для enter

```
enter locals16, 0               ;push ebp
                                ;sub ESP, locals
```

Gas синтаксис для inc (збільшення на 1)

```
incb reg8                       ;regn := regn + 1
incw reg16                      ;regn, регістр ЗП n=8, 16, 32 біт
incl reg32

decb mem8                       ;memn := memn + 1
decw mem16                      ;memn, комірка пам'яті розміром n=8, 16, 32 біт
decl mem32
```

Nasm синтаксис для inc

```
dec reg                         ;reg := reg + 1
                                ;regn, регістр ЗП n=8, 16, 32 біт

dec [type] mem                  ;mem := mem + 1
                                ;memn, комірка пам'яті розміром n=8, 16, 32 біт
                                ;type=byte, word, dword
```

Gas/Nasm синтаксис для умовного переходу jxx після команди cmp

```
ja  label    ;перехід на позначку label, якщо (unsigned) більше (above)
jae label    ;перехід на позначку label, якщо (unsigned) більше-дорівнює (leave)
jb  label    ;перехід на позначку label, якщо (unsigned) менше (below)
jbe label    ;перехід на позначку label, якщо (unsigned) менше-дорівнює
jc  label    ;перехід на позначку label,якщо прапор carry = 1
je  label    ;перехід на позначку label, якщо дорівнює
jg  label    ;перехід на позначку label, якщо (signed) більше (greater)
jge label    ;перехід на позначку label, якщо (signed) більше-дорівнює
jl  label    ;перехід на позначку label, якщо (signed) менше
jle label    ;перехід на позначку label, якщо (signed) менше-дорівнює
jna label    ;перехід на позначку label, якщо (unsigned) не більше
jnae label   ;перехід на позначку label, якщо (unsigned) не менше-дорівнює
jnb label    ;перехід на позначку label, якщо (unsigned) не менше
jnbe label   ;перехід на позначку label, якщо (unsigned) не менше-дорівнює
jnc label    ;перехід на позначку label,якщо прапор carry = 0
jne label    ;перехід на позначку label, якщо не-дорівнює
jng label    ;перехід на позначку label, якщо(signed) не більше
jnge label   ;перехід на позначку label, якщо(signed) не більше-дорівнює
jnl label    ;перехід на позначку label, якщо(signed) не менше
jnle label   ;перехід на позначку label, якщо(signed) не менше-дорівнює
jno label    ;перехід на позначку label, якщо прапор overflow = 0
jns label    ;перехід на позначку label, якщо прапор sign = 0
jnz label    ;перехід на позначку label, якщо прапор zero = 0
jo  label    ;перехід на позначку label, якщо прапор overflow = 1
js  label    ;перехід на позначку label, якщо прапор sign = 1
jz  label    ;перехід на позначку label, якщо прапор zero = 1
jcxz label   ;перехід на позначку label, якщо регістр cx = 0
jecxz label  ;перехід на позначку label, якщо регістр ecx = 0
```

Gas синтаксис для jmp (безумовний перехід)

```
jmp label    ;перехід на позначку label
```

```
jmp *reg32   ;перехід на комірку пам'яті, з адресою в регістрі ЗП reg32
jmp mem32    ;перехід на комірку пам'яті, з адресою в комірці mem32
```

Nasm синтаксис для jmp

```
jmp label ;перехід на позначку label
jmp reg   ;перехід на адресу з регістра reg
jmp [reg] ;перехід на адресу, прочитаної з пам’яті, адреса якої міститься в
          ;регістрі reg ЗП 8,16,32
jmp [mem] ;перехід на адресу, прочитаної з пам’яті, адреса якої міститься в
          ; комірці mem 8,16,32
```

Gas синтаксис для lea

(завантажити регістр ефективною адресою комірки пам'яті)

```
leal mem, reg32       ;reg 32 := address of mem
                      ;reg 32, 32 біт регістр загального призначення
                      ;mem, комірка пам'яті того ж розміру
```

Nasm синтаксис для lea

```
lea reg, [mem]        ;reg := address of mem
                      ;reg - регістр ЗП 8, 16, 32 біт
                      ;mem, комірка пам'яті того ж розміру
```

Gas/Masm синтаксис для leave (вихід з підпрограми)

```
leave             ; mov esp, ebp
                  ; pop ebp
```

Gas синтаксис для mov

```
movb constant, destreg8       ;destregn := constant
movw constant, destreg16      ;destregn, регістр ЗП n=8, 16, 32 біт
movl constant, destreg32

movb constant, destmem8       ;destmemn := constant
movw constant, destmem16      ;destmemn, комірка памяті роміром 8, 16, 32 біти
movl constant, destmem32

movb srcreg8, destreg8        ;destregn := srcregn
movw srcreg16, destreg16      ;destregn і srcregn, регістр ЗП n=8, 16, 32 біт
movl srcreg32, destreg32      ;регістри мають бути одного розміру

movb srcmem8, destreg8        ;destregn :=srcmemn
movw srcmem16, destreg16      ;destregn, регістр ЗП n=8, 16, 32 біт
movl srcmem32, destreg32      ;srcmemn, комірка пам'яті того ж розміру

movb srcreg8, destmem8        ;destmemn := srcregn
movw srcreg16, destmem16      ;srcregn, регістр ЗП n=8, 16, 32 біт
movl srcreg32, destmem32      ;destmemn, комірка пам'яті того ж розміру
```

Nasm синтаксис для mov

```
mov destreg, constant         ;destreg := constant
                              ;destreg, регістр ЗП n=8, 16, 32 біт
mov type [destmem], constant  ;destmem := constant
                              ;destmem, комірка пам’яті розміром 8, 16, 32 біти
                              ;type=byte, word, dword
mov destreg, srcreg           ;destreg := srcreg
                              ;destreg і srcreg,  регістр ЗП n=8, 16, 32 біт
                              ;регістри мають бути одного розміру
mov destreg, [srcmem]         ;destreg := srcmem
```

```
                                ;destreg, регістр ЗП n=8, 16, 32 біт
                                ;srcmemn, комірка пам'яті того ж розміру
mov destmem, [srcreg]           ;desttmem := srcreg
                                ;srcreg, регістр ЗП n=8, 16, 32 біт
                                ;destmemn, комірка пам'яті того ж розміру
```

Nasm синтаксис для movs

```
movsb           ; [edi] := [esi], копіює один байт за адресою [esi] в адресу
                ; [edi].
                ; після копіювання, змінює адреси esi+=1, edi+=1,
                ; якщо df=0, (esi-=1, edi-=1, df=1)
movsw           ; [edi] := [esi], edi+=2, esi+=2,
movsd           ; [edi] := [esi], edi+=4, esi+=4,

rep movsb       ; [edi] := [esi], копіює один байт з адреси [esi] в адресу [edi]
                                  після копіювання, змінює адреси esi+=1, edi+=1,
                                  якщо df=0, (esi-=1, edi-=1, df=1)
                                  кількість ітерацій копіювання задається в рег.
ecx
rep movsw
rep movsd
```

Gas синтаксис для mul (беззнакове множення), **imul** (знакове множення)

```
mulb reg8         ;ax := al * reg8
                  ;reg8, регістр загального призначення 8-біт
mulw reg16        ;dx:ax := ax * reg16
                  reg16, регістр загального призначення 16-біт
mull reg32        ;edx:eax := eax * reg32
                  ;reg32, регістр загального призначення 32-біт

mulb mem8         ;ax := al * mem8
                  ;mem8, комірка пам'яті 8-біт
mulw mem16        ;dx:ax := ax * mem16
                  ;mem16, комірка пам'яті 16-біт
mull mem32        ;edx:eax := eax * mem32
                  ;mem32, комірка пам'яті 32-біт
```

Nasm синтаксис для mul

```
mul reg8          ;ax := al * reg8
                  ;reg8, регістр ЗП 8-біт
mul reg16         ;dx:ax := ax * reg16
                  ;reg16,   регістр ЗП 16-біт
mul reg32         ;edx:eax := eax * reg32
                  ;reg32, регістр ЗП 32-біт

mul byte [mem8]     ;ax := al * mem8
                    ;mem8, комірка пам'яті 8-біт
mul word [mem16]    ;dx:ax := ax * mem16
                    ;mem16, комірка пам'яті 16-біт
mul double [mem32] ;edx:eax := eax * mem32
                    ;mem32, комірка пам'яті 32-біт
```

Gas синтаксис для neg (доповнення значення до 2)

```
negb reg8         ;regn := -regn
negw reg16        ;regn, регістр загального призначення n=8, 16, 32-біт
negl reg32

negb mem8         ;memn := -memn
negw mem16        ;memn, комірка пам'яті розміром n=8, 16, 32-біт
```

```
negl mem32
```

Nasm синтаксис для neg (доповнення значення до 2)

```
neg reg       ;reg := -reg
              ;reg, регістр загального призначення n=8, 16, 32-біт

neg type [mem] mem := -memn
               mem, комірка пам'яті розміром n=8, 16, 32-біт, type=byte,word,dword
```

Gas синтаксис для not (доповнення значення до 1)

```
not reg8     ;regn := not regn
not reg16    ;regn, регістр загального призначення n=8, 16, 32-біт
not reg32

not mem8     ;memn := not memn
not mem16    ;memn, комірка пам'яті розміром n=8, 16, 32-біт
not mem32
```

Nasm синтаксис для not (доповнення значення до 1)

```
not reg       ;reg := not reg
              ;reg, регістр загального призначення n=8, 16, 32-біт

not type [mem] ;mem := not memn
               ;mem, комірка пам'яті розміром n=8, 16, 32-біт,
type=byte,word,dword
```

Gas синтаксис для pushw, pushd, pushfd

```
pushw constant             ;помістити 16-біт константу в стек
*pushl constant            ;помістити 32-біт константу в стек
pushw srcreg16             ;помістити 16-біт регістр в стек
*pushl srcreg32            ;помістити 32-біт регістр в стек
pushw srcmem16             ;помістити 16-біт комірку пам'яті в стек
*pushl srcmem32            ;помістити 32-біт комірку пам'яті в стек
pushfd                     ;помістити копію регістра EFLAGS в стек
```

Nasm синтаксис для push

```
push type constant         ;помістити константу 8,16,32-біт в стек
push srcreg                ;помістити регістр ЗП 8,6,32-біт в стек
push type [srcmem]         ;помістити 8,16,32-біт комірку пам'яті в стек
pushfd                     ;помістити копію регістра EFLAGS в стек (i386)
pushfq                     ;помістити копію регістра RFLAGS в стек (x64)
```

Gas синтаксис для popw, popol, popfd

```
popw srcreg16             ;перемістити із стеку дані в 16-біт регістр
*popl srcreg32            ;перемістити із стеку дані в 32-біт регістр
popw srcmem16             ;перемістити із стеку дані в 16-біт комірку пам'яті
*popl srcmem32            ;перемістити із стеку дані в 32-біт комірку пам'яті
popfd                     ;перемістити із стеку 32-біт дані в регістр EFLAGS
```

Nasm синтаксис для pop, popfd, popfq

```
pop srcreg                ;перемістити із стеку дані 8,16,32-біт регістр
pop type [srcmem]         ;перемістити із стеку дані 8,16,32-біт комірку пам'яті
popfd                     ;перемістити із стеку 32-біт дані в регістр EFLAGS
popfq                     ;перемістити із стеку 64-біт дані в регістр RFLAGS
```

Gas/Masm синтаксис для ret

```
ret              ;взяти із стеку адрес повернення із підпрограми, передати
керування
                 ;по адресу повернення
ret constant16   ;взяти із стеку адрес повернення із підпрограми, додати 16-біт
                 ;константу
                 ;до ESP регістра, передати керування по адресу повернення
```

Gas синтаксис для sub (віднімання)

```
Команда                          Описання
subb constant, destreg8          ;destregn := destregn - constant
subw constant, destreg16         ;destregn, регістр ЗП n=8, 16, 32 біт
subl constant, destreg32

subb constant, destmem8          ;destmemn := destmemn - constant
subw constant, destmem16         ;destmemn, комірка пам'яті розміром n=8, 16, 32
біт
subl constant, destmem32

subb screg8, destreg8            ;destregn := destregn - srcregn
subw srcreg16, destreg 16        ;destregn, srcregn, регістри загального
призначення n=8, 16, 32 біт
subl srcreg32, destreg 32        ;Регістри мають бути одного розміру

subb srcmem8, destreg8           ;destregn := destregn - srcmemn
subw srcmem16, destreg 16        ;destregn, регістр загального призначення n=8,
16, 32 біт
subl srcmem32, destreg 32        ;srcmemn, комірка пам'яті розміром як регістр

subb srcreg8, destmem8           ;destmemn := destmemn - srcregn
subw srcreg16, destmem 16        ;destregn, регістр загального призначення n=8,
16, 32 біт
subl srcreg32, destmem 32        ;destmemn, комірка пам'яті розміром як регістр
```

Nasm синтаксис для sub

```
sub destreg, constant            ;destreg := destreg - constant
                                 ;destreg, регістр ЗП n=8, 16, 32 біт
sub destmem, constant            ;destmem := destmem - constant
                                 ;destmem, комірка пам'яті розміром n=8, 16, 32
біт

sub destreg, srcreg              ;destreg := destreg - srcreg
                                 ;destreg і srcreg, регістр ЗП n=8, 16, 32 біт.
                                 ;Регістри мають бути одного розміру

sub destreg, [srcmem]            ;destreg := destreg - srcmem
                                 ;destreg, регістр ЗП=8, 16, 32 біт
                                 ;srcmem, комірка пам'яті такого ж розміру

sub [destmem], srcreg            ;destmem := destmem - srcreg
                                 ;src, регістр ЗП 8, 16, 32 біт
                                 ;destmem, комірка пам'яті такого ж розміру
```

Запитання.

1. Асемблювання і компонування асемблерних програм на NASM і GNU AS.
2. Шаблони програм на асемблері.
3. Послідовність створення виконуваних файлів асемблера.
4. Створення `makefile` для асемблювання і компонування програм.
5. Призначення налагоджувача GDB і основні його команди.

6. На які групи поділяються команди асемблера;
7. Які особливості синтаксису і форматів команд асемблерів NASM і GAS.
8. Команди цілочисельної арифметики `add, sub, mul, imul, div, idiv, inc, dec;`
9. Логічні команди, сканування, перевірки і модифікації бітів;
10. Команди зсуву звичайного і циклічного;
11. Команди передачі керування `cmp, jxx, jmp, ret, call`;
12. Команди роботи з блоками даних `movs` ;

Завдання.

1. Написати NASM програму, яка: додає і віднімає числа `a` і `b`, `a`=5, `b`=7. Один з результатів повернути в ОС. Запустити програму на виконання у налагоджувачу `gdb`. Виконати вказані операції вручну над бінарними поданням чисел `a` і `b`.

2. Написати NASM програму, яка: множить числа `a`=5, `b`=7; `a`=5, `b`=-7; ділить числа `a`=15, `b`=7; `a`=15, `b`=-7. Один з результатів повернути в ОС. Запустити програму на виконання у налагоджувачу `gdb`. Виконати вказані операції вручну над бінарними поданням чисел `a` і `b`.

3. Написати програму, яка інвертує біти (`not`) і знак чисел (`neg`) `a`=5, `b`=5; Один з результатів повернути в ОС. Запустити програму на виконання у налагоджувачу `gdb`. Виконати вказані операції вручну над бінарними поданням чисел `a` і `b`.

4. Написати NASN програму, яка зчитує стрічку “Hello world” з області пам'яті у сегменті `.data` і записує її у сегмент даних `.bss`.

5. Написати NASM програму, яка знаходить суму значень 5-ти комірок пам’яті типу подвійне слово: `array dd 4,2,8,7,1`. Результат повернути в ОС.

6. Переписати программу 2_7.s на асемблері NASM.

7. Записати у файл `gcd.c` функцію і згенерувати асемблерний код

```
/*  gcd.c - GCD функція
   генерація asm коду: gcc -S gcd.c
*/
unsigned gcd(unsigned x, unsigned y) {
    while (y > 0) {
        unsigned old_x = x;
        x = y;
        y = old_x % y;
    }
    return x;
}
```

3. Приклади для самостійної роботи

1. 2_1.asm – повернення коду завершення в ОС Linux

```
;ПРИЗНАЧЕННЯ: Проста програма яка повертає
;             код завершення в ОС Linux
; nasm -f elf 2_1.asm -o 2_1.o
; ld -m elf_i386 2_1.o -o 2_1

;ВХІД: немає
;
;ВИХІД:   повертає код завершення. Його можна продивитися
;         вводячи в командному рядку консолі echo $?
;         після виконання програми

;ЗМІННІ:
;   eax містить номер системного виклику
;   ebx містить код завершення програми

section .data  ; секція даних
section .text  ; секція коду
```

```
global _start    ; глобальна позначка точки входу в програму
_start:

mov eax,1      ; системний виклик для виходу (sys_exit)
mov ebx,0      ; код завершення програми 0
                ; В ОС код завершення можна вивести командою echo $?
int 0x80       ; виклик ядра ОС

; Запуск програми в консолі:
>./2_1
>echo $?
0
```

2. 2_2.asm – виведення на екран повідомлення Hello

```
; nasm –f elf 2_2.asm –o 2_2.o
; ld –m elf_i386 2_2.o -o 2_2

%include "stud.inc"           ; файл макросів
segment .data                 ; сегмент даних
var1 db "Hello byte World",10 ; ”Hello byte World" + символ ’\n’  (0xA,0xD)
len1 equ $-var1               ; довжина стрічки var
var2 dd "Hello word World",10 ; ”Hello byte World" + символ ’\n’
len2 equ $-var1               ; довжина стрічки var
var3 db "Hello dword World",10 ; ”Hello byte World" + символ ’\n’
len3 equ $-var1               ; довжина стрічки var

segment .text                 ; сегмент коду
global _start                 ; глобальна позначка точки входу в програму
_start:
  nop                  ; порожня команда (для налагоджувача)
  PRINT "Student"      ; виклик макросу друку стрічки
  mov ebx,var          ; переслати в регістр ebx адресу ”HELLO WORD”
  mov eax,len-1        ; переслати в регістр eax довжину стрічки ”HELLO WORD”

  PRINT_MEM var1,len1  ; макрос друку пам’яті з адресою var1 довжиною len1
  PRINT_MEM var2,len2
  PRINT_MEM var3,len3

  FINISH 0             ; виклик макросу завершення програми

; Запуск програми в консолі:
>./2_2
Student
Hello byte World
Hello word World
Hello dword World
```

3. 2_3.asm - цілочисельне множення

```
; nasm –f elf 2_3.asm –o 2_3.o
; ld –m elf_i386 2_3.o -o 2_3

%define myaddr 400h  ; макродиректива
segment .data
var  dw 5            ; резервуваня слова word і присвоєння значення 5
segment .text        ; сегмент коду
global _start
 _start:
dec dword [var]   ; зменшення на 1 значення за адресою var (5-1)
mov eax,2         ; переслати ціле число 2 в регістр eax
mul dword [var]   ; помножити eax на значення за адресою var (2*4)
mov ecx,eax       ; переслати eax в ecx
```

```
mov eax,1         ; системний виклик для виходу (sys_exit)
mov ebx,ecx       ; код завершення програми в ecx
int 0x80          ; виклик ядра ОС

; Запуск програми в консолі:
>./2_3
>echo $?
8
```

4. 2_4.asm - цикл loop

```
; nasm –f elf 2_4.asm –o 2_4.o
; ld –m elf_i386 2_4.o -o 2_4

segment .data     ; сегмент даних
x  dd 1,2,3,4,5   ; резервування і ініціалізація пам'яті dword
section .bss      ; секція буфера
y  resd 5         ; резервування 5-ти dword
segment .text     ; сегмент коду
global _start
 _start:

mov dword [y],1    ; ініціалізація буфера значеннями 1,2,3,4,5
mov dword [y+4],2
mov dword [y+8],3
mov dword [y+12],4
mov dword [y+16],5

mov ecx,5          ; переслати в лічильник циклів ecx число 5
mov esi,y          ; переслати в індексний регістр esi адресу y
mov eax,0          ; переслати в регістр акумулятор eax число 0
lp: add eax,[esi]  ; початок циклу: додати в регістр eax значення за адресою esi
add esi,4          ; збільшити значення адреси в esi на 4
loop lp            ; ecx=ecx-1 і перейти на позначку lp, цикл триває поки ecx != 0
mov ecx,eax        ; результат з eax переслати в ecx

mov eax,1       ; системний виклик для виходу (sys_exit)
mov ebx,ecx     ; код завершення програми в ecx
int 0x80        ; виклик ядра ОС

; Запуск програми в консолі:
>./2_4
>echo $?
15
```

5. 2_5.asm - команда and

```
; nasm –f elf 2_5.asm –o 2_5.o
; ld –m elf_i386 2_5.o -o 2_5

segment .data  ; сегмент даних
segment .text  ; сегмент коду
global _start
 _start:

mov eax,100          ; переслати в регістр eax число     100 -> 0110_0100
mov ecx,0x0000_000f  ; переслати в регістр ecx маску число 15-> 0000_1111
                     ;                                          0000_0100
and ecx,eax          ; побітове AND над eax і ecx, результат 4

mov eax,1    ; системний виклик для виходу (sys_exit)
mov ebx,ecx  ; код завершення програми в ecx
int 0x80     ; виклик ядра ОС

; Запуск програми в консолі:
```

```
>./2_5
>echo $?
4
```

6. 2_6.asm – команда test

```
; nasm –f elf 2_6.asm –o 2_6.o
; ld –m elf_i386 2_6.o -o 2_6

%include "stud.inc"           ; файл макросів
segment .data
segment .text
global _start
 _start:

mov eax,0
mov ecx,0
mov ax,0000_1000b
mov cx,0000_1111b
test ax,cx          ; порозрядна операція AND на регістрами ax і cx
jz m0               ; якщо прапор zero = 0, то перейти на позначку m0
mov cx,1            ; якщо прапор zero = 1 переслати в cx число 1
jmp m1              ; і перейти на позначку m1
m0:
mov cx,0
m1:

;mov eax,1       ; системний виклик для виходу (sys_exit)
;mov ebx,ecx     ; код завершення програми в ecx
;int 0x80        ; виклик ядра ОС
FINISH ecx

; Запуск програми в консолі:
>./2_6
>echo $?
1
```

7. 2_7.s – знаходження максимального з трьох чисел

```
; максимальне з трьох чисел (32-бітна версія)
; gcc -nostdlib 2_7.s -o 2_7
; секція даних
.section .data
   var1: .int 40
   var2: .int 20
   var3: .int 30

; секція коду
.section .text
   .globl _start
   _start:

; move вміст змінних
      movl  (var1), %ecx
      cmpl  (var2), %ecx
      jg    check_var3         # var2 < %ecx
      movl  (var2), %ecx

   check_var3:
     cmpl  (var3), %ecx        # var3 < %ecx
     jg    _exit
     movl  (var3), %ecx

   _exit:
```

```
      movl  $1, %eax
      movl  %ecx, %ebx
      int   $0x80

; Запуск програми в консолі:
>./2_7
>echo $?
40
```

8. Структура об'єктного модуля

Команда nm виводить список символів об'єктного модуля

>nm 2_7.o

```
        1        2   3
0000000000600148 D   __bss_start
0000000000400123 t   check_var3
0000000000600148 D   _edata
0000000000600148 D   _end
0000000000400133 t   _exit
000000000040010c T   _start
000000000060013c d   var1
0000000000600140 d   var2
0000000000600144 d   var3
1 – довжина модуля/ідентифікатора
2 – секція (t – text, локальна; T – text, глобальна; d – даних)
3 - модуль/ідентифікатор
```

9. 2_9.asm – знаходження максимального числа із набору чисел у пам'яті

```
; nasm –f elf 2_9.asm –o 2_9.o
; ld –m elf_i386 2_9.o -o 2_9

;ПРИЗНАЧЕННЯ: Програма знаходить максимальне значення
;             із заданого набору чисел.
;ЗМІННІ: Використовуються наступні регістри:
; edi - індекс для пошуку серед заданого набору чисел
; ebx - знайдене найбільше число
; eax - поточне значення
;
; Розміщення даних в пам'яті:
; data_items - містить елементи даних
; 0 використовується як ознака кінця даних
;
segment .data
data_items dd 3,67,34,222,45,75,54,34,44,33,22,11,66,0
segment .text
  global _start
_start:
mov edi,0       ; завантвжити 0 в індексний регістр
mov eax,[data_items+edi*4] ; переслати в eax перше значення довжиною 4 байти
mov ebx,eax     ; так як це перший елемент то eax містить найбільше число

start_loop:     ; позначка початку циклу
cmp eax,0       ; перевірка, чи не досягнуто кінець даних 0
je loop_exit    ; якщо 0 то вихід з циклу
inc edi         ; збільшити індекс edi
mov eax, [data_items+edi*4]  ; завантажити наступне значення
cmp eax,ebx     ; порівняти значення
jle start_loop  ; перехід на початок циклу, якщо нове значення не більше max
mov ebx,eax     ; оновити найбільше значення
jmp start_loop  ; перехід на початок циклу
loop_exit:

mov eax,1       ; системний виклик для виходу (sys_exit)
```

```
;mov ebx,ecx    ; в ebx найбільше значення
int 0x80        ; виклик ядра ОС

; Запуск програми в консолі:
>./2_9
>echo $?
222
```

Файл макророзширень stud.inc

```
; File stud.inc for Linux
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PRINT string
; PRINT_MEM memory_addr memory_length
; FINISH code
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro PRINT 1
; sys_write:
; 4->eax, addr string->ecx, size->edx
;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; %1 - message
pusha
pushf
jmp %%astr
%%str db %1,10
%%len equ $-%%str
%%astr:
mov eax,4          ; 4=SYS_write
mov ebx,1          ; 1=STDOUT
mov ecx,%%str
mov edx,%%len      ; msg length
int 0x80
popf
popa
%endmacro

;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro PRINT_MEM 2
;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; %1 - memory_addr
;; %2 - memory_len
pusha
pushf

mov eax,4   ; 4=SYS_write
mov ebx,1   ; 1=STDOUT
mov ecx,%1  ; mem addr
mov edx,%2  ; mem length
int 0x80
popf
popa
%endmacro


;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro _syscall_exit 1
;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; %1 - exit code
   mov ebx,%1
   mov eax,1  ; 1 = sys_exit
   int 0x80
%endmacro
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro FINISH 0-1 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; %1  exit code
_syscall_exit %1
%endmacro
```

Лабораторна робота № 3.

3. Команди mov, lea, цілочисельні арифметичні

Мета роботи: вивчення директив препроцесора, практичне застосування методів адресації і арифметичних цілочисельних інструкцій

1. Короткі теоретичні відомості

1.1. Команда `mov`

Команда `mov` одна із найчастіше використовуваних команд асемблера. Команда копіює дані із джерела в отримувач без ніяких перетворень:

```
mov отримувач, джерело
```

Приклади використання команди `mov`:

```
section .data
    count dd 0
    var1  dd 0x00000072
    var2  dd 0x00000001, 0x00000002, 0x00000003
section .text
global _start
_start:
   nop
   mov bl,ch        ; 8-біт
   add di,ax        ; 16-біт
   mov  eax,0x48    ; 32-біт, помістити число 0x00000048 в рег. eax
   mov  eax,var1    ; помістити в eax адресу позначки var1
   mov  eax,[var1]  ; скопіювати в eax значення за адресою var1, тобто 0x00000072
   mov  eax,[var2 + 4]   ; var2 вказує на 0x00000001, розмір одного значення типу
                         ; dword — 4 байта, отже var2 + 4 вказує на 0x00000002
   mov  ecx, 1           ; помістити число 1 в рег. ecx
   mov  eax,[var2+ecx*4] ; скопіювати в %eax перший (нумерація з нуля) елемент
                         ; масиву var2, використовуючи ecx як індексний регістр
   mov  ebx,var2         ; помістити в ebx адресу масиву other_var
   mov  eax,[ebx+4]      ; скопіювати в eax значення за адресою ebx + 4, тобто
                                                              0x00000002
   mov  eax, var2+4      ; скопіювати в eax значення за адресою var2+4, тобто
                                                    число   0x00000002
   mov  [0x15], eax      ; скопіювати за адресою 0x15 вміст eax
   mov  [count],eax      ; скопіювати дані з рег. eax в область пам'яті count
   mov [var],byte 'G'    ; записати у комірку пам'яті var байтовий символ 'G'
   mov  eax,[count]      ; скопіювати дані з області пам'яті count в рег. Eax
   mov  eax,[ebx]        ; скопіювати дані в рег. ebx з області пам'яті, адреса
якої
                            знаходиться у регістрі ebx
   mov  eax,[ebx+ecx+11] ; -"- в регістрах ebx+ecx + 11
   mov  eax,[ebx+ecx+esi]; -"- в регістрах ebx+ecx+esi
```

Команда переставлення місцями старшого і молодшого байту `xchg low,high`:

```
xchg cl,ch
```

Приклади роботи з масивом однобайтних чисел:

```
section .bss
   array resb 256 ;  неініціалізований масив розміром 256 байт
section .text
; ...
   mov ecx, 256     ; кількість елементів масиву -> у лічильник ECX
   mov edi, array   ; адреса масиву в EDI
```

```
   mov al, '@'      ; символ заповнювач в AL
again: mov [edi],al ; занесення коду в черговий елемент
   inc edi          ; збільшення адреси
   dec ecx ;        ; зменшення лічильника
jnz again           ; якщо не нуль, то повторення циклу
```

1.2. Команда lea

Команда `lea` обчислює виконавчу адресу джерела без звернення до пам'яті і поміщає отриману адресу в *отримувач*. *Джерело* має знаходитися в пам'яті (не може бути безпосереднім значенням або регістром).

```
segment .data
var dd 0x00000072
segment .text
        lea eax,var    ; аналогічно mov eax, var
        lea eax, esp+4 ; помістить в eax адресу попереднього елемента в стеку
```

1.3. Команди цілочисельної арифметики

Команди *додавання* `add` і *віднімання* `sub`:

```
add   приймач, джерело
sub   приймач, джерело
```

Принцип роботи:

- `add`: *приймач* = *приймач* + *джерело*
- `sub`: *приймач* = *приймач* - *джерело*.

Приклад:

```
add edx,12        ; збільшити на 12 вміст рег. edx
add eax,ebx       ; до значення рег. eax додати значення рег. ebx
add dword [x],12  ; до значення 4-х байтної комірки x додати 12
sub [x],ecx       ; із значення 4-х байтної комірки x відняти значення рег. ecx
```

В результаті виконання команд `add, sub` виставляються прапори `ZF, SF, OF, CF`.

`ZF=1`, якщо в результаті останньої операції отримано нуль.

`SF=1`, якщо отримано негативне число. Для знакових чисел це означає негативне число, а для беззнакових він не має ніякого змісту.

`OF=1`, якщо відбулося переповнення. Це означає, що в результаті додавання двох позитивних чисел отримано негативне число, або навпаки, при додаванні двох негативних чисел отримано позитивне число.

`CF=1`, якщо для беззнакових чисел відбулося перенесення із старшого розряду або відбулася позика із неіснуючого розряду. В цьому розумінні прапор `CF` аналогічний до пропору `OF` для беззнакових чисел (результат не помістився в розмір операнда або отримано негативний). Для знакових чисел прапор `CF` не має змісту.

Наявність прапора переносу дозволяє організувати додавання і віднімання чисел з врахуванням перенесення або позики із старшого розряду. Для цього існують команди `adc` і `sbb`. По своїй роботі вони аналогічні командам `add` і `sub`, але враховують значення прапора `CF` на момент виконання команди. Команда `adc` додає до кінцевого результату значення прапора `CF`, а команда `sbb` – віднімає. Нехай є два 64-бітні числа, перше записано в пару регістрів `edx:eax`, а друге – в `ebx:ecx`. Тоді додати ці два числа можна командами

```
add eax, ecx ; додавання молодших частин
adc edx, ebx ; додавання старших частин з врахуванням перенесення
```

якщо потрібно їх відняти то використовуються команди

```
sub eax, ecx ; віднімання молодших частин
sbb ebx, ecx ; віднімання старших частин з врахуванням позики
```

Команди *інкременту* `inc`, *декременту* `dec`, *зміни знаку* `neg` і *порівняння* `cmp`.

```
inc операнд
dec операнд
```

Принцип роботи:

- `inc`: збільшує *операнд* на 1.
- `dec`: зменшує *операнд* на 1.

Операнд може бути регістровим або типу ”пам'ять” `[mem]`. Команди встановлюють прапори `ZF, OF, SF`. Приклад:

```
dec eax
inc dword [count]   ; необхідно вказати розмір операнда
```

Приклад організації циклу:

```
mov eax,5
DoMore: dec eax
...
jnz DoMore
```

Команда `not операнд` інвертує біти операнда (операція “доповнення до 1”).

Команда `neg операнд` змінює знак операнда (операція ”доповнення до 2” аналогічна до операції ”унарний мінус”). При цьому не встановлюється знаковий біт, а інвертуються всі біти операнда і до результату інверсії додається біт у самий молодший розряд. Команду `neg` застосовується до знакових чисел.

```
neg al    ; зміна знаку значення в регістрах
neg dx
neg ecx
neg byte [bx]   ; зміна знаку значення у комірці памяті [bx] довжиною byte
neg word [di]   ; зміна знаку значення у комірці памяті [di] довжиною word
neg dword [eax] ; зміна знаку значення у комірці памяті [eax] довжиною dword
```

Порівняння команд `neg` і `not` (інверсія бітів):

```
section .text
global _start
_start:
  mov ecx,0
  mov cl,0000_0101b   ;   5
  ;not cx ; 1111_1010b -> 250
  neg cx  ; 1111_1011b -> 251 -> -5
  mov eax,1        ; повернення значення ecx в ОС
  mov ebx,ecx
  int 0x80
```

Сума числа і його “доповнення до 2” дає нуль:

```
mov eax,42
neg eax
add eax,42
```

В асемблері x86 знакові числа зберігаються у формі “доповнення до 2”, яке задає віддаль числа від 0 в обох напрямках, як позитивному, так і негативному.

```
0xffffffff (-1), 0x00000001 (1),
0xfffffffe (-2), 0x00000002 (2),
0xfffffffd (-3), 0x00000003 (3),
…
0x10000010(-126), 0x01111110 (126).
0x10000001(-127), 0x01111111 (127).
0x10000000 (-128)
```

Таблиця 1 – **Діапазони знакових чисел**

Розмір	Найбільше негативне		Найбільше позитивне	
	десяткове	шістнадцяткове	десяткове	шістнадцяткове
8	-128	80h	127	7Fh
16	-32768	8000h	32767	7FFFh

32	-2147483648	80000000h	2147483647	7FFFFFFFh

Команда поширення знакового біту movs:

```
movsx reg16, reg8/mem8      ; 8-й біт розширюється  до 16-го біту
movsx reg32, reg8/mem8      ; 8-й біт розширюється  до 32-го біту
movsx reg32, reg16/mem16    ; 16-й біт розширюється до 32-го біту
```

Приклад копіювання знакового числа без і з поширенням знаку:

```
mov ax,-42                          mov ax,-42
mov ebx,eax ; ebx=65494 (0xffd6)    movsx ebx,ax   ; ebx=-42
```

Команда `cmp` (від слова “compare” – “порівняти”) здійснює такі ж обчислення, як і команда `sub`, але результат нікуди не записує. Команда викликається тільки для встановлення прапорів, звичайно за нею слідує команда умовного переходу.

Команди цілочисельного *множення* `mul` і *ділення* `div` мають один операнд, який задає *другий множник* в командах множення і *дільник* в командах ділення, причому цей операнд може бути тільки регістровим або типу “пам’ять”. Для задання першого множника і діленого використовується неявний операнд, в якості якого виступають регістри `eax/ax/al`, а при необхідності і регістрові пари `dx:ax, edx:eax`. Необхідно зауважити, що результат множення двох *n*-розрядних чисел може поміститися тільки в 2*n*-розрядному регістрі результату. В табл. 1 показано розміщення неявного операнду і результату операцій цілочисельного множення і ділення в залежності від розрядності явного операнду.

Таблиця 2 – **Розміщення операндів і результату операцій `mul` і `div`**

Розряди явного операнду	Множення			Ділення		
	явний 1-й множник	неявний 2-й множник	результат множення	неявне ділене	частка	залишок
8	reg8/mem8	al	ax	ax	al	ah
16	reg18/mem16	ax	dx:ax	dx:ax	ax	dx
32	reg32/mem32	eax	edx:eax	edx:eax	eax	edx

Для множення беззнакових чисел використовується команда `mul`, а для множення знакових – `imul`. Команди `mul` і `imul` встановлюють прапори `CF` і `OF`, якщо старша половина результату дорівнює нуль.

Для ділення беззнакових чисел використовується команда `div`, а для ділення знакових – `idiv`. Значення прапорів після операцій цілочисельного ділення не визначені.

Приклади:

```
segment .text
   mov eax,72
   inc eax  ; в eax число 73
   dec eax  ; в eax число 72
   mov eax,48
   add eax,16  ; в eax число 64
   mov al,5
   mov bl,5
   mul bl    ; в регістрі ax добуток al×bl=25
```

Запитання. Яким буде результат виконання наступного коду на Сі:

```
char x, y;
x = 250;
y = 14;
x = x + y;
printf("%d", (int) x);
```

Відповідь. 250 + 14 = 264, більше, ніж може поміститися в один байт. Тому програма надрукує 8. Додавання виконується в двійковій системі.

```
  11111010       250
+ 00001110     + 14
----------       ---
1 0000 1000      264
  |      |
  |<------>|
     8 біт
```

В результаті додавання виникає перенесення із старшого біту результату, яке називають переповненням. В Сі переповнення не може бути перехоплено, але в мікропроцесорі ця ситуація реєструється і її можна обробити. Коли виникає переповнення, встановлюється прапор `cf`. Команди умовного переходу `jc` і `jnc` аналізують стан цього прапору.

```
   mov  ah,0     ; ah = 0
   mov  al,250   ; al = 250
   add  al,14    ; al = al + 14, виникає переповнення cf=1, а в al число 8
   jnc  no_carry ; якщо переповнення не було, перейти на позначку no_carry
   mov  ah,1     ; ah = 1
no_carry:
                 ; ax = 264 = 0x0108
```

Цей код видає вірну суму в регістрі `ax` з врахуванням переповнення, якщо воно відбулося.

Команда `lea` для арифметичних операцій

Для виконання деяких арифметичних операцій можна використати команду `lea`. Вона обчислює адрес свого операнда-джерела і поміщає цей адрес в операнд-отримувач. Адреса операнда формується так

```
зміщення(база, індекс, множник)
[база + індекс*множник + зміщення]
```

Обчислений адрес буде дорівнювати *база* + *індекс* × *множник* + *зміщення*. Враховуючи це можна отримати команду з двома операндами-джерелами і одним результатом:

```
mov  eax,10
mov  ebx,7
lea ecx,[eax+5]         ; ecx = eax + 5 = 15
lea ecx,[eax-3]         ; ecx = eax - 3 = 7
lea ecx, [eax+ebx]      ; ecx = eax + ebx × 1 = 17
lea ecx, [eax+ebx*2]    ; ecx = eax + ebx × 2 = 24
lea ecx, [eax+ebx*2+1]  ; ecx = eax + ebx × 2 + 1 = 25
lea ecx, [eax*8]        ; ecx = eax × 8 = 80
lea ecx, [eax+eax*2]    ; ecx = eax + eax × 2 = eax × 3 = 30
lea ecx, [eax+eax*8]    ; ecx = eax + eax × 8 = eax × 9 = 90
```

При додаванні командою `add` результат записується на місце одного з доданків. Перевага команди `lea` в тому, що вона не перезаписує операнди-джерела. Так можна додати константу до регістра і записати в інший регістр, додати два регістри і записати в третій регістр. Також `lea` можна застосувати для множення регістра на цілі числа (3 і 9), як показано вище.

Запитання.

1. Варіанти використання команди `mov`.
2. Використання команд `xchg, lea`.
3. Команди цілочисельної арифметики `add, sub, adc, sbb`.
4. Команди цілочисельної арифметики `inc, dec, neg, cmp`.
5. Як отримати “доповнення до 1” і “доповнення до 2” знакового числа.

6. Для чого потрібна команда поширення знакового біту `movsx`.

7. Команди цілочисельної арифметики `mul, imul, div, idiv`.

8. Використання команди `lea` для арифметичних операцій.

Завдання.

1. Оголосити і ініціалізувати вектор однобайтних чисел значеннями від 1 до 16. Скопіювати у масив в секції `.bss` тільки ті числа, залишок від ділення яких на 3 дорівнює 1.

2. Оголосити неініціалізований однобайтний масив розміром 256. Заповнити масив, як двовимірний – непарні рядки нулями, а парні рядки – одиницями.

3. Оголосити два цілих числа (одне додатне `0x0000_1234` і одне від'ємне `0x1000_0012`) типу `dword`. Обчислити їх суму, різницю, добуток, частку та залишок від ділення.

4. Нехай є два 128-бітні числа, перше записано в пару регістрів `edx:eax (0x9876_1234: 0x9876_4321), ebx:ecx (0x5432_1234: 0x5432_4321)`, а друге – в `esp:ebp (0x1234_1234: 0x1234_1212), esi:edi (0x4321_4321: 0x4321_4343)`. Додати ці два числа використовуючи команди `add, adc`. Відняти ці два числа використовуючи команди `sub, sbb`.

5. Використовуючи команду `lea`, базовий регістр `ebx`, індексний регістр `esi`, переписати у пам'яті вектор з шістнадцяти 4-байтних чисел з адреси `mas1` у адресу `mas2`.

6. Написати программу для обчислення виразу z=(x*y)/(x+y). Всі числа 16-бітові цілі із знаком.

2. Приклади для самостійної роботи

Для виконання завдань потрібні файли:

```
driver.c
c_decl.h
asm_io.asm
asm_io.o      ( nasm -f elf -d ELF_TYPE asm_io.asm -o asm_io.o )
asm_io.inc
stud.inc
```

1. 3_1.asm – зміна регістру букв стрічки

```
; nasm -f elf 3_1.asm -o 3_1.o
; ld -m elf_i386 3_1.o -o 3_1
section .data
  var db 'HELLO WORD',10
  len equ $-var
section .text
global _start
_start:
  nop           ; для gdb
  mov ebx,var   ; адресу var у ebx
  mov eax,len-1 ; к-ть букв

DoMore: add byte [ebx],32  ; додати 32 до значення ebx, зробивши букви малими
  inc ebx       ; інкремент
  dec eax       ; декремент
jnz DoMore      ; перехід поки eax != 0

  mov eax,4     ; виведення значення var, можна замінити макросом PRINT_MEM var
  mov ebx,1
  mov ecx,var
  mov edx,len
  int 80h

  mov eax,1 ; 0-код завершення програми. Фрагмент можна замінити макросом FINISH 0
  mov ebx,0
  int 80h
  nop           ; для gdb
```

>./3_1
```
hello@word
```

2. 3_2.asm – зміна регістру букв з використання макросів
```
; nasm –f elf 3_2.asm –o 3_2.o
; ld –m elf_i386 3_2.o –o 3_2

%include "stud.inc"
segment .data
var db 'HELLO WORLD',10
len equ $-var
global _start
segment .text
_start:
  nop
  PRINT "Student"

  mov ebx,var
  mov eax,len-1
m1: add [ebx],byte 32
  inc ebx
  dec eax
  jnz m1

  PRINT_MEM var,len
  FINISH 0
```

>./3_2
```
Student
hello@world
```

3. 3_3.asm – цілочисельне множення чисел
```
; nasm –f elf 3_3.asm –o 3_3.o
; ld –m elf_i386 3_3.o –o 3_3
%include "stud.inc"
segment .data
   var db 2 ; 0x2_01
   res dd 0
segment .text
global _start
_start:
  nop         ; для gdb
  mov eax,0
  ;
  mov ax,64   ; множене    100_0000 -> 0x40
  mov bx,2    ; множник   0000_0010 -> 0x10
  mul bx      ; результат 100_00000 -> 0x80 -> ax

  ;mov ax,64
  ;mul byte [var]  ; 0x80-> ax

  mov ebx,eax
  mov eax,1
  int 0x80
  nop         ; для gdb
```

>./3_3
>echo $?
```
128
```

4. 3_4.asm - програма знаходження суми двох чисел
```
; В програмі вводиться два числа і друкується їх сума;
```

```
; Using Linux and gcc:
; nasm -f elf -d ELF_TYPE asm_io.asm -o asm_io.o
; nasm -f elf 3_4.asm
; gcc 3_4.o driver.c asm_io.o -o 3_4

%include "asm_io.inc"
; initialized data is put in the .data segment
;
segment .data
; Позначки для повідомлень діалогу і результату
;
prompt1 db    "Enter a number: ", 0       ; признак закінчення стрічки
prompt2 db    "Enter another number: ", 0
outmsg1 db    "You entered ", 0
outmsg2 db    " and ", 0
outmsg3 db    ", the sum of these is ", 0
;
; неініціалізовані дані поміщаються в .bss сегмент
segment .bss
;
; Позначки подвійних слів в яких будуть зберігатися введені дані
;
input1  resd 1
input2  resd 1
;
; код поміщається в .text segment
;
segment .text
        global  asm_main
asm_main:
        enter   0,0               ; пролог
        pusha

        mov     eax, prompt1      ; виведення повідомлення 1
        call    print_string
        call    read_int          ; читання integer
        mov     [input1], eax     ; зберігання його в input1
        mov     eax, prompt2      ; виведення повідомлення 2
        call    print_string
        call    read_int          ; читання integer
        mov     [input2], eax     ; зберігання його в input2
        mov     eax, [input1]     ; eax = dword at input1
        add     eax, [input2]     ; eax += dword at input2
        mov     ebx, eax          ; ebx = eax
        dump_regs 1               ; виведення dump значень регістрів
        dump_mem 2, outmsg1, 1    ; виведення dump пам'яті
                                  ; виведення результатів
        mov     eax, outmsg1
        call    print_string      ; вивести повідомлення 1
        mov     eax, [input1]
        call    print_int         ; вивести input1
        mov     eax, outmsg2
        call    print_string      ; вивести повідомлення 2
        mov     eax, [input2]
        call    print_int         ; вивести input2
        mov     eax, outmsg3
        call    print_string      ; вивести повідомлення 3
        mov     eax, ebx
        call    print_int         ; вивести суму (ebx)
        call    print_nl          ; вивести new-line

        popa
        mov     eax, 0            ; повернутися в C
```

```
        leave                     ; епілог
        ret
```

```
    >./3_4
    Enter a number:  12
    Enter another number:13
```

5. 3_5.asm - програма множення і ділення цілих чисел

```
;
; To create executable:
; nasm -f elf –d ELF_TYPE asm_io.asm –o asm_io.o
; nasm -f elf 3_5.asm
; gcc 3_5.o driver.c asm_io.o -o 3_5

%include "asm_io.inc"

segment .data
;
; Вихідні повідомлення
prompt          db    "Enter a number: ", 0     ; признак завершення стрічки
square_msg      db    "Square of input is ", 0
cube_msg        db    "Cube of input is ", 0
cube25_msg      db    "Cube of input times 25 is ", 0
quot_msg        db    "Quotient of cube/100 is ", 0
rem_msg         db    "Remainder of cube/100 is ", 0
neg_msg         db    "The negation of the remainder is ", 0

segment .bss
input   resd 1
segment .text
        global  asm_main
asm_main:
        enter   0,0               ; пролог
        pusha

        mov     eax, prompt
        call    print_string

        call    read_int
        mov     [input], eax

        imul    eax               ; edx:eax = eax * eax
        mov     ebx, eax          ; зберегти результат в ebx
        mov     eax, square_msg
        call    print_string
        mov     eax, ebx
        call    print_int
        call    print_nl

        mov     ebx, eax
        imul    ebx, [input]      ; ebx *= [input]
        mov     eax, cube_msg
        call    print_string
        mov     eax, ebx
        call    print_int
        call    print_nl

        imul    ecx, ebx, 25      ; ecx = ebx*25
        mov     eax, cube25_msg
        call    print_string
        mov     eax, ecx
        call    print_int
        call    print_nl

        mov     eax, ebx
```

```
        cdq                     ; ініціалізувати edx знаковим розширенням
        mov     ecx, 100        ; не ділити явно на значення
        idiv    ecx             ; edx:eax / ecx
        mov     ecx, eax        ; зберегти залишок в ecx
        mov     eax, quot_msg
        call    print_string
        mov     eax, ecx
        call    print_int
        call    print_nl
        mov     eax, rem_msg
        call    print_string
        mov     eax, edx
        call    print_int
        call    print_nl

        neg     edx             ; доповнення залишку
        mov     eax, neg_msg
        call    print_string
        mov     eax, edx
        call    print_int
        call    print_nl

        popa
        mov     eax, 0          ; повернення в C
        leave
        ret
```

```
>./3_5
Enter a number: 9
```

6. 3_6.asm - програма знаходження простих чисел

```
;
; Створення бінарного файлу:
; nasm -f elf –d ELF_TYPE asm_io.asm –o asm_io.o
; nasm -f elf 3_6.asm
; gcc 3_6.o driver.c asm_io.o -o 3_6
;
; Works like the following C program:
; #include <stdio.h>
;
;int main()
;{
;  unsigned guess;          /* current guess for prime      */
;  unsigned factor;         /* possible factor of guess     */
;  unsigned limit;          /* find primes up to this value */
;
;  printf("Find primes up to: ");
;  scanf("%u", &limit);
;
;  printf("2\n");    /* treat first two primes as special case */
;  printf("3\n");
;
;  guess = 5;        /* initial guess */
;  while ( guess <= limit ) {
;    /* look for a factor of guess */
;    factor = 3;
;    while ( factor*factor < guess && guess % factor != 0 )
;      factor += 2;
;    if ( guess % factor != 0 )
;      printf("%d\n", guess);
;    guess += 2;    /* only look at odd numbers */
;  }
;  return 0;
;}
```

```
%include "asm_io.inc"

segment .data
Message  db "Find primes up to: ", 0  ; признак закінчення стрічки
segment .bss
Limit    resd    1              ; знаходження простих чисел до limit
Guess    resd    1              ; поточне передбачення
segment .text
        global  asm_main
asm_main:
        enter   0,0             ; setup routine
        pusha

        mov     eax,  Message
        call    print_string

        call    read_int            ; scanf("%u", & limit );
        mov     [Limit], eax

        mov     eax, 2              ; printf("2\n");
        call    print_int
        call    print_nl
        mov     eax, 3              ; printf("3\n");
        call    print_int
        call    print_nl

        mov     dword [Guess], 5    ; Guess = 5;  Передбачення = 5

while_limit:                        ; while ( Guess <= Limit )
        mov     eax,[Guess]
        cmp     eax, [Limit]
        jnbe    end_while_limit     ; використовуємо jnbe так як числа беззнакові
        mov     ebx, 3              ; ebx = 3;
while_factor:
        mov     eax,ebx
        mul     eax                 ; edx:eax = eax*eax
        jo      end_while_factor    ; if answer won't fit in eax alone
        cmp     eax, [Guess]
        jnb     end_while_factor    ; if !(factor*factor < guess)
        mov     eax,[Guess]
        mov     edx,0
        div     ebx                 ; edx = edx:eax % ebx
        cmp     edx, 0
        je      end_while_factor    ; if !(guess % factor != 0)

        add     ebx,2               ; factor += 2;
        jmp     while_factor
end_while_factor:
        je      end_if              ; if !(guess % factor != 0)
        mov     eax,[Guess]         ; printf("%u\n")
        call    print_int
        call    print_nl
end_if:
        mov     eax,[Guess]
        add     eax, 2
        mov     [Guess], eax        ; guess += 2
        jmp     while_limit
end_while_limit:

        popa
        mov     eax, 0           ; повернення в C
```

```
    leave
    ret
```

```
>./3_6
Find primes up to: 156
```

Лабораторна робота № 4.

4. Безумовні та умовні переходи, цикли, логічні побітові команди, бітові операції, зсуви

Мета роботи: вивчення безумовних та умовних переходів, бітових операцій та логічних побітових команд

1. Безумовні та умовні переходи

1.1. Безумовні переходи

В системі команд процесора i386 всі команди передачі керування, в залежності від дальності "переходу", поділяються на наступні групи:

1. **Далекі** (`far`) переходи передають керування у фрагмент програми розміщеної в *іншому сегменті*. У "плоскій" моделі пам'яті є тільки один сегмент, тому в ній далекі переходи не використовуються.

2. **Близькі** (`near`) переходи передають керування у довільне місце всередині одного сегменту. Фактично такі переходи явно змінюють значення вказівника команд `EIP`. У "плоскій" моделі пам'яті цей перехід дозволяє перейти у довільне місце адресного простору одного сегменту.

3. **Короткі** (`short`) переходи дозволяють перейти на 127 байт вперед і 128 байт назад. У машинному коді такої команди зміщення задається одним байтом, що і зумовлює відповідні обмеження.

Команда умовного переходу `jmp` (від слова "`jump`"). Команда має один операнд, який визначає адресу переходу. При написанні команди можна явно вказати вид потрібного переходу, поставивши після команди слово `far, near або short`. Якщо цього не зробити, асемблер вибирає тип переходу за замовчуванням (для безумовних `near`, а для умовних `short`).

Якщо в команді безумовного переходу задається безпосередня адреса (як позначка), то такий перехід є прямим. Для непрямих переходів адреса задається у регістровому операнді або операнді типу "пам'ять". Приклад команд безумовних переходів:

```
jmp cycle  ; перехід за адресою позначки cycle
jmp label  ; перехід за адресою позначки label
jmp [label] ; перехід за адресою, яка знаходиться в комірці пам'яті
              з адресою позначки label
jmp eax    ; перехід за адресою з регістра EAX
jmp [eax]  ; перехід за адресою, яка знаходиться в комірці пам'яті
              з адресою, яка знаходиться у регістрі eax
```

Якщо позначка знаходиться близько до команди безумовного переходу, то використовується короткий перехід:

```
label:
. . .
jmp short label
```

1.2. Прості умовні переходи

Прості команди умовного переходу переходять за вказаною адресою у випадку, якщо один з прапорів встановлений (дорівнює одиниці) або скинутий (дорівнює нулю). Імена цих команд утворюються з букви `j` (від слова "`jump`"), першої букви назви прапора (наприклад, `z` для прапора `zf=1`) і, можливо, вставленої між ними букви `n` (від слова "`not`"), якщо перехід необхідно здійснити за умови рівності `zf=0`.

Такі команди умовного переходу ставлять зразу після арифметичних операцій або команди `cmp`, наприклад

```
cmp eax, ebx     ; порівняти значення регістрів
jz are_equal     ; якщо вони однакові перейти на позначку are_equal
```

Таблиця 1 – Найпростіші команди умовних переходів

Команда	Умова переходу	Команда	Умова переходу
jz	zf=1	jnz	zf=0
js	sf=1	jns	sf=0
jc	cf=1	jnc	cf=0
jo	of=1	jno	of=0
jp	pf=1	jnp	pf=0

1.3. Переходи за результатами порівнянь

За результатами команди порівняння `cmp` можуть встановлюватися два прапори, наприклад `sf=1 (знак), of=0 (переповнення)` (переповнення не було, число отримане негативне) або `sf=0, of=1` (число позитивне, але це результат переповнення, а в дійсності результат негативний). Тобто, потрібно аналізувати ситуації коли `sf≠of`. Для цього використовуються команди умовного переходу за результатами порівнянь `cmp a,b`.

Таблиця 2 – Переходи за результатами порівнянь

Команда	Перехід якщо	Вираз	Умова переходу	Синонім
нерівність для знакових чисел				
je	equal	a=b	zf=1	jz
jne	not equal	a≠b	zf=0	jnz
нерівності для знакових чисел				
jl jnge	less not greater or equal	a<b	sf≠of	
jle jng	less or equal not greater	a≤b	sf≠of або zf=1	
jg jnle	greater not less or equal	a>b	sf=0 і zf=0	
jge jnl	greater or equal not less	a≥b	sf=of	
нерівності для беззнакових чисел				
jb jnae	below not above or equal	a<b	cf=1	jc
jbe jna	below or equal not above	a≤b	cf=1 або zf=1	
ja jnbe	above not below or equal	a>b	cf=0 і zf=0	
jae jnb	above or equal not below	a≥b	cf=0	jnc

Приклад використання команд:

```
.text
    mov eax,15
    cmp eax,15      ; порівняння
    jne not_equal   ; якщо операнди не рівні, перейти на позначку not_equal
```

```
        ; команди для eax = 15
        jmp out
 not_equal:
        ; команди для eax != 15
 out:
```

Крім команд переходів за результатами порівнянь `jcc`, існує родина команд `setcc`. Вони перевіряють стан прапорів так як `jcc`. За значенням прапорів операнд встановлюється в 1, якщо перевірювана умова `cc` істинна, і в 0, якщо умова фальшива. Команди `setcc` працюють тільки з операндами, які зберігаються в пам'яті і мають розмір один байт. Синтаксис команд `setcc`: `setcc операнд`.

Приклади команд `setcc`:

```
        cmp   a,5
        seta  al      ; a>5 (above)
        cmp   b,10
        setb  bl      ; a<5  (below)
        and   al,bl
        cmp   c,0
        sete  bl      ; a==5 (equal)
        or    al,bl
        jz    is_false
is_true:
        ...
is_false:
```

1.4. Команда циклів `loop`

Команда **`loop`** призначена для організації циклів з наперед відомою кількістю ітерацій.

Синтаксис команди `loop`:

```
позначка:
..
loop позначка
```

Принцип роботи:

- зменшити значення регістра `ecx` на 1;
- якщо `ecx` = 0, передати керування наступній за `loop` команді;
- якщо `ecx` ≠ 0, передати керування на *позначка*.

Як лічильник ітерацій команда `loop` використовує регістр **`ecx`**, в який перед початком циклу записується потрібне число ітерацій. Сама команда `loop` виконує дві дії: зменшує на одиницю значення в регістрі `ecx` і, якщо результат не нульовий, переходить на задану позначку. Команда `loop` здійснює тільки короткі переходи на позначки, які розміщені від самої команди не далі як на 128 байт. Приклад підрахунку суми елементів масиву із 10 подвійних слів із проходженням масиву з початку:

```
segment .data
   array db 1,2,3,4,5,6,7,8,9,10
segment .tex
   mov ecx, 10     ; кількість ітерацій
   mov esi, array  ; адреса першого елементу
   mov eax, 0      ; початкове значення суми
lp: add eax, [esi] ; додати число до суми, esi індекс масиву array
   add esi, 4      ; адреса наступного елементу
   loop lp         ; зменшення лічильника і якщо ecx не 0, перехід на lp
```

Замість команди `loop` можна використати дві команди:

```
dec ecx
jnz lp
```

У наступному прикладі суми обчислюється без регістра esi, але у цьому випадку елементи масиву вибираються з кінця:

```
mov ecx,10
mov eax,0
lp: add eax,[array+4*ecx-4]
loop lp
```

Команда loop має дві модифікації. Команда loope (синонім loopz) здійснює перехід, якщо в регістрі ecx не нуль і прапор zf встановлений. Команда loopne (синонім loopnz) здійснює перехід, якщо в регістрі ecx не нуль і прапор zf скинутий.

Для аналізу стану регістра cx/ecx є дві додаткові команди умовного короткого переходу: jcxz – умовний перехід, якщо в регістрі cx нуль, jecxz - умовний перехід, якщо в регістрі ecx нуль.

Команди не враховують прапори. Вони використовуються для запобігання виконання циклу у випадку, коли значення регістра ecx є нульовим на початку циклу. Якщо на момент входу у цикл ecx=0, то виконується тіло циклу, а потім від лічильника віднімається 1. В результаті лічильник отримає значення максимально можливого цілого числа 2^{32}. Для запобігання таких ситуацій перед циклом потрібно поставити команду jecxz:

```
; присвоєння значення ecx
jecxz lpq
lp: ; тіло циклу
; ...
loop lp
lpq:
```

На асемблері можна створювати довільні цикли з післяумовою, подібні до do{} while(); в мові програмування С. Фрагмент коду такого циклу на асемблері:

```
loop_start:                 /* початок циклу                        */
        /* тіло циклу */
        cmp  ...            /* порівняння */
        je    loop_end      /* команда умовного переходу для виходу з  циклу */
        jmp   loop_start    /* інакше повторити цикл спочатку      */
loop_end:
```

На асемблері також можна створювати довільні цикли з передумовою, подібні до while(){}в мові програмування С. Фрагмент коду такого циклу на асемблері:

```
loop_start:                 /* початок циклу                         */
        cmp   ...           /* порівняння */
        je    loop_end      /* умова переходу для виходу з циклу   */
        /* тіло циклу */
        jmp   loop_start    /* перехід на перевірку умови циклу    */
loop_end:
```

2. Логічні побітові команди і бітові операції

2.1. Логічні побітові команди

Для роботи з комірками пам'яті і регістрами використовуються логічні побітові команди. Найпростішими з них є двомісні команди **and, or, xor** (виключне або), **not** (інверсія).

```
and   джерело, приймач
or    джерело, приймач
xor   джерело, приймач
not   операнд
test  операнд_1, операнд_2
```

Команда `xor` часто використовується для обнуління регістра, так як вона коротша від команди `mov`:

```
xor eax, eax  ;  розмір команди 2 байти
mov eax, 0    ;  розмір команди 5 байт
```

Коли застосовувати команду `xor` замість `mov`? Команда `xor` коротша, а значить, займає менше місця в процесорному кеші, менше часу тратиться на декодування. Але команда `xor` встановлює прапори. Тому, якщо потрібно зберегти стан прапорів, використовується команда `mov`.

Іноді для обнуління регістра застосовують команду `sub`. Вона також встановлює прапори.

```
sub eax,eax  ; тепер eax == 0
```

Команди `and`, `or` і `xor` подібні до інструкцій мови програмування С `&`, `|`, `^`. Всі ці команди встановлюють прапори `zf, cf i pf` у відповідності з результатом.

Команда `not` інвертує кожний біт операнду (змінює на протилежний), так як інструкція мови програмування С `~`.

Якщо необхідно перевірити наявність в числі одного із заданих бітів, використовується команда `test`. Команда виконує побітове `I` над бітами операндів, як і команда `and`, але результат нікуди не записує, а виставляє тільки прапор `zf`, якщо всі біти результату нульові. Команду `test eax,eax` часто використовують замість команди `cmp eax,0`.

Команда `test` називається також командою логічного порівняння, так як дозволяє перевірити, чи встановлені задані біти, наприклад:

```
test al, 0b0000_1001

 0010_0101              0010_1000              0010_0110
 0000_1001              0000_1001              0000_1001
 ----------             ----------             ----------
 0000_0001  zf=0        0000_1000  zf=0        0000_0000  zf=1
```

```
test al,0b0000_1000   ; чи встановлений 3-й (від нуля) біт?
je not_set
   ; потрібні біти встановлені
not_set:
   /* біти не встановлені */
Командою test можна порівнювати значення регістра з нулем:
test eax,eax
je is_zero
   ; eax != 0 */
is_zero:
   ; eax == 0
```

Рекомендується використовувати команду `test` замість `cmp` для порівняння значення регістра з нулем.

2.2. Бітові операції над виразами

Над виразами асемблера можна виконувати:

- арифметичні операції **+** (додавання), **-** (віднімання), ***** (множення), **/** (ділення беззнакове), **//** (ділення знакове), **%** (ділення за модулем беззнакове), **%%** (ділення за модулем знакове);

- унарні оператори **+** (плюс), **-** (мінус), **~** (інверсія, доповнення до 1), **!** (логічне заперечення), **SEG** (отримання адреси сегменту);

- бітові операції **&** (and), **|** (or), **^** (xor);

- операції зсуву << (вправо), >> (вліво);

- операції обчислення поточних адрес **$**, **$$**.

Вирази повинні бути абсолютними, тобто такими, числові значення яких можуть бути обчислені транслятором.

Синтаксис бітових операцій над виразами:

```
[+ | - | ! | seg] Вираз1 [… & | | | ^ …] [[+ | - | ! | seg] Вираз2
```

Обчислення адреси сегмент:позначка:

```
mov ax,seg symbol
mov es,ax
mov bx,symbol ;   es:bx -> seg::symbol
```

Бітова операції ^ (xor):

```
mask equ 1000_0011b
     mov al,mask^01h   ; переслати в регістр al маску
                       ; з інвертованим правим бітом al=1000_0010
```

Операція зсуву (<<):

```
mov ax, 2<<5 ; зсунути число 2 на 5 розрядів вліво і переслати число 64 в ax
```

2.3. Обчислення поточних адрес `$, $$`

Асемблер обчислює поточну адресу виразу з операндом **`$`**:

```
section .data
  var db 'HELLO WORD',10
  len equ $-var    ; різниця адрес поточної команди і позначки var
```

Так нескінченний цикл можна записати так `jump $`.

Асемблер обчислює початок поточної секції операндом **`$$`**. Так виаз `$-$$` визначає зміщення поточної команди від початку поточного сегменту.

3. Операції зсуву

Часто приходиться виконувати операції побітового зсуву. Операції побітового зсуву реалізовуються командами:

- логічного зсуву;
- арифметичного зсуву;
- простого циклічного зсуву;
- циклічного зсуву через прапор CF.

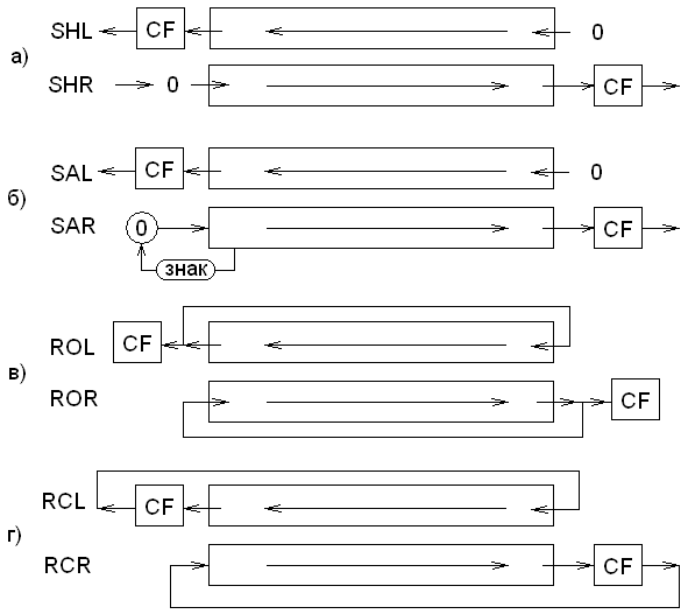


Рис. 1. Операції побітового зсуву: а) логічного; б) арифметичного; в) простого циклічного; г) циклічного зсуву через прапор CF

Команди **логічного зсуву** `shr` (`shift right`), `shl` (`shift left`) мають два операнди, перший вказує, що зсувати, а другий – на скільки бітів зсувати. Перший операнд може бути регістровим або типу "пам'ять". Другий операнд може бути безпосереднім числом від 1 до 31 або регістром `cl`.

```
shl операнд, кількість_зсувів    ; зсув вліво
shr операнд, кількість_зсувів    ; зсув вправо

mov ax, 0C123H
shl ax, 1  ; зсунути на 1 біт  вліво,  ax = 8246H, CF = 1
shr ax, 1  ; зсунути на 1 біт  вправо, ax = 4123H, CF = 0
shr ax, 1  ; зсунути на 1 біт  вправо, ax = 2091H, CF = 1
mov ax, 0C123H
shl ax, 2  ; зсунути на 2 біти вліво,  ax = 048CH, CF = 1
mov cl, 3
shr ax, cl ; зсунути на 3 біти вправо, ax = 0091H, CF = 1
```

Кожний "висовуваний" біт попадає в прапор `cf`, причому звільнений біт з другої сторони заповнюється 0. Таким чином, в прапорі `cf` виявляється самий останній "висунутий" біт. Це вповні допустимо для роботи з беззнаковими числами, але числа із знаком будуть оброблені невірно, так як знаковий біт може бути втрачений. Для беззнакових чисел зсув на `n` біт вліво еквівалентний множенню на 2^n, а зсув вправо – цілочисельному ділення на 2^n із відкиданням залишку.

Для роботи з знаковими числами використовуються команди **арифметичного зсуву** `sal` (`shift arithmetic left`), `sar` (`shift arithmetic rigth`). Ці команди дозволяють швидко множити і ділити числа на 2^n.

```
sal операнд, кількість_зсувів
sar операнд, кількість_зсувів

mov ax, 0C123H
sal ax, 1 ; ax = 8246H, CF = 1
sal ax, 1 ; ax = 048CH, CF = 1
sar ax, 2 ; ax = 0123H, CF = 0
```

Команди `sal` зсовує вліво вміст операнду на задану кількість бітів. Справа в молодші біти записуються нулі. Команда `sal` не зберігає знаку, але встановлює прапор `cf` у випадку зміни знаку черговим висовуваним бітом.

Для зсуву вправо використовується команда `sar`. Вона зсовує вправо вміст операнду на задану кількість бітів. Зліва в операнд записуються нулі. Команда зберігає знак, відновлюючи його після зсуву кожного чергового біту. Операція арифметичного зсуву вправо еквівалентна діленню на 2^n з відкиданням залишку для знакових цілих чисел.

Циклічні зсуви поділяються на прості циклічні зсуви і циклічні зсуви через прапор CF.

```
ror операнд, кількість_зсувів
rol операнд, кількість_зсувів

mov ax, 0C123H
rol ax, 1 ; ax = 8247H, CF = 1
rol ax, 1 ; ax = 048FH, CF = 1
rol ax, 1 ; ax = 091EH, CF = 0
ror ax, 2 ; ax = 8247H, CF = 1
ror ax, 1 ; ax = C123H, CF = 1
```

При простому циклічному зсуві біти, які “висовуються” з однієї сторони записуються у звільнені біти з іншої сторони. При цьому в прапор `cf` записується самий останній “висунутий” біт.

Існує ще один вид зсувів – циклічний зсув через прапор `cf`. Ці команди використовують прапор `cf` як продовження операнду.

```
rcr операнд, кількість_зсувів
rcl операнд, кількість_зсувів

mov ax, 0C123H
clc ; clear the carry flag (CF = 0)
rcl ax, 1 ; ax = 8246H, CF = 1
rcl ax, 1 ; ax = 048DH, CF = 1
rcl ax, 1 ; ax = 091BH, CF = 0
rcr ax, 2 ; ax = 8246H, CF = 1
rcr ax, 1 ; ax = C123H, CF = 0
```

Система команд останніх моделей процесорів Intel містить додаткові команди зсуву подвійної точності:

```
shld операнд_1, операнд_2, лічильник_зсувів
shrd операнд_1, операнд_2, лічильник_зсувів
```

Команда `shld` зсовує біти `операнду_1` вліво і заповнює його біти справа бітами `операнда_2`. Кількість зсовуваних бітів задається значенням `лічильник_зсувів`, яке може бути в діапазоні 0…31. Це значення може задаватися безпосередньо операндом або міститися в регістрі `CL`. Значення `операнд_2` при зсувах не змінюється. Аналогічно працює і команда `shrd`, але `операнд_1` зсовується вправо.

Запитання.

1. Команди безумовних переходів.
2. Прості умовні переходи.
3. Переходи за результатами порівнянь.
4. Команда циклів `loop`.
5. Логічні команди `and, or, xor, not, test.`
6. Команди простого побітового зсуву `shl, shr`
7. Команди арифметичного побітового зсуву `sal, sar`
8. Команди циклічного побітового зсуву `rol, ror, rcl, rcr.`

Завдання.

1. Написати програму на асемблері в якій обчислюється вираз:

```
a=10;a=5;a=0;
if (a >= b) c=a+b;
else c=a-b;
```

2. Написати програму на асемблері для підрахунку суми перших 20 непарних чисел.

3. Написати програму на асемблері для підрахунку числа бітів в символьному рядку “Hello world”.

4. Написати програму на асемблері, яка з масиву цілих чисел розміром 5×5, оголошених в сегменті `.data`

```
segment .data
mas  db 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25
```

формує в пам’яті сегменту `.bss` вектор діагональних елементів

```
segment .bss
diag resb 5
```

5. Написати програму на асемблері в якій присвоюється значення масиву розміром 5×5 за виразом з використання ланцюжкових команд:

```
for(i=0;i<5i++)
fof(j=0;j<5;j++) mas[i][j]=i+j;
```

2. Приклади для самостійної роботи

1. 4_1.asm - програма для обчислення суми чисел від 1 до 10.

```
; nasm –f elf 4_1.asm –o 4_1.o
; ld –m elf_i386 4_1.o –o 4_1
%include ”student.inc”
segment .text
global _start
_start:
   nop             ; для наладника
   mov eax,0x1a2b  ; регістр із значенням  1_1010_0010_1011
   mov ebx,0
   mov bl, 0       ; bl буде містити число одиничних бітів
   mov ecx, 32     ; ecx лічильник циклів
count_loop:
   shl eax, 1      ; зсув біта в прапор перенесення CF(carry flag)
   jnc skip_inc    ; if CF == 0, перейти на skip_inc
   inc bl
skip_inc:
   loop count_loop
   mov eax,bl
   FINISH ebx
   nop              ; для наладника
```

```
./4_1
echo $?
7
```

2. 4_2.asm - програма для обчислення суми чисел від 1 до 10.

```
; nasm –f elf 4_2.asm –o 4_2.o
; ld –m elf_i386 4_2.o –o 4_2
%include ”stud.inc”
segment .text
global _start
_start:
   nop
   mov  ebx,0    ; в ebx буде результат, тому його треба обнулити
   mov  ecx,10   ; 10 кроків циклу
sum:
   add  ebx,ecx  ; ebx=ebx+ecx
   loop  sum
   ; ebx = 55, ecx = 0
   FINISH ebx
```

```
./4_2
$?
55
```

3. 4_3.asm - програма пошуку найбільшого елементу в масиві

```
; nasm –f elf 4_1.asm –o 4_1.o
; ld –m elf_i386 4_1.o –o 4_1
%include ”stud.inc”
segment .data
array dd 10, 15, 148, 12, 151, 3, 72
array_end:
segment .text
globl _start
_start:
        nop
        mov eax,0         ; в eax буде зберігатися результат,
                          ; спочатку найбільше значення — 0
        mov ebx,array     ; в ebx адресу першого елемента масиву
loop_start:               ; початок циклу
```

```
        cmp [ebx],eax      ; порівняти поточний елемент масиву з
                           ; поточним найбільшим значенням в eax
        jbe less           ; якщо поточний елемент масиву <= найбільшому,
                           ; пропустити наступний код (jbe – для беззнакових)
        mov eax,[ebx]      ;  якщо елемент масиву більший поточного, значить, це
                           ;  значення і є новий максимальним
less:
        add ebx,4          ; збільшити ebx на 4 байти, розмір елементу масиву
        cmp ebx,array_end ;порівняти адреси поточного елементу і кінця масиву
        je loop_end        ; якщо вони рівні, вийти з циклу
        jmp loop_start     ; інакше повторити цикл
loop_end:
        mov ebx,eax
        FINISH ebx         ; повернення числа з ebx в Linux і завершення програми
```

```
./4_3
$?
151
```

Лабораторна робота № 5.

5. Ланцюжкові команди

Мета роботи: вивчення і робота з ланцюжковими командами

1. Короткі теоретичні відомості

Асемблер має спеціальні команди для роботи з масивами. Їх називають ланцюжковими командами або командами для обробки рядків символів. У даному випадку під рядком символів розуміють послідовність байтів. Ланцюжкові команди дозволяють виконувати дії над блоками даних розміром байт, слово, подвійне слово. Ланцюжкові команди використовують індексні регістри **esi** (індекс джерела, англ. source index) і **edi** (індекс приймача, англ. destination index) для виконання операцій і автоматично збільшують або зменшують їх значення. Прийнято, що в регістрі `esi` знаходиться вказівник на наступний необроблений елемент ланцюжка-джерела, а в регістрі `edi` – вказівник на наступний елемент ланцюжка-приймача. Напрямок перегляду ланцюжка задається прапором напрямку **df** у регістрі eflags, який визначає чи індексні регістри збільшуються чи зменшуються: 0 – збільшення (переміщення вперед), 1 – зменшення (переміщення назад).

Для модифікації значення прапора напрямку `df` є дві команди:

`cld` – очистити прапор (встановити в 0);

`std` – встановити прапор в 1.

2. Читання і записування пам'яті

Найпростішими ланцюжковими командами є читання **lods** і записування пам'яті **stos.**

Команда `loads` копіює елемент з ланцюжка-джерела, який адресується регістром `esi` в регістр `eax`/`ax`/`al`. Після цього значення регістра `esi` збільшується або зменшується (в залежності від прапора напрямку) на значення, яке дорівнює розміру елемента ланцюжка.

Команда `stos` записує вміст регістра `eax`/`ax`/`al` в ланцюжок-приймач, який адресується регістром `esi`. Після цього значення регістра `edi` збільшується або зменшується (в залежності від прапора напрямку) на значення, яке дорівнює розміру елемента ланцюжка.

lodsb al=[ds:esi] esi=esi ± 1	stosb [es:esi]=al edi=edi ± 1
lodsw ax=[ds:esi] esi=esi ± 2	stosw [es:esi]=ax esi=esi ± 2
lodsd eax=[ds:esi] esi=esi ± 4	stosd [es:esi]=eax esi=esi ± 4

Пара інструкцій `lods/stos` може бути замінена однією інструкцією **movs.**

movsb byte [es:edi] = byte [ds:esi] esi=esi ± 1 edi=edi ± 1
movsw word [es:edi] = word [ds:esi] esi=esi ± 2 edi=edi ± 2
movsd dword [es:edi] = dword [ds:esi] esi=esi ± 4 edi=edi ± 4

При використанні формату **movs адреса_приймача,адреса_джерела** асемблер сам визначає за типом операндів, яку з трьох форм команд вибрати. Після виконання команди вміст регістрів esi і edi збільшується (прапор df=1) або зменшується (прапор df=0) на розмір елемента ланцюжка.

3. Команди cmps, scas і префікс інструкцій rep

Для порівняння і пошуку елементів ланцюжків використовуються команди:

cmps – порівнює елемент ланцюжка-джерела і ланцюжка-приймача.

scas – шукає заданий елемент у ланцюжку.

Команда cmps порівнює елементи, які адресуються регістрами esi і edi.

Команда scas порівнює вміст регістра eax/ax/al з елементами, які адресуються регістром edi (виконує віднімання eax/ax/al – [es:edi], результат не записується, але прапори встановлюються). Адрес ланцюжка має бути поміщений в регістр edi.

Після того, як ці команди виконали свою основну дію, вони збільшують/зменшують індексні регістри на розмір елемента ланцюжка.

cmpsb compares byte [ds:esi] and byte [es:edi] esi=esi ± 1 edi=edi ± 1
cmpsw compares word [ds:esi] and word [es:edi] esi=esi ± 2 edi=edi ± 2
cmpsd compares dword [ds:esi] and dword [es:edi] esi=esi ± 4 edi=edi ± 4
scasb compares al and [es:edi] edi=edi ± 1
scasw compares ax and [es:edi] edi=edi ± 2
scasd compares eax and [es:edi] edi=edi ± 4

Ці команди обробляють тільки один елемент ланцюжка. Для обробки всього ланцюжка використовуються *префікси повторень* інструкцій:

```
rep
repe/repz
repne/repnz
```

Вони записуються перед ланцюжковими командами, наприклад: rep scas. Префікс виконує інструкцію задане число разів, яке попередньо записується в регістр ecx. На кожному кроці циклу значення регістра ecx автоматично зменшується на 1.

Префікси повторень вказуються перед ланцюжковою командою. Розміщення префікса rep перед ланцюжковою командою заставляє її виконуватися у циклі. Префікс repe задає ітерації до першого співпадіння елементів, а префікс repne – до першого неспівпадіння елементів. Рішення про циклічне виконання ланцюжкової команди приймається на основі стану регістра ecx/cx (repe/repne) або прапора zf (repz/repnz).

Так команда rep stow заміняє блок команд:

```
Clear:  mov [es:di],ax ; копіювання AX в es:di
inc di                 ; збільшення індекса на 2 для word у буфері,
inc di
```

```
dec cx                  ; зменшення cx на 1
jnz Clear               ; цикл поки cx не 0
```

Пефікси повторень обробляють різні прапори:

- repnz (або repz) виконує інструкцію поки прапор zf=1 або ecx не дорівнює 0. Аналізуючи значення регістра ecx, можна встановити точну причину виходу з циклу: якщо ecx дорівнює нулю, значить zf завжди був встановлений, і весь ланцюжок пройдений до кінця, якщо ecx більше нуля - значить, прапор zf в якийсь момент був скинутий.
- repne (або repe) виконує інструкцію поки регістр ecx не дорівнює 0.

Фрагмент програми порівняння двох блоків пам'яті.

```
segment .text
  cld
  mov esi, block1   ; адресу першого блоку
  mov edi, block2   ; адреса другого блоку
  mov ecx, size     ; розмір блоку в байтах
  repe cmpsb        ; повторювати поки ZF=1
  je equal          ; якщо ZA=1, блоки однакові
  ; код, якщо блоки не однакові
  jmp onward
equal:
  ; код, якщо блоки однакові
onward:
```

Запитання.

1. Ланцюжкові команди для читання/записування пам'яті lods, stos, movs.
2. Ланцюжкові команди порівняння і пошуку cmps, scas.
3. Префікси ланцюжкових команд rep, repe/repz, repne/repnz.

Завдання.

1. Написати програму, яка переписує символьний рядок "Hello world" із секції даних .data у секцію буферних даних .bss з використанням ланцюжкових команд.

2. Написати програму, яка переписує у циклі від 1 до 100 значення індексного регістра esi у секцію даних .data з використанням команд повторень.

3. Написати програму пошуку числа у масиві цілих чисел від 1 до 100 чисел, які діляться на 7 без залишку з використанням ланцюжкових команд.

4. Написати програму пошуку адреси початку слова "level" у ланцюжку байтів "Студенти комп'ютерної інженерії вивчають низькорівневу (low level) мову програмування ассемблер" з використанням ланцюжкових команд.

2. Завдання для самостійної роботи

1. **5_1.asm** - читання значення елементів масиву array1 і записування їх у буфер array2.

```
; nasm -f elf32 -d ELF_TYPE asm_io.asm -o asm_io.o
; nasm -f elf32 5_1.asm -o 5_1.o
; gcc 5_1.o driver.c asm_io.o -o 5_1

%include "asm_io.inc"
segment .data
array1 dd 1,2,3,4,5,6,7,8,9,10
outmsg db "Значення буфера ",0
segment .bss
array2 resd 10
segment .text
global _asm_main
_asm_main:
   nop
```

```
   enter 0,0  ; пролог
   pusha

   cld  ; очистка прапора напрямку
   mov esi,array1
   mov edi,array2
   mov ecx, 10
   mov eax,outmsg
   call print_string
lp:
   lodsd
   stosd
   mov eax,[edi]
   call print_int
   call print_nl
loop lp

   popa
   mov eax,0  ; повернення в C
   leave
   ret
```

```
  >./5_1
  Значення буфера: 1,2,3,4,5,6,7,8,9,10
```

2. **5_2.asm** - програма на асемблері GAS, яка збільшує кожний байт символьного рядка `str_in` на 1, тобто замінює *a* на *b*, *b* на *c*, і т. д.

```
     ; nasm -f elf32 5_2.asm -o 5_2.o
     ; ld -m elf_i386 5_3.o -o 5_3

     %include "stud.inc"
     segment .data
     printf_format db "%s\n"
     str_in db "abc123()1234567890",0
     str_length equ $-str_in
     segment .bss
     str_out resb str_length
     segment .text
     global _start
     _start:
             cld
             mov  esi,str_in     ; адреса ланцюга джерела
             mov  edi,str_out    ; адреса ланцюга приймача
             mov  ecx,str_length-1  ; довжина рядка без нульового байту

             mov al,0
     m1:     lodsb               ; завантажити байт al з джерела esi
             inc al              ; приріст значення al
             stosb               ; запис al у приймач  edi
             loop m1

             movsb                ; копіювати нульовий байт признак завершення
рядка

             mov eax,4
             mov ebx,1
             mov ecx,str_out
             mov edx,str_length
             int 0x80

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

```
>./5_2
bcd234)*23456789:1
```

3. **5_3.asm** – заповнення буфера значенням 0xff

```
; nasm -f elf 5_3.asm -o 5_3.o
; ld -m elf_i386 5_3.o -o 5_3
segment .bss
array resd 10
segment .text
  global _start
_start:
  nop

  cld  ; очистка прапора напрямку
  mov edi, dword array
  mov ecx, 10
  mov eax,0xff
  rep stosd

  mov eax,1
  mov ebx,0
  int 80h
  nop
```

```
>./5_3
> echo $?
0
```

4. **5_4.asm** – пошук числа 12 в масиві чисел типу dword

```
; ld -V  режим емуляції
; nasm -f elf 5_4.asm -o 5_4.o
; ld -m elf_i386 5_4.o -o 5_4
segment .data
array dd 10,20,34,16,0,1,12,98,66,99
segment .text
global _start
_start:
  cld
  mov edi,array   ;вказівник на початок масиву array
  mov ecx,10      ; число елементів масиву
  mov eax,12      ; число для пошуку
lp:
  scasd
  je found
  loop lp
  ; код, якщо не знайдено
  jmp onward
found:
  sub edi,4     ; edi тепер вказує на 12 у масиві
  sub edi,array ; зміщення від початку коду
  mov eax,edi   ; неявне ділене
  mov ebx,4     ; явний дільник, результат в eax
  div ebx
onward:

  mov ebx,eax
  mov eax,1
  int 80h

>./5_4
> echo $?
6
```

5. **5_5.s** – функція memcopy на асемблері GAS:

```
# as --32 -g 5_5.s -o 5_5.o
# ld -m elf_i386 5_5.o -o 5_5
# або gcc -nostdlib 5_5.s -o 5_5
.data
printf_format:
        .string "%s\n"

str_in:
        .string "abcdef_123456789"
        .set str_in_length, .-str_in
.bss
str_out:
        .space str_in_length
.text

/* void *my_memcpy(void *dest, const void *src, size_t n); */

.global _start
_start:
       nop
       pushl $str_in_length
       pushl $str_in
       pushl $str_out
       call memcpy
       addl $12,%esp

       mov $4,%eax
       mov $1,%ebx
       mov $str_out,%ecx
       mov $str_in_length,%eax
       int $0x80

       mov $1,%eax
       mov $0,%ebx
       int $0x80
       nop

my_memcpy:
        nop
        pushl %ebp
        movl  %esp, %ebp

        pushl %esi
        pushl %edi

        movl  8(%ebp), %edi     /* ланцюжок-призначення              */
        movl  12(%ebp), %esi    /* ланцюжок-джерело                  */
        movl  16(%ebp), %edx    /* довжина                           */

        movl  %edx, %ecx
        shrl  $2, %ecx          /* ділити на 2^2 = 4; тепер в %ecx
                                   знаходиться кількість 4-байтних
                                   частин                         */
        rep movsl
        movl  %edx, %ecx
        andl  $3, %ecx          /* $3 == $0b11, залишити тільки два
                                   молодших біти, тобто залишок від
                                   ділення на 4                   */
        jz    1f                /* якщо результат 0, пропустити
                                   ланцюжкову команду             */
        rep movsb
```

```
1:

        movl  8(%ebp), %eax     /* повернути dest                        */
        popl  %edi
        popl  %esi

        movl  %ebp, %esp
        popl  %ebp
        nop
        ret

   >./5_5
   1234567890987654
```

6. **5_6.s** – реалізація функції strlen

У функції strlen порівнюється кожний байт ланцюжка з 0, якщо знайдено 0, то повертається кількість ненульових байтів. Як лічильник використовується регістр %ecx, який автоматично змінює всі префікси. Префікси зменшують лічильник і припиняють виконання команди, коли %ecx дорівнює 0. Тому перед ланцюжковою командою поміщається в %ecx число 0xffffffff, і цей регістр буде зменшуватися в процесі виконання ланцюжкової команди. Отриманий результат буде в оберненому коді, тому використовується команда not для інвертування всіх бітів. Після цього зменшується результат на 1, так як нульовий байт також був порахований.

```
# 5_6.s
# as --32 5_6.s -g -o 5_6.o
# ld –m elf_i386 5_6.o –o 5_6
# або gcc –nostdlib 5_6.s –o 5_6
.data
printf_format:
        .string "%u\n"

str_in:
        .string "1234567890"

.text
/* size_t my_strlen(const char *s); */
my_strlen:
        pushl %ebp
        movl  %esp, %ebp

        pushl %edi

        movl  8(%ebp), %edi             /* ланцюжок */

        movl  $0xffffffff, %ecx
        xorl  %eax, %eax                /* %eax = 0 */

        repne scasb

        notl  %ecx
        decl  %ecx

        movl  %ecx, %eax

        popl  %edi

        movl  %ebp, %esp
        popl  %ebp
        ret

.globl _start
```

```
_start:
        nop
        pushl %ebp
        movl  %esp, %ebp

        pushl $str_in
        call  my_strlen

        movl  %eax,%ecx
        movl  $1, %eax
        movl  %ecx, %ebx
        int $0x80
        nop
```

```
>./5_6
>echo $?
10
```

Реалізація інших стандартних ланцюжкових функцій є у сирцевих кодах ядра Linux в файлах `/usr/src/linux/arch/x86/include/asm/string_*.h`, `/usr/src/linux/arch/x86/lib/{mem*,str*}`.

Лабораторна робота № 6.

6. Директиви препроцесора

Мета роботи: вивчення директив препроцесора

1. Короткі теоретичні відомості

1.1. Директиви препроцесора NASM

Асемблер NASM має потужний препроцесор, який підтримує асемблювання за умовою, одно- і багаторядкові макровизначення, механізм “контекстного стеку”. Директиви препроцесора починаються із символу ‘%’.

Для перенесення рядків препроцесор використовується символ ‘\’.

```
%define THIS_VERY_LONG_MACRO_NAME_IS_DEFINED_TO \
        THIS_VALUE
```

1.2. Однорядкові макроси

1.2.1. Директиви `%define, %xdefine, %assign`

Однорядкові макроси (macros, макроси) визначаються за допомогою директиви **`%define`.** Вона використовує символ ‘**%**’, подібно як мова С використовує символ ‘#’.

Директива `%define` замінює значення змінних або розширює вирази:

```
%define size 100
%define ctrl 0x1F &
%define param(a,b) ((a)+(a)*(b))
…
mov byte [param(2,ebx)], ctrl 'D'
```

буде розширено до

```
mov byte [(2)+(2)*(ebx)], 0x1F & 'D'
```

Коли однорядковий макрос використовується в іншому однорядковому макросі, то таке розширення відбувається під час *виконання* (*підставлення*), а не визначення:

```
%define a(x) 1+b(x)
%define b(x) 2*x
mov ax,a(8)         ; макророзширення під час виконання 1+2*8
```

Макроси визначені **`%define`** є регістро залежними, а **`%idefine`** – регістро незалежними

```
%define size 100     ;  стосується тільки size
%idefine Size 100    ;  стосується size, SIZE, Size
```

Для запобігання циклічних розширень в макросах використовується механізм розширення тільки для першого появлення макросу:

```
%define a(x) 1+a(x)
mov ax,a(3)     ; 1+a(3)
```

Макроси можна перевантажувати:

```
%define fun(x) 1+x
%define fun(x,y) 1+x*y
```

Макроси без параметрів можна перевизначати в одному файлі:

```
%define fun bar
…
%define fun baz
```

Директива **%undef** відміняє попереднє визначення однорядкового макроса, наприклад:

```
%define foo bar
%undef foo
mov eax, foo
```

Для того, щоб макроси розширювалися під час *визначення* (*негайно*), а не під час виконання, використовується директива **%xdefine.**

```
%define isTrue 1
%define isFalse isTrue
%define isTrue 0
  val1: db isFalse
%define isTrue 1
  val2: db isFalse
```

Позначка `val1` має значення 0, а `val2` – 1, це тому що однорядковий макрос розширюється тільки при його виклику. Так як `isFalse` розширюється до `isTrue`, то розширення матиме значення `isTrue`. У позначці `val1` розширення матиме значення 0, а у позначці `val2` – матиме 1.

Для того, щоб `isFalse` розширювався до значення назначеному вбудованому макросу `isTrue` у *момент визначення* `isFalse`, необхідно використати директиву **%xdefine.**

```
%xdefine isTrue 1
%xdefine isFalse isTrue
%xdefine isTrue 0
  val1: db isFalse
%xdefine isTrue 1
  val2: db isFalse
```

Тепер, при кожному виклику `isFalse` воно буде розширюватися до 1.

1.2.2. Макрозмінна змінна %assign (%iassign)

%assign (**%iassign** – регістро нечутлива версія) використовується для визначення однорядкових макросів, які не мають параметрів, а мають числове значення. Значення можна задати як вираз, який оцінюється один раз при обробці препроцесорної змінної:

```
%assign i 25
%assign i i+1
```

З макрозмінною `%assign` часто використовуються директива макроповторень **%rep** … **%endrep.** В тілі макросу може застосувати директиву **%exitrep,** яка за умовою припиняє виконання макросу, наприклад:

```
%assign i 0
%rep 10
   %if i > 10
       %exitrep
   %endif
   %assign j i+1
   %assign i j
%end rep
```

1.2.3. Директива зчеплення виразів макросів %+

Окремі вирази однорядкових макросів можуть бути зчеплені з використанням виразу `'%+'` (після якого потрібно вставити символ пропуску, щоб відрізнити від синтаксису багаторядкових макросів виду `%+1`). Розглянемо фрагмент коду

```
%define BDASTART 400h  ; початок області даних BIOS
struc tBIOSDA          ; структура
.COM1addr RESW 1
```

```
.COM2addr RESW 1
; ..
endstruc
```

Доступ до різних елементів структури даних дають інструкції

```
mov ax,BDASTART + tBIOSDA.COM1addr
mov bx,BDASTART + tBIOSDA.COM2addr
```

Ці інструкції можна записати як макрос

```
; Макрос для доступу до BIOS змінних за їх іменами (відносно tBDA):
%define BDA(x) BDASTART + tBIOSDA. %+ x
```

Доступ до різних елементів структури з використанням макросу

```
mov ax,BDA(COM1addr)
mov bx,BDA(COM2addr)
```

1.2.4. Спеціальні символи %?, %??

Спеціальні символи **%?, %??** можуть використовуватися для посилання на свої імена макросів всередині макророзширень. Директива **%?** посилається на ім'я макросу *при виклику*, а **%??** – *при оголошенні* макросу, наприклад:

```
%idefine Foo mov %?,%??
foo
FOO
```

буде розширена до:

```
mov foo,Foo
mov FOO,Foo
```

1.2.5. Директива %strcat

Директива **%strcat** об'єднує символьні рядки і назначає їх однорядковому макросі:

```
%strcat alpha "Alpha: ", '12" screen'
```

1.2.6. Директива %strlen

Директива **%strlen** назначає довжину стрічки макросу

```
%strlen charcnt 'my string'
```

В результаті `charcnt` буде присвоєно значення 9.

1.2.7. Директива видобування підстрічок %substr

Директива **%substr** видобуває рядки з підрядків:

```
%substr mychar 'xyzw' 1    ; аналогічно до %define mychar 'x'
%substr mychar 'xyzw' 2    ; аналогічно до %define mychar 'y'
%substr mychar 'xyzw' 3    ; аналогічно до %define mychar 'z'
%substr mychar 'xyzw' 2,2  ; аналогічно до %define mychar 'yz'
%substr mychar 'xyzw' 2,-1 ; аналогічно до %define mychar 'yzw'
%substr mychar 'xyzw' 2,-2 ; аналогічно до %define mychar 'yz'
```

1.3. Багаторядкові макроси

В багаторядкових макросах макровизначення поміщається між директивами **%macro** і **%endmacro.** Після директиви **%macro** задається ім'я макросу і кількість параметрів, наприклад:

```
%macro prolog 1
```

```
    push ebp
    mov ebp,esp
    sub esp,%1
%endmacro
```

де `prolog` – ім'я багаторядкового макросу, а `%1` – число параметрів, яке отримує макрос.

Виклик макросу

```
myfunc: prolog 12
```

буде розширений до наступних інструкцій:

```
myfunc: push ebp
mov ebp,esp
sub esp,12
```

Макрос з двома параметрами:

```
%macro silly 2
   %2: db %1
%endmacro
```

буде розширений до наступних інструкцій (параметр з комою береться у фігурні дужки):

```
silly 'a',  letter_a   ;  =>  letter_a:  db 'a'
silly 'ab', string_ab  ;  =>  string_ab: db 'ab'
silly {13,10}, crlf    ;  =>  crlf:      db 13,10
```

1.3.1. Перевантаження макросів

Багаторядкові макроси можна перевантажити використовуючи одне і те ж ім'я, але різну кількість параметрів. Макрос без параметрів:

```
%macro push 0
   push ebp
   mov ebp,esp
%endmacro
```

Макрос з двома параметрами:

```
%macro push 2
   push %1
   push %2
%endmacro

push ebx     ; це не виклик макросу
push eax,ecx ; це виклик макросу
```

1.3.2. Локальні позначки у макросах

Макроси можуть містити локальні позначки, перед якими ставиться знак `%%`:

```
%macro retz 0
   jnz %%skip
   ret
%%skip:
%endmacro
```

1.3.3. Жадібні макропараметри

Іноді потрібно визначити макрос, який виділяє декілька параметрів, а решту параметрів об'єднує з останнім разом із комами. Для цьому після числа, яке задає кількість параметрів ставиться символ '**+**'. Нехай є макрос:

```
%macro writefile 2+
```

```
   jmp %%endstr
%%str: db %2
%%endstr:
   mov dx,%%str
   mov cx,%%endstr-%%str
   mov bx,%1
   mov ah,0x40
   int 0x21
%endmacro
```

При виклику, макрос з параметрами `writefile [filehandle],"hello, world",13,10` приймає як перший параметр `[filehandle]`, а другий доповнює рештою параметрів – `"hello, world",13,10` (розміщується після `db`).

1.3.4. Змінне число і діапазон параметрів

Для багаторядкових макросів можна задати змінне число параметрів через тире. Наприклад, макрос з числом параметрів від 0 до 1:

```
%macro DIE 0-1 "Аварійне завершення програми"
   writefile 2,%1
   mov ax,0x4c01
   int 0x21
%endmacro
```

Так без параметрів, видається повідомлення `"Аварійне завершення програми"`, а з параметром виводиться текст заданого повідомлення у `STDERR`.

Якщо верхнє обмеження на число параметрів відсутнє, то задається символ *****:

```
%macro die 0-*
```

Для розширення параметрів у заданому діапазоні використовується спеціальна конструкція **`%{start:end}`**, наприклад виклик:

```
%macro mpar 1-*
db %{3:5}
%endmacro
mpar 1,2,3,4,5,6
```

розшириться до діапазону 3,4,5, а виклик

```
%macro mpar 1-*
db %{5:3}
%endmacro
mpar 1,2,3,4,5,6
```

розшириться до діапазону 5,4,3.

1.3.5. Лічильник параметрів макросу %0

Параметр макросу **`%0`** повертає числову константу з числом отриманих параметрів, тобто якщо `%0` дорівнює `n`, то `%n` є останній параметр.

1.3.6. Прокручування параметрів макросу

Директива **`%rotate`** `1` прокручує параметри по колу справа наліво на 1 позицію (`%rotate -1` – зліва на право).

```
%macro multipush 1-*    ; відсутнє обмеження на верхнє число параметрів
%rep %0
   push %1
%rotate 1
```

```
%endrep
%macro
```

1.3.7. Відміна макросів

Для відміни багаторядкових макросів використовується директива `%unmacro:`

```
%macro foo 1-3
   ; Do something
%endmacro
%unmacro foo 1-3
```

1.3.8. Коди умов як макропараметри

Для використання кодів умов як макропараметрів використовується синтаксис **%+1**, **%-1.** Синтаксис %+1 вказує, що він містить код умови і буде впливати на препроцесор так, що він буде видавати повідомлення про помилки, якщо макрос викликається з параметром, який не є дійсним кодом умови (%-1 – параметр, який є інверсним кодом умови).

```
%macro retc 1
   j%-1 %%skip
   ret
%%skip:
%endmacro
```

Цей макрос може бути викликаний як `retc ne` і в результаті спричинить інструкцію умовного переходу `je` у макророзширенні.

1.3.9. Зчеплення макросів для утворення таблиць кодів

Макроси можна зчеплювати і створювати, наприклад, таблиці кодів:

```
%macro keytab_entry 2
keypos%1 equ $-keytab
db %2
%endmacro

keytab:
keytab_entry F1,128+1
keytab_entry F2,128+2
keytab_entry Return,13
```

макрос розшириться до:

```
keytab:
keyposF1 equ $-keytab
db 128+1
keyposF2 equ $-keytab
db 128+2
keyposReturn equ $-keytab
db 13
```

1.4. Директиви асемблювання з умовами

Для *тестування однорядкових макросів* у `Nasm` використовується директива `%ifdef:`

```
%ifdef DEBUG
   writefile 2,"Function performed successfully",13,10
%endif
```

Для *тестування багаторядкових макросів* використовується директива `%ifmacro:`

```
%ifmacro MyMacro 1-3
    %error "MyMacro 1-3" помилка виклику макросу
%else
   %macro MyMacro 1-3
      ; код макросу
   %endmacro
%endif
```

Для *тестування стеку контекстів* використовується директива **%ifcntx** (**%ifnctx, %elifctx, %elifnctx**). Директива асемблює код, якщо вміст верхівки препроцесорного стеку контекстів має таке ж ім'я, як один із аргументів.

Для тестування довільних числових виразів використовується директиви **%if, %elif, %ifn, %elifn.** Вираз **%if** розширює синтаксиси Nasm, дозволяючи набір наступних умов порівняння =(==),<, >, <=, >=,<>(!=),&&(and),||(or),^^(xor).

Для *тестування ідентичності виразів* text1 і text2, які отримуються після розширення однорядкових макросів, використовується директива **%ifidn** text1,text2 (**%ifidni** – регістро нечутлива).

```
%macro pushparam 1
   %ifidni %1,ip
      call %%label
   %%label:
   %else
      push %1
   %endif
%endmacro
```

Для *тестування чи у макрос, як перший параметр, передається ідентифікатор, число або стрічка* використовуються директиви **%ifid, %ifnum, %ifstr.**

```
%macro writefile 2-3+
   %ifstr %2
      jmp %%endstr
   %if %0 = 3
      %%str: db %2,%3
   %else
      %%str: db %2
   %endif
      %%endstr: mov dx,%%str
                mov cx,%%endstr-%%str
   %else
                mov dx,%2
                mov cx,%3
   %endif
                mov bx,%1
                mov ah,0x40
                int 0x21
%endmacro
writefile [file], strpointer, length
writefile [file], "hello", 13, 10
```

Для *тестування кількості параметрів*, які передаються в макрос використовуються директиви:

%ifempy – немає параметрів

%iftoken 1 – один параметр

%iftoken -1 – більше одного параметра

1.5. Директива асемблювання фрагментів макросу

Директиви **%rep** і **%endrep** використовуються для дублювання фрагментів коду макросу:

```
%assign i 0
%rep 64
   inc word [table+2*i]
%assign i i+1
%endrep
```

1.6. Директиви включення файлів

Директива **%include** file вставляє вміст файлу file перед початковим файлом:

```
%include "macros.inc"
```

Для запобігання багатократного включення файлу file використовується макрос:

```
%ifndef MACROS_MAC
%define MACROS_MAC
   ; now define some macros
%endif
```

Директива **%use** name вставляє іменовані стандартні макропакети (які захищені від багаторазового включення) у початковий файл.

```
%use altreg
%use 'altreg'
```

1.7. Стек контекстів

Nasm підтримує стек контекстів, кожен з яких задається іменем. Можна додати новий контекст у стек або вилучити директивами **%push** і **%pop**. Директива **%push** створює новий контекст і поміщає на його верхівку ім'я контексту:

```
%push foobar
```

Директива **%pop** вилучає верхній контекст стеку і руйнує його разом з усіма асоційованими позначками.

Контекстно локальні позначки у певному макровизначенні задаються як **%%label.** Подібно визначаються позначки директивою **%$label**, які є локальними до контексту на верхівці контекстного стеку.

Приклади макросів з контекстним стеком:

```
%macro repeat 0
   %push repeat
   %$begin:
%endmacro

%macro until 1
   j%-1 %$begin
   %pop
%endmacro
```

Виклик макросів:

```
mov cx,string
repeat
add cx,3
scasb
until e
```

які сканують кожний четвертий байт стрічки у пошуку байту в AL.

Якщо потрібно визначення (або доступ до) локальних позначок контексту нижче від першої верхівки стеку можна використати **%$$label**, або **%$$$label** і так далі.

1.8. Стандартні макроси

Макроси **struc** і **endstruc** використовуються для визначення даних типу структура. Макрос struc може мати один або два параметри. Перший параметр задає ім'я типу даних, а другий – зміщення структури від бази. Всередині структури необхідно визначити поля з використанням псевдоінструкцій resb.

```
struc mytype
   mt_long: resd 1
   mt_word: resw 1
   mt_byte: resb 1
   mt_str: resb 32
endstruc
```

У структурі визначено наступні символи, які мають зміщення: mytype, mt_long – 0, mt_word – 4, mt_byte – 6, mt_str – 7 і mytype_size – 39.

Якщо імена полів структури співпадають з іменами у інших структурах, то можна визначити структуру наступним чином:

```
struc mytype
   .long: resd 1
   .word: resw 1
   .byte: resb 1
   .str: resb 32
endstruc
```

Це визначає зміщення полів структури як mytype.long, mytype.word, mytype.byte, mytype.str.

Іноді відоме тільки зміщення структури, наприклад у стандартному стековому фреймі:

```
push ebp
mov ebp,esp
sub esp,40
```

У цьому випадку можна доступитися до елементу структури віднімаючи зміщення:

```
mov [ebp - 40 + mytype.word],ax
```

Щоб не вказувати зміщення в команді, його можна задати у визначенні структури:

```
struc mytype, -40
```

Тоді доступ до елементів структури буде наступним:

```
mov [ebp + mytype.word],ax
```

Маючи визначений тип структури можна оголосити екземпляр структури у сегменті даних

```
mystruc:
   istruc mytype
      at mt_long, dd 123456
      at mt_word, dw 1024
      at mt_byte, db 'x'
      at mt_str, db 'hello, world', 13, 10, 0
   iend
```

Макрос align або **alignb** вирівнює код або дані на word, doubleword, longword, paragraph або інші границі.

```
align 4        ; вирівняти на 4-байтну границю
align 16       ; вирівняти на 16-байтну границю
align 8,db 0   ; вирівняти і заповнити нулями а не NOPs
align 4,resb 1 ; вирівняти на 4 у секції BSS
alignb 4       ; вирівняти на 4 у секції BSS
```

1.9. Стандартні макропакети

altreg – забезпечує альтернативні імена регістрів

smartalign – забезпечує вирівнювання з врахуванням типу платформи

fp – макроси з плаваючою крапкою:

```
%define Inf __Infinity__
%define NaN __QNaN__
%define QNaN __QNaN__
%define SNaN __SNaN__
%define float8(x) __float8__(x)
%define float16(x) __float16__(x)
%define float32(x) __float32__(x)
%define float64(x) __float64__(x)
%define float80m(x) __float80m__(x)
%define float80e(x) __float80e__(x)
%define float128l(x) __float128l__(x)
%define float128h(x) __float128h__(x)
```

Запитання.

1. Що таке препроцесор асемблера і для чого він призначений.
2. Однорядкові макровизначення **%define, %assign** і їх вкладені розширення.
3. Директиви **%+, %?, %??, %undef, %strcat, %strlen, %substr.**
3. Багаторядкові макровизначення.
4. Параметри макросів – жадібні, змінне число і діапазон, лічильник.
5. Асемблювання з умовами.
6. Директиви препроцесора для роботи з початковим файлом.
7. Для чого призначений стек контекстів і директиви для роботи з ним.
8. Стандартні макроси.
9. Стандартні макропакети.

Завдання.

1. Написати макрос для виділення із стрічки, підстрічки із заданої позиції і заданої довжини.

2. Написати макрос для множення двох цілих чисел і виведення результату на екран.

3. Написати макрос для ділення двох цілих чисел і виведення результату на екран.

4. Написати програму, яка виводить повідомлення “Привіт світ”, з використанням макросимволу, який включається ключем командного рядка:

```
nasm -f elf -dDEBUG_PRINT prog.asm
```

5. Написати макрос для копіювання з однієї області пам’яті (`.data`), заданої адресою позначки у область пам’яті (`.bss`). Довжина області пам’яті довільна. Признак кінця даних 0.

2. Приклади для самостійної роботи

1. Макродирективa визначення довжини стрічки:

```
%strlen sl 'my string'
```

Макрозмінна `s1` отримає значення 9.

2. Макродирективa виділення символів із стрічки:

```
%substr var1 'abcd' 1
%substr var2 'abcd' 2
%substr var3 'abcd' 3
```

Макрозмінні `var1, var2, var3` отримають значення `'a','b','c'`.

3. Оголошення і ініціалізація області пам’яті із 100 комірок:

```
%assign n 50
%rep 100
db n
%assign n n+1
```

```
%endrep
```

4. Макрос для генерації послідовності з 128 команд inc, які збільшують на 1 значення кожного елементу масиву.

```
%assign a array
%rep 128
inc word [a]
%assign a a+2
%endrep
```

5. Генерація 100 директив dd з ініціалізацією комірок числами Фібоначчі (1, 1, 2, 3, 5,...):

```
fibonacci
%assign i 1
%assign j 1
%rep 1000
%if j > 100
%exitrep
%endif
dd j
%assign k j+i
%assign i j
%assign j k
%endrep
fib_count equ ($-fibonacci)/4
```

6. Макрос для очистки області пам'яті заданої позначкою і довжиною

```
%macro ZEROMEM 2 ; (два параметри – адреса і довжина)
push ecx
push esi
push dword %2
push dword %1
pop esi
28
pop ecx
%%lp: mov byte [esi], 0
inc esi
loop %%lp
pop esi
pop ecx
%endmacro
```

7. Макрос для виклику процедури з двома і трьома параметрами

```
%macro PCALL2 3
push %3
push %2
call %1
15
add esp, 8
%endmacro

%macro PCALL3 4
push %4
push %3
push %2
call %1
add esp, 12
%endmacro
```

8. Макрос для виклику процедури із довільним числом параметрів (параметри мають розмір 4 байти і у стек поміщаються у зворотному порядку).

```
%macro PCALL 1-*
%rep %0 - 1 ; цикл по всіх параметрах крім першого
```

```
%rotate -1 ; останній параметр стає %1
push dword %1
%endrep
%rotate -1 ; адреса процедури стає %1
call %1
add esp, (%0 - 1) * 4
%endmacro
```

9. hello_1.asm – програма виводить повідомлення hello worl з використанням макросів

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro PRINT_STR 1
;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; %1 - message
pusha
pushf
jmp %%astr
%%str db %1,10     ; 10 - new line
%%len equ $-%%str
%%astr:
mov eax,4          ; 4=SYS_write
mov ebx,1          ; 1=STDOUT
mov ecx,%%str
mov edx,%%len      ; msg length
int 0x80
popf
popa
%endmacro
```

; Макрос завершення програми з передачею значення ebx в ядро ОС.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;
%macro FINISH 1
;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; %1 - exit code
mov eax, 1   ; 1=SYS_exit
mov ebx,%1   ; exit code
int 0x80
%endmacro

global _start
segment .text
_start:
   PRINT_STR "Hello world"
   FINISH 0
>./hello_1
```

>hello world

Лабораторна робота № 7.

7. Виклики підпрограм

Мета роботи: вивчення механізмів виклику підпрограм в асемблері ОС Linux

1. Основні теоретичні відомості

1.1. Поняття стеку і його призначення

Під стеком в програмуванні розуміють структуру побудовану за принципом ”перший прийшов – останній вийшов” на якій визначені операції ”добавити елемент”, ”вилучити елемент”.

При програмуванні на асемблері під стеком розуміють неперервну область пам’яті для якої зберігається адреса верхівки стеку. Стек розміщується у верхній частині оперативної пам’яті (найбільший адрес), тому дно стеку має найбільшу адресу. Добавлення елементу у стек зменшує значення адреси верхівки стеку, а вилучення елементу зі стеку збільшує значення адреси верхівки стеку.

Стек використовується для тимчасового зберігання значень регістрів у випадках коли не вистачає регістрів загального призначення.

Більш важливим є використання стеку при викликах підпрограм для зберігання адрес повернення, для передачі фактичних аргументів і для зберігання локальних змінних. Саме використання стеку дозволяє реалізувати механізм рекурсії, при якому підпрограма може явно або неявно викликати сама себе.

1.2. Організація стеку в процесорі i386

Для адресації верхівки стеку використовується регістр загального призначення `ESP` (stack pointer) – вказівник стеку.

Команда `push operand` заносить вміст операнду (регістру або комірки пам’яті) в стек, при цьому вказівник стеку `esp` зменшує своє значення на розмір операнду (`word, dword`). Якщо операнд не регістровий то його розмір необхідно вказати явно.

Команда `pop operand` видобуває значення з верхівки стеку і заносить в операнд (регістр або комірку пам’яті), при цьому вказівник стеку `esp` збільшує своє значення на розмір операнду (`word, dword`). Команди `push, pop` не підтримують операнди розміром байт

Якщо потрібно тільки звернутися до значення на верхівці стеку (без видобування його зі стеку) можна використати операнд `[esp]`:

```
mov eax,esp     - читання адреси верхівки стеку
mov eax,[esp]   - читання значення на верхівці стеку
mov eax,[esp+4] - читання наступного значення нижче верхівки стеку
```

Стек використовується для розміщення аргументів виклику, значень повернення з функцій і локальних змінних функції.

Приклад використання стеку для тимчасового зберігання значень регістра `BX`.

Нехай регістр `ESI` містить адресу деякої стрічки символів в пам’яті, причому відома, що вона закінчується байтом із значенням 0. Необхідно переписати елементи стрічки у зворотному порядку використовуючи стек.

```
; 5_1.asm – перепис елементів стрічки у зворотному порядку
; отримання бінарного файлу: ./asm.sh 5_1
%include ”student.inc”
segment .data
str:  db  ’1234567890’,10
len equ $-str
```

```
segment .text
global _start
_start:
    nop
    PRINT_STR str,len
    xor ebx, ebx   ; обнуління  ebx
    xor ecx, ecx   ; обнуління ecx
    mov esi,str
lp: mov bl, [esi+ecx] ; черговий байт із стрічки
    cmp bl, 0      ; кінець рядка?
    je lpquit      ; якщо так – кінець циклу
    push bx        ; bl не підтримується, потрібно bx
    inc ecx        ; наступний індекс
    jmp lp         ; повторити цикл
lpquit: jecxz done ; якщо стрічка порожня, вихід з циклу
    mov edi,esi    ; копіювати адресу з esi в edi
lp2: pop bx        ; видобуваємо із стек в bx
    mov [edi],bl   ; копіюємо у комірку з адресою edi
    inc edi        ; наступна адреса
    loop lp2       ; цикл ecx разів
done:
    PRINT_STR str,len
    FINISH 0
```

```
./5_1
1234567890
0123456789
```

1.3. Підпрограми

Підпрограмою називається деяка окрема частина програми, яка може бути викликана з головної програми або іншої підпрограми. Під викликом у даному випадку розуміють тимчасову передачу керування підпрограмі. При виклику необхідно запам'ятати *адресу повернення*, тобто адресу початку машинної команди, наступної за командою виклику підпрограми.

Підпрограми можуть отримувати параметри і використовувати в роботі локальні змінні. Для цього підпрограми використовують стек. Перед викликом підпрограми у стек записують значення параметрів виклику, потім здійснюють сам виклик підпрограми, який суміщений із зберіганням у стеку адреси повернення. В кінці, коли підпрограма отримає управління, воно резервує певну кількість пам'яті для зберігання змінних, просто зменшуючи адресу верхівки стеку. Область стекової пам'яті, яка містить зв'язані з одним викликом значення параметрів, адреси повернення і локальні змінні, називається *стековим фреймом*.

1.3.1. Виклик підпрограми і повернення з неї

В асемблері є дві команди для організації роботи підпрограм:

`call позначка` – використовується для виклику підпрограми, код якої знаходиться за адресом позначки. Принцип роботи команди `call`:

- помістити в стек, адресу команди наступної після `call`;
- передати керування на позначку.

Аргумент команди `call` (як і команди `jmp`) може бути безпосередній, регістровий і типу "пам'ять".

Для повернення з підпрограми використовується команда `ret`:

```
ret
ret число
```

У своїй найпростішій формі вона не має аргументів. Виконуючи цю команду, процесор видобуває 4 байти з верхівки стеку і записує їх в регістр `EIP`, в результаті чого керування передається за адресою, яка знаходилася в пам'яті на верхівці стеку.

Існують наступні методи передачі аргументів в підпрограму:

- через регістри;
- через загальну пам'ять;
- через стек.

Приклад виклику підпрограми. Нехай в підпрограмі потрібно заповнювати якимось однобайтним значенням область пам'яті різної довжини. Адреса потрібної області пам'яті передається через регістр `EDI`, кількість однобайтних комірок – через регістр `ECX`, а саме однобайтне значення – через регістр `AL`. Код підпрограми:

```
; fill memory (edi=address, ecx=length, al=value)
fill_memory:
    jecxz lq
lp: mov [edi],al
    inc edi
    loop lp
lq: ret
```

Звернення до підпрограми:

```
; 5_2.asm – звернення до підпрограми
; отримання бінарного файлу: ./asm.sh 5_2
segment .bss
len equ 8
my_array resb len
segment .text
global _start
_start:
    nop
    mov edi,my_array
    mov ecx,len
    mov al,'#'
    call fill_memory
    PRINT_STR my_array,len
    FINISH 0
```

```
./5_1
########
```

1.4. Організація стекового фрейму

У прикладі виклику підпрограми фактично не використовувався механізм стекових фреймів, у стеку зберігалася лише адреса повернення, а параметри передавалися через регістри і в підпрограмі не використовувалися локальні змінні.

На практиці підпрограми рідко бувають такими простими. У більш складних випадках може використовуватися велике число локальних змінних та параметрів, на які не вистачить регістрів. Крім того передача параметрів через загальну область пам'яті або регістри унеможливлює використання рекурсії.

Тому звичайно у складних програмах (особливо при трансляції з мов програмування високого рівня) параметри передаються через стек і в стеку розміщуються локальні змінні.

Параметри (які розміщує в стеку викликаючи програма), адреса повернення (яку розміщує команда `call`) і локальні змінні (які розміщує підпрограма) утворюють *стековий фрейм*. Як реперну точку, при зверненні до елементів стекового фрейму, можна використати адресу повернення. Якщо в стек занести три 4-х байтні параметри, а потім викликати підпрограму, то адреса повернення буде в `[esp]`, а адреси параметрів будуть `[esp+4]`, `[esp+8]`, `[esp+12]`. Якщо в підпрограмі використовуються дві 4-х розрядні локальні змінні, то їх адреси будуть `[esp-4]`, `[esp-8]`.

Так як регістр вказівник верхівки стеку може використовуватися у підпрограмі (для виклику інших підпрограм), то його значення зберігають в іншому регістрі, переважно в `ebp` (регістрі вказівника бази). Тоді регістр `ebp` буде містити адресу повернення, а регістр `esp` буде використовуватися як вказівник верхівки стеку.

В програмі можуть викликатися декілька підпрограм, які також використовують регістр `ebp`. Для збереження значення `ebp` кожної підпрограми і, враховуючи те, що в програмі є значно більше викликів підпрограм, ніж самих підпрограм, прийняте просте правило: *кожна підпрограма сама зберігає старе значення ebp і відновлює його перед поверненням управління.*

Для зберігання `ebp` також використовується стек, причому збереження здійснюється командою `push ebp` зразу після отримання управління підпрограмою. Таким чином, старе значення `ebp` розміщується в стеку після адреси повернення з підпрограми. Саме це старе значення `ebp` використовується як реперна точка для адресації стекового фрейму:

```
[ebp+16]
[ebp+12]    параметри
[ebp+8]
[ebp+4]     адреса повернення
[ebp]       збережене старе значення ebp
[ebp-4]
[ebp-8]     локальні змінні
```

Кожна підпрограма повинна виконати наступні команди перед початком роботи:

```
push ebp
mov ebp,esp
sub esp,8 ; замість 8 резервується потрібний обсяг пам'яті  для локальних змінних
```

і перед завершенням роботи:

```
mov esp, ebp
pop ebp
ret
```

Процесор підтримує спеціальні команди для обслуговування стеку. На початку підпрограми замість трьох команд може використовуватися одна команда `enter 16,0`, а замість двох команд перед `ret` – одна команда `leave`.

ОС Linux створює стек автоматично при запуску любої програми і, більш того, під час її виконання при необхідності збільшує розмір доступної для стеку пам'яті.

1.4.1. Послідовність дій при роботі зі стековим фреймом

Перед викликом підпрограми ***головна програма виконує такі дії:***

- записує значення всіх параметрів виклику функції в стек у зворотному порядку (справа наліво);

- викликає команду `call`.

Команда ***`call` виконує наступні дії:***

- записує у стек адресу наступної команди після `call` (адрес повернення);

- модифікує регістр команд `eip` так, щоб він містив адресу підпрограми виклику.

До виконання підпрограми стек матиме наступний вигляд:

```
   Аргумент N
  ...
  Аргумент 2
  Аргумент 1
  Адреса повернення   <- [esp] верхівка стеку
```

Викликувана ***підпрограма на початку виконання здійснює***:

- записує в стек поточне значення вказівника бази стеку `ebp` командою `push ebp`;

- копіює вказівник верхівки стеку `esp` у вказівник бази `ebp` командою `mov ebp,esp`. Це дозволяє отримати доступ як до параметрів підпрограми, так і локальних змінних шляхом

індексування вказівника бази. Вказівник бази `ebp` завжди вказуватиме на значення вказівника стеку `esp` на початку виконання підпрограми. Вказівник бази стеку є константою і дозволяє звертатися до всіх значень стекового фрейму (параметри, адрес повернення, `ebp`, локальні змінні).

- резервує місце під локальні змінні. Наприклад, під два слова `sub esp,8` (зменшує адресу бази стеку на 8 байтів).

Після цього стек матиме наступний вигляд:

```
Аргумент N                 <-  [ebp+(N+1)*4]
...                        ...
Аргумент 2                 <-  [ebp+34]
Аргумент 1                 <-  [ebp+2*4]
Адреса повернення (esp)    <-  [ebp+4]
Старе (ebp)                <-  [ebp]
Локальна змінна1            <- [esp-4]
Локальна змінна2            <- [esp-8]
```

Викликувана ***підпрограма в кінці виконання здійснює***:

- записує значення, яке повертає підпрограма у регістр `eax`;
- повертає стек в стан коли викликалася підпрограма (відновлює `esp`, `ebp`);
- повертає керування в точку виклику командою `ret`, яка знімає значення з верхівки стеку (адресу повернення) і записує його в регістр вказівник команд `eip`.

Ці кроки підпрограми виконують наступні команди:

```
mov esp,ebp
pop ebp
ret
```

Якщо потрібно перед викликом підпрограми зберегти регістри загального призначення то їх поміщають на зберігання в стек перед параметрами функції командою `pusha`, а потім відновлюють після завершення підпрограми командою `popa`.

1.5. Виклик підпрограм на мові С

При трансляції викликів функції мови С параметри поміщаються у стек у порядку справа наліво і розміщуються у фреймі стеку у порядку знизу вверх. Таким чином, перший параметр завжди буде доступним за адресою [ebp+8], незалежно від загальної кількості параметрів. Тому у підпрограму можна передавати змінне число параметрів. Так як підпрограма не знає скільки їй передали параметрів, то очистка стеку виконує викликаючи програма прости збільшенням значення `esp` на число, яке дорівнює сукупній довжині фактичних параметрів. Наприклад, якщо підпрограма `proc1` приймає три 4-х байтні параметри, то її виклик виглядатиме так:

```
push dword a3 ; занесення в стек параметрів
push dword a2
push dword a1
call proc1    ; виклик підпрограми
add esp, 12  ; вилучення параметрів зі стеку
```

1.6. Передача командного рядка програмі

При запуску програм ОС виділяє в адресному просторі програми спеціальну область для командного рядка. Адреса цього рядка і кількість елементів поміщається у стек запускаємої задачі, після чого керування передається програмі. Таким чином, у момент, коли починає виконуватися програма з позначки `_start`, на верхівці стеку [esp] розміщується число елементів командного рядка (включно з іменем програми), в [esp+4] – адреса де розміщується ім’я, за яким викликається програма, [esp+8] – адреса першого параметру, а далі адреси наступних параметрів

Запитання.

1. Поняття стеку і його призначення.
2. Організація стеку в процесорі i386.
3. Підпрограми
4. Виклик підпрограми і повернення з неї.
5. Організація стекового фрейму.
6. Послідовність дій при роботі зі стековим фреймом.
7. Виклик підпрограм на мові С.
8. Як передається командний рядок програмі.

Завдання.

1. Написати програму, яка викликає дві функції з цілочисельними параметрами *a* і *b*. Одна з яких обчислює *a+b*, а інша *a-b*. Добуток повернених значень повернути з основної програми в ОС і вивести на екран командою `echo $?`.

2. Написати функцію на асемблері, яка викликається із С-програми і визначає максимальне з 5 чисел.

3. Написати рекурсивну програму на асемблері для обчислення факторіалу непарних цілих чисел.

2. Приклади для самостійної роботи

1. 5_4.asm - програма, яка друкує цілі з пам'яті і регістра

```
; Assemble: nasm -f elf -g -F stabs -l 5_4.lst  5_4.asm
; Link:           gcc 5_4.o -o 5_4
; Run:            ./5_4
; Output:   a=5, eax=7

; Еквівалентний С код
; #include <stdio.h>
; int main()
; {
;   int a=5;
;   printf("a=%d, eax=%d\n", a, a+2);
;   return 0;
; }

; Оголошення зовнішніх функцій
      extern      printf      ; С функція, яка викликається
      SECTION .data           ; Data сегмент, ініціалізація змінних
a:    dd    5                 ; int a=5;
fmt:  db "a=%d, eax=%d", 10, 0 ; printf формат, "\n",'0'

      SECTION .text     ; Code секція
      global main       ; Стандартна точку входу в gcc
main:                   ; позначка для точки входу
      push    ebp       ; налаштування фрейму
      mov     ebp,esp

      mov   eax, [a]    ; копіювання з пам'яті у регістр
      add   eax, 2      ; a+2
      push  eax         ; значення регістра eax у стек
      push  dword [a]   ; значення змінної a у стек
      push  dword fmt   ; адреса формату у стек
      call  printf            ; виклик С функції
      add   esp, 12           ; очищення стеку від трьох параметрів 3x4=12 байтів

     mov     esp, ebp   ; звільнення стекового фрейму
     pop     ebp        ; відновлення старого ebp
```

```
     mov eax,0          ;  код повернення для return
     ret                ;  повернення в ОС
```

2. intarith.asm - простий код C і відповідний nasm код

```
; compile:  nasm -f elf -g -F stabs -l intarith.lst  intarith.asm
; link:     gcc -o intarith  intarith.o
; run:      ./intarith
;
; результат виконання intarith.asm і intarith.c is:
; c=5  , a=3, b=4, c=5
; c=a+b, a=3, b=4, c=7
; c=a-b, a=3, b=4, c=-1
; c=a*b, a=3, b=4, c=12
; c=c/a, a=3, b=4, c=4
;
;  /* intarith.c */
;  #include
;  int main()
;  {
;    int a=3, b=4, c;
;
;    c=5;
;    printf("%s, a=%d, b=%d, c=%d\n","c=5  ", a, b, c);
;    c=a+b;
;    printf("%s, a=%d, b=%d, c=%d\n","c=a+b", a, b, c);
;    c=a-b;
;    printf("%s, a=%d, b=%d, c=%d\n","c=a-b", a, b, c);
;    c=a*b;
;    printf("%s, a=%d, b=%d, c=%d\n","c=a*b", a, b, c);
;    c=c/a;
;    printf("%s, a=%d, b=%d, c=%d\n","c=c/a", a, b, c);
;    return 0;
; }

      extern printf     ; the C function to be called

%macro      pabc 1      ; a "simple" print macro
      section .data
.str  db    %1,0        ; %1 is first actual in macro call
      section .text
                        ; помістити у стек у зворотньому порядку
      push  dword [c]   ; int c
      push  dword [b]   ; int b
      push  dword [a]   ; int a
      push  dword .str  ; стрічка користувача
      push  dword fmt   ; адреса стрічки формату
      call  printf      ; виклик C функції
      add   esp,20      ; звільнити в стеку 5*4 bytes
%endmacro

      section .data     ; preset constants, writeable
a:    dd    3           ; 32-біт змінна a ініціалізована 3
b:    dd    4           ; 32-біт змінна b ініціалізована 4
fmt:    db "%s, a=%d, b=%d, c=%d",10,0    ; формат стрічки для printf

      section .bss      ; неініціалізована пам'ять
c:    resd  1           ; зарезервовано 32-біт

      section .text     ; code сегмент
      global       main ; gcc стандартна точка входу для компонування (linking)
main:                   ; позначка
```

```
lit5:                   ; c=5;
      mov   eax,5       ; 5 константа
      mov   [c],eax     ; запис у комірку c
      pabc  "c=5  "     ; виклик print macro

addb:                   ; c=a+b;
      mov   eax,[a]     ; завантажити a у регістр eax
      add   eax,[b]     ; додати b
      mov   [c],eax     ; записати у c
      pabc  "c=a+b"     ; викликати macro

subb:                   ; c=a-b;
      mov   eax,[a]     ; завантажити a
      sub   eax,[b]     ; відняти b
      mov   [c],eax     ; записати у c
      pabc  "c=a-b"     ; викликати print macro

mulb:                   ; c=a*b;
      mov   eax,[a]     ; завантажити a (має бути eax для множення)
      imul  dword [b]   ; знакове цілочисельне множення на b
      mov   [c],eax     ; записати старшу частину добутку у c
      pabc  "c=a*b"     ; викликати print macro

diva:                   ; c=c/a;
      mov   eax,[c]     ; завантажити c
      mov   edx,0       ; завантажити старшу половину діленого нулем
      idiv  dword [a]   ; поділити пару регістрів edx:eax на a
      mov   [c],eax     ; записати результат у c
      pabc  "c=c/a"     ; викликати print macro

      mov eax,0         ; exit код, 0=normal
      ret               ; повернення з main програми в ОС
```

3. subr1.asm - приклад передачі параметрів в підпрограму і повернення результату

```
segment .data
a: dd 5
b: dd 4
segment .text
global _start
_start:
     push dword [a]
     push dword [b]
     call sub1
     add esp,8  ; звільнити стек від параметрів
                ; в ebx результат з підпрограми
     mov eax,1  ; передати результат у Linux
     int 0x80

sub1:
     push ebp
     mov ebp,esp          ; вхід в підпрограму

     sub esp,8            ; місце під локальні змінні
     mov dword [ebp-4],2  ; локальна змінна 2
     mov dword [ebp-8],8  ; локальна змieea 8
     mov eax,[ebp-4]      ; 2->eax
     mul dword [ebp-8]    ; eax = eax * локальну змінну 9

     mov ebx,[ebp+12]  ; [a]
     add ebx,[ebp+8]   ; [a]+[b]
     add ebx,eax       ; [a]+[b] + eax -> 25
     add esp,8         ; звільнити локальні змінні
```

```
     mov esp,ebp       ; вихід з підпрограми
     pop ebp
     ret
./subr1
>echo $?
25
```

4. echo_32.s - програма, яка висвітлює аргументи командного рядка

```
# ----------------------------------------------------------------------
# echo_32.s - 32-bit програма, яка висвітлює аргументи командного рядка
# gcc echo_32.s -o echo_32
# ----------------------------------------------------------------------

        .global main

        .text
main:
        movl     4(%esp), %ecx           # argc
        movl     8(%esp), %edx           # argv
top:
        pushl     %ecx                    # збереження регістрів, які
                                          # використовує printf
        pushl     %edx

        pushl     (%edx)                  # рядок аргументу для виводу на екран
        pushl     $format                 # форма виводу
        call    printf
        addl     $8, %esp                 # вилучення двох параметрів

        popl     %edx                     # відновлення регістрів, які
                                          # використовував  printf
        popl     %ecx

        addl     $4, %edx                 # вказівник на наступний аргумент
        decl     %ecx                     # зменшення лічильника
        jnz     top                      # якщо не нуль, то продовження циклу

        ret
format:
        .asciz  "%s\n"
```

5. Обчислення степенів (32-біт версія)

```
/*  power_32.s - обчислення степені викликом функції power() (32-біт версія) */
#ПРИЗНАЧЕННЯ:
#      Програма обчислює степені викликом функції
#      2^3 + 2^5
#  >as power_64.s -o power_64.o
#  >ld power_64.o -o power_64
#  >./power_64
#   33

#Все в основній програмі зберігається в регістрах, стандартні дії по закінченню
# функції, тому секція data не містить нічого.

.section .data  # секція даних
.section .text  # секція програм
.globl _start
.arch pentium   # архітектура Pentium

_start:
```

```
                #занесення параметрів в стек перед викликом функціїї
pushl $3        #push другий аргумент
pushl $2        #push перший аргумент
                #Виклик функції power
call power      #Команда call записує в стек адрес повернення (адрес команди addl)
addl $8, %esp   #перемістити вказівника стеку
pushl %eax      #зберегти перший результат перед
                #перед наступним викликом функції

pushl $2        #push другий аргумент
pushl $5        #push перший аргумент
call power      #виклик функції. Значення повертається в %eax
addl  $8, %esp  #перемістити вказівник стеку
popl %ebx       #перший аргумент в %ebx, а другий аргумент є в %eax
addl %eax, %ebx #додати аргументи
movl $1, %eax   # вихід, значення повертається в %ebx
int $0x80

# функція power -  обчислює n^m, int n>=1, int m>=1 (32-бітна версія)
# перший аргумент - n, другий - m
# змінні:
#  %ebx містить n, %ecx містить m
#  -4(%ebp) містить поточний результат
#  %eax - тимчасова змінна
.type power, @function # power - функція
power:
                    #стандартні дії функції на початку виконання
pushl %ebp          #функція записує в стек поточне значення %ebp
movl %esp, %ebp     #функція присвоює базі стеку %ebp значення вказівника стеку
%esp
subl $4, %esp       #функція виділяє місце під локальну змінну (слово має 4
байти)

movl 8(%ebp), %ebx  #поміщення першого аргументу (основи)  з фрейму стеку в %ebx
movl 12(%ebp), %ecx #поміщення другого аргументу (степені) з фрейму стеку в %ecx
movl %ebx, -4(%ebp) #збереження поточного результату в локальній змінній

power_loop_start:
cmpl $1, %ecx       #порівняння значення в %ecx з 1
je end_power
movl -4(%ebp), %eax #пересилання поточного результату в %eax
imull  %ebx, %eax   #множення поточного результату на основу
movl %eax, -4(%ebp) #збереження поточного результату
decl  %ecx          #зменшення степені
jmp power_loop_start
end_power:
movl -4(%ebp), %eax # отримання поверненого значення через %eax
                    # стандартні дії по закінченню функції
movl %ebp, %esp     #відновлення збереженого вказівника стеку
popl %ebp           #відновлення вказівника бази
ret
>power_32
>echo $?
33
```

6. Обчислення максимального з трьох чисел викликом asm програми з С-програми

```
/*
 * maxof3.c
 * Асемблювання, компонування і виконання програми з двох частин:
 * gcc maxof3_32.s callmaxof3.c -o maxof3
 * ./maxof3
 *
 * Виклик asm програми maxof3_32.s   для 32-бітних машин
```

```
*/

#include <stdio.h>

int maxof3_32(int, int, int);

int main() {
    printf("%d\n", maxof3_32(1, -4, -7));
    printf("%d\n", maxof3_32(2, -6, 1));
    printf("%d\n", maxof3_32(2, 3, 1));
    printf("%d\n", maxof3_32(-2, 4, 3));
    printf("%d\n", maxof3_32(2, -6, 5));
    printf("%d\n", maxof3_32(2, 4, 6));
    return 0;
}

# ------------------------------------------------------------------------------
# 32-бітна функція повертає максимум з трьох int аргументі.
# Шаблон (сигнатура) функції
#   int maxof3_32(int x, int y, int z)
#
# Примітка. Використовуються тільки eax, ecx і edx регістри, тому немає
необхідності зберігати і відновлювати інші регістри.
# ------------------------------------------------------------------------------
        .global maxof3_32
        .text
maxof3_32:
        mov     4(%esp), %eax
        mov     8(%esp), %ecx
        mov     12(%esp), %edx
        cmp     %ecx, %eax
        cmovl   %ecx, %eax
        cmp     %edx, %eax
        cmovl   %edx, %eax
        ret
```

```
>./maxof3
-4
-6
3
-2
-6
6
```

7. Виклик рекурсивної функції з основної при обчисленні факторіалу

```
/* factorial1.s */
.data
printf_format:
        .string "%d\n"

.text
/* int factorial(int) - рекурсивне обчислення факторіалу*/
/* функція */
factorial:
        pushl %ebp
        movl  %esp, %ebp

        /* видобути аргумент в %eax */
        movl  8(%ebp), %eax

        /* факторіал 0 равен 1 */
        cmpl  $0, %eax
        jne   not_zero
```

```
        movl  $1, %eax
        jmp   return
not_zero:

        /* наступні 4 рядки обчислюють вираз
           %eax = factorial(%eax - 1) */

        decl  %eax
        pushl %eax
        call  factorial
        addl  $4, %esp

        /* видобути в %ebx аргумент і обчислити %eax = %eax * %ebx */

        movl  8(%ebp), %ebx
        mull  %ebx

        /* результат в парі %edx:%eax, але старші 32 біти потрібно
           відкинути, так як вони не поміщаються в int */

return:
        movl  %ebp, %esp
        popl  %ebp
        ret

 /* основна програма */
.globl main
main:
        pushl %ebp
        movl  %esp, %ebp

        pushl $5
        call  factorial

        pushl %eax
        pushl $printf_format
        call  printf

        /* стек можна не вирівнювати, це буде зроблено
           під час виконання епілога */

        movl  $0, %eax                  /* закінчити програму */

        movl  %ebp, %esp
        popl  %ebp
        ret
```

8. Ітераційне обчислення факторіалу

```
/* factorial2.s */
.data
printf_format:
        .string "%d\n"

.text

# ітераційне обчислення факторіалу (32-бітна версія)
/* функція */
factorial:
        movl  4(%esp), %ecx

        cmpl  $0, %ecx
        jne   not_zero
```

```
        movl  $1, %eax
        ret

not_zero:

        movl  $1, %eax
loop_start:
        mull  %ecx
        loop  loop_start

        ret

/* основна програма */
.globl main
main:
        pushl %ebp
        movl  %esp, %ebp

        pushl $5
        call  factorial

        pushl %eax
        pushl $printf_format
        call  printf

        /* стек можна не вирівнювати, це буде зроблено
           під час виконання епілога */

        movl  $0, %eax                  /* закінчити програму */

        movl  %ebp, %esp
        popl  %ebp
        ret
```

9. Рекурсивний виклик функції

```
# 32-біт програма  factorial_32.s обчислює факторіал,
# наприклад
# 4! = 4 * 3 * 2 * 1, або 2.
#
# Асемблювання, компонування і виклик програми
# >as factorial.s -o factorial.o
# >ld factorial.o -o factorial
# >./factorial
# >echo $?
# 24
#
# Програма використовує рекурсивний виклик функції.
.section .data
# Програма не має глобальних даних
.section .text

.globl _start
.globl factorial
_start:
pushl $4         # функція факторіал використовує один аргумент
call factorial   #виклик функції факторіал
addl $4, %esp    #очистити параметр, який передавався в стек
movl %eax, %ebx  #factorial повертає результат в %eax, але
                 #він пересилається в %ebx як exit код
movl #1,%eax     # виклик функції ядра exit
int  $0x80
```

```
# визначення функції factorial
.type factorial,@function
factorial:
pushl %ebp          #стандартна дія - збереження поточного вказівника бази в стек
movl %esp,%ebp      #нове значення вказівника бази
movl 8(%ebp), %eax #переслати перший аргумент в %eax
                    #4(%ebp) містить адрес повернення
                    #8(%ebp) містить перший аргумент
cmpl $1, %eax       #якщо число 1, то це умова виходу
                    #(1 уже є в %eax як return значення)
je end_factorial

decl %eax           #інакше зменшити значення
pushl %eax          #помістити його в стек для виклику факторіалу
call factorial
movl 8(%ebp),%ebx   #%eax містить return значення
                    #яке замінює параметр в %ebx
imull %ebx, %eax    #множення %ebx на результат останнього виклику
                    #factorial (в %eax)
                    #результат записується в %eax

end_factorial:
movl %ebp, %esp     #стандартна дія по відновленню значень
popl %ebp           # %ebp і %esp в стан до початку виклику функції
ret                 #return у функцію (зчитується pop із стеку значення повернення)
```

10. Внутрішній рекурсивний виклик функції

```
#  factorial_32.s обчислює факторіал (32-біт версія)
# наприклад
# 4! = 4 * 3 * 2 * 1, або 2.
#
# Асемблювання, компонування і виклик програми
# >as factorial.s -o factorial.o
# >ld factorial.o -o factorial
# >./factorial
# >echo $?
# 24
#
# Програма використовує рекурсивний виклик функції.
.section .data
# Програма не має глобальних даних
.section .text

.globl _start
.globl factorial
_start:
pushl $4          # функція факторіал використовує один аргумент
call factorial    #виклик функції факторіал
addl $4, %esp     #очистити параметр, який передавався в стек
movl %eax, %ebx   #factorial повертає результат в %eax, але
                  #він пересилається в %ebx як exit код
movl #1,%eax      # виклик функції ядра exit
int  $0x80

# визначення функції factorial
.type factorial,@function
factorial:
pushl %ebp          #стандартна дія - збереження поточного вказівника бази в стек
movl %esp,%ebp      #нове значення вказівника бази
movl 8(%ebp), %eax #переслати перший аргумент в %eax
                    #4(%ebp) містить адрес повернення
                    #8(%ebp) містить перший аргумент
cmpl $1, %eax       #якщо число 1, то це умова виходу
```

```
                    #(1 уже є в %eax як return значення)
je end_factorial

decl %eax           #інакше зменшити значення
pushl %eax          #помістити його в стек для виклику факторіалу
call factorial
movl 8(%ebp),%ebx   #%eax містить return значення
                    #яке замінює параметр в %ebx
imull %ebx, %eax    #множення %ebx на результат останнього виклику
                    #factorial (в %eax)
                    #результат записується в %eax

end_factorial:
movl %ebp, %esp     #стандартна дія по відновленню значень
popl %ebp           # %ebp і %esp в стан до початку виклику функції
ret                 #return у функцію (зчитується pop із стеку значення повернення)
```

Лабораторна робота № 8.

8. Взаємодія з ОС. Переривання

Мета роботи: вивчення взаємодії асемблера з ОС і переривань Linux

1. Короткі теоретичні відомості

1.1. Системні виклики

Програма асемблера взаємодіє з ОС, використовуючи так звані системні виклики, яві в свою чергу реалізовані через механізми програмних переривань.

Апаратні переривання поділяються на:

- зовнішні;
- внутрішні;
- програмні.

Зовнішні переривання поступають від зовнішніх пристроїв, використовуючи одну з доріжок загальної шини. При зовнішньому перериванні процесор переходить у привілейований режим роботи.

Внутрішні переривання виникають при аварійних помилках при виконанні програми (ділення на нуль, порушення захисту пам'яті, виконання неіснуючої команди, спроба читання слова за непарною адресою).

Програмні переривання здійснюються за ініціативою користувача з використанням команди `int (interrupt)`. Програмне переривання також змінює режим виконання з користувацького на привілейований.

Ядро ОС Linux на платформі i386 використовує для системного виклику переривання з номером 80h. Номер системного виклику передається ядру через регістр `eax`. Якщо системний виклик приймає параметри, то вони розміщуються в регістрах `ebx`, `ecx`, `edx`, `esi`, `edi`. Параметри системних викликів є 4-байтними цілими або адресними. Результат виконання виклику повертається через регістр `eax`, причому значення між `ffff_f000h` і `ffff_ffffh` вказують на помилку і задають її код.

В ОС Linux інтерфейс користувача до ядра організований через системні виклики. Звернення задачі користувача до ядра ОС послугами є *системним викликом*. Кожний системний виклик має ім'я і номер. Всі вони перераховані в файлі `/usr/include/asm-86/unistd_32.h (32-розрядна ОС) або /usr/include/asm-86/unistd_64.h (64-розрядна ОС).`

restart_ syscall 0	setregid 71	newselect 142	setuid32 213	keyctl 288
exit 1	sigsuspend 72	flock 143	setgid32 214	ioprio set 289
fork 2	sigpending 73	msync 144	setfsuid32 215	ioprio_get 290
read 3	sethostname 74	readv 145	setfsgid32 216	inotify init 291
write 4	setrlimit 75	writev 146	pivot_root 217	inotify_add_watch 292
open 5	getrlimit 76	getsid 147	mincore 218	inotify rm watch 293
close 6	getrusage 77	fdatasync 148	madvise 219	migrate_pages 294
waitpid 7	gettimeofday 78	sysctl 149	getdents64 220	openat 295
creat 8	settimeofday 79	mlock 150	fcntl64 221	mkdirat 296
link 9	getgroups 80	munlock 151	gettid 224	mknodat 297
unlink 10	setgroups 81	mlockall 152	readahead 225	fchownat 298
execve 11	select 82	munlockall 153	setxattr 226	futimesat 299
chdir 12	symlink 83	sched_setparam 154	lsetxattr 227	fstatat64 300
time 13	oldlstat 84	sched_getparam 155	fsetxattr 228	unlinkat 301
mknod 14	readlink 85	sched_setscheduler 156	getxattr 229	renameat 302
chmod 15	uselib 86	sched_getscheduler 157	lgetxattr 230	linkat 303
lchown 16	swapon 87	sched_yield 158	fgetxattr 231	symlinkat 304
break 17	reboot 88	sched_get_priority_	listxattr 232	readlinkat 305

		max 159		
oldstat 18	readdir 89	sched_get_priority_min 160	llistxattr 233	fchmodat 306
lseek 19	mmap 90	sched_rr_get_interval 161	flistxattr 234	faccessat 307
getpid 20	munmap 91	nanosleep 162	removexattr 235	pselect6 308
mount 21	truncate 92	mremap 163	lremovexattr 236	ppoll 309
umount 22	ftruncate 93	setresuid 164	fremovexattr 237	unshare 310
setuid 23	fchmod 94	getresuid 165	tkill 238	set_robust_list 311
getuid 24	fchown 95	vm86 166	sendfile64 239	get robust list 312
stime 25	getpriority 96	query_module 167	futex 240	splice 313
ptrace 26	setpriority 97	poll 168	sched_setaffinity 241	sync_file_range 314
alarm 27	NR_profil 98	nfsservctl 169	sched_getaffinity 242	tee 315
oldfstat 28	NR_statfs 99	setresgid 170	set_thread_area 243	vmsplice 316
pause 29	fstatfs 100	getresgid 171	get_thread_area 244	move_pages 317
utime 30	NR_ioperm 101	prctl 172	io_setup 245	getcpu 318
stty 31	NR_socketcall 102	rt_sigreturn 173	io_destroy 246	epoll_pwait 319
gtty 32	NR_syslog 103	rt_sigaction 174	io_getevents 247	utimensat 320
access 33	NR_setitimer 104	rt_sigprocmask 175	io_submit 248	signalfd 321
nice 34	getitimer 105	rt_sigpending 176	io_cancel 249	timerfd_create 322
ftime 35	stat 106	rt_sigtimedwait 177	fadvise64 250	eventfd 323
sync 36	lstat 107	rt_sigqueueinfo 178	exit_group 252	fallocate 324
kill 37	fstat 108	rt_sigsuspend 179	lookup_dcookie 253	timerfd_settime 325
rename 38	olduname 109	pread64 180	epoll_create 254	timerfd_gettime 326
mkdir 39	iopl 110	pwrite64 181	epoll_ctl 255	signalfd4 327
rmdir 40	vhangup 111	chown 182	epoll_wait 256	eventfd2 328
dup 41	idle 112	getcwd 183	remap_file_pages 257	epoll_create1 329
pipe 42	vm86old 113	capget 184	set_tid_address 258	dup3 330
times 43	wait4 114	capset 185	timer_create 259	pipe2 331
prof 44	swapoff 115	sigaltstack 186	timer_settime 260	inotify_init1 332
brk 45	sysinfo 116	sendfile 187	timer_gettime 261	preadv 333
setgid 46	ipc 117	getpmsg 188	timer_getoverrun 262	pwritev 334
getgid 47	fsync 118	putpmsg 189	timer_delete 263	rt_tgsigqueueinfo 335
signal 48	sigreturn 119	vfork 190	clock_settime 264	perf_event_open 336
geteuid 49	clone 120	ugetrlimit 191	clock_gettime 265	recvmmsg 337
getegid 50	setdomainname 121	mmap2 192	clock_getres 266	fanotify_init 338
acct 51	uname 122	truncate64 193	clock_nanosleep 267	fanotify_mark 339
umount2 52	modify_ldt 123	ftruncate64 194	statfs64 268	prlimit64 340
lock 53	adjtimex 124	stat64 195	fstatfs64 269	name_to_handle_at 341
ioctl 54	mprotect 125	lstat64 196	tgkill 270	open_by_handle_at 342
fcntl 55	sigprocmask 126	fstat64 197	utimes 271	clock_adjtime 343
mpx 56	create_module 127	lchown32 198	fadvise64_64 272	syncfs 344
setpgid 57	init module 128	getuid32 199	vserver 273	sendmmsg 345
ulimit 58	delete module 129	getgid32 200	mbind 274	setns 346
oldolduname 59	get_kernel_syms 130	geteuid32 201	get_mempolicy 275	process_vm_readv 347
umask 60	quotactl 131	getegid32 202	set_mempolicy 276	process_vm_writev 348
chroot 61	getpgid 132	setreuid32 203	mq_open 277	kcmp 349
ustat 62	fchdir 133	setregid32 204	mq_unlink 278	finit module 350
dup2 63	bdflush 134	getgroups32 205	mq_timedsend 279	
getppid 64	sysfs 135	setgroups32 206	mq_timedreceive 280	
getpgrp 65	NR_personality 136	fchown32 207	mq_notify 281	
setsid 66	NR_afs_syscall 137	setresuid32 208	mq_getsetattr 282	
sigaction 67	NR_setfsuid 138	getresuid32 209	kexec_load 283	
sgetmask 68	setfsgid 139	setresgid32 210	waitid 284	
ssetmask 69	llseek 140	getresgid32 211	add key 286	
setreuid 70	getdents 141	chown32 212	request_key 287	

Приклади оголошення системних викликів у програмі:

```
#Номер виклику звернення до ядра ОС
LINUX_SYSCALL equ 0x80
#Номери системних викликів
SYS_EXIT  equ 1
SYS_READ  equ 3
SYS_WRITE equ 4
SYS_OPEN  equ 5
```

```
SYS_CLOSE equ 6
SYS_BRK   equ 45

#Стандартні номери (дескриптори) потоків введення/виведення
STDIN  equ  0
STDOUT equ  1
STDERR equ  2
```

Таким чином, використовуючи всі регістри загального призначення, можна передати максимум 6 параметрів. Такий спосіб виклику (з передачею параметрів через регістри) називається швидким `fastcall`. В інших системах (наприклад, BSD) можуть застосовуватися інші способи виклику.

Необхідно зауважити, що не потрібно використовувати системні виклики скрізь, де тільки можна, без особливої необхідності. В різних версіях ядра порядок аргументів у деяких системних викликах відрізняється, і це спричиняє помилки, які досить важко знайти. Тому варто використовувати функції стандартної бібліотеки Сі, так як їх сигнатури не змінюються, що забезпечує переносимість коду на Сі. Якщо ж пишеться невелика частина самого навантаженого коду і є недопустимими накладні витрати, які вносить виклик стандартної бібліотеки Сі, тоді використовуються системні виклики.

Розглянемо для прикладу системний виклик `write`, який дозволяє виводити дані через один з відкритих потоків введення-виведення (на екран або у файл). Системний виклик має номер 4 і приймає три параметри: номер (дескриптор) потоку виведення, адресу пам'яті даних, кількість цих даних в байтах.

Інший важливий системний виклик – `_exit`, який використовується для завершення програми. Він має номер 1 і приймає один параметр – код завершення програми.

Тоді програма, яка друкує повідомлення на екран і завершає роботу з поверненням коду у середовище Linux буде наступною:

```
SYS_EXIT  equ 1
SYS_WRITE equ 4
STDOUT    equ 1
section .data
msg db "Привіт світ", 0xA  ;  0xA -> '\0'
msg_len   equ $-msg

section .text
global _start
_start:
  mov eax,SYS_WRITE ; системний виклик write
  mov ebx,STDOUT    ; виведення на екран
  mov ecx,msg
  mov edx,msg_len
  int 80h           ; виконання системного виклику (звернення до ядра)

  mov eax,SYS_EXIT  ; системний виклик _exit
  mov ebx,0         ; код повернення
  int 80h           ; звернення до ядра
```

Для введення даних (як з клавіатури, так із файлів) використовується виклик `read`, який має номер 3. Він має три параметри: номер (дескриптора) потоку введення, адресу пам'яті, де будуть розміщені вхідні дані, кількість цих даних в байтах. Після виклику `read` *потрібно проаналізувати* результат його роботи. Якщо читання пройшло успішно – виклик поверне кількість прочитаних символів. Виклик повертає 0 у ситуації "кінець файлу". Така ситуація виникає коли досягнуто кінець файлу і в ньому більше немає даних або натиснуто комбінацію клавіш CTRL+D. Від'ємне значення вказує на помилку при читанні. Приклад програми, яка читає дані з консолі і виводить їх на екран:

```
SYS_EXIT  equ 1
SYE_READ  equ 3
SYS_WRITE equ 4
STDIN     equ 0
```

```
STDOUT    equ 1
section .bss
msg resb 64
msg_len   equ $-msg
global _start
section .text
_start:
  nop
  mov eax,SYS_READ   ; системний виклик read
  mov ebx,STDIN      ; читання з консолі
  mov ecx,msg        ; адреса буфера, коди поміщаються дані
  mov edx, msg_len   ; довжина даних
  int 0x80           ; звернення до ядра
  mov edx,eax        ; результат системного виклику повертається в eax
  jl label
  mov eax, SYS_WRITE ; системний виклик write
  mov ebx, STDOUT    ; виведення на екран
  mov ecx, msg       ; адреса даних
  mov edx, msg_len   ; довжина даних
  int 80h            ; виконання системного виклику (звернення до ядра)
  label:
  mov eax,SYS_EXIT   ; системний виклик _exit
  mov ebx,edx        ; код результату читання або msg_len
  int 0x80           ; звернення до ядра
```

Для роботи з файлами потрібно створити додаткові потоки введення/виведення на додаток до стандартних потоків. Додаткові потоки створюються системним викликом `open` з номером 5. Виклик має три параметри: *адреса рядка тексту*, який містить ім'я файлу (ім'я має закінчуватися нульовим байтом, як відіграє роль обмежувача); число, яке задає *режим використання файлу* (читання, запис), значення якого задається бітовою стрічкою; *права доступу до файлу* (задається тільки при створенні файлу, переважно як `0666q`).

Деякі прапори режиму використання файлу:

O_RDONLY	Тільки читання	000h
O_WRONLY	Тільки запис	001h
O_RDWR	Читання і запис	002h
O_CREAT	Дозволити створення файлу	040h
O_EXCL	Вимагати створення файлу	080h
O_TRUNC	Якщо файл існує, знищити його вміст	200h
O_APPEND	Якщо файл існує добавити у кінець	400h

Наприклад комбінації прапорів `O_WRONLY|O_CREAT|O_TRUNC` мови Сі у Linux задається значенням `241h`.

Результат виклику `open` повертається в регістрі `eax`. Якщо виклик закінчився успішно, то `eax` містить дескриптор відкритого файлу (номер потоку введення або виведення). Саме цей дескриптор потрібно використовувати як перший параметр в командах `read` і `write` для роботи з файлами. Звичайно це значення копіюється у спеціально відведену область пам'яті. Якщо виклик закінчився не успішно, то `eax` містить негативне значення.

Коли робота з файлом закінчена його потрібно закрити системним викликом `close`, який має номер 6. Виклик має один параметр, який дорівнює дескриптору файлу, який потрібно закрити.

Приклад:

```
SYS_open  equ 5
SYS_close equ 6
```

```
    SYS_w     equ 0101h   ; прапор запису
    SYS_mod   equ 0666q   ; права доступу
    SYS_write equ 4
    SYS_exit  equ 1
    segment .bss          ; сегмент неініціалізованих даних
       fd   resb  1
    segment .data         ; сегмент ініціалізованих даних
       file db "7.dat\0"
       msg  db "Привіт світ",10
       len equ $-msg
    segment .text
    global _start         ; експорт точки входу в програму
    _start:
       nop
       mov eax,SYS_open
       mov ebx,file
       mov ecx,SYS_w
       mov edx,SYS_mod
       int 0x80           ; відкриття файлу із заданим прапором і правами
       mov [fd],eax       ; запис файлового дескриптора
       jl label           ; аналіз результату

       mov eax,SYS_write
       mov ebx,[fd]
       mov ecx,msg
       mov edx,len
       int 0x80

       mov eax,SYS_close
       mov ebx,[fd]
       int 0x80        ; запис у файл повідомлення

    label:
       mov eax,SYS_exit
       mov ebx,[fd]
       int 0x80           ; завершення програми з поверненням дескриптора файлу
    ret
```

Для роботи із системними викликами з різним число параметрів можна використати макровизначення `my_syscall`, записавши його у файл `my_syscall.inc`:

```
%macro my_syscall 1-*
%rep %0
%rotate -1
  push dword %1
%endrep
  pop eax
%if %0 > 1
  pop ebx
  %if %0 > 2
    pop ecx
    %if %0 > 3
      pop edx
      %if %0 > 4
      pop esi
      %if %0 > 5
        pop edi
        %if %0 > 6
          %error " Забагато параметрів для Linux системних викликів"
        %endif
      %endif
      %endif
    %endif
```

```
  %endif
%endif
  int 80h
%endmacro
```

Використання макросу `my_syscall` значно скорочує текст програми виведення стрічки з пам'яті на екран:

```
%include "my_syscall.inc"
section .data
msg db "Hello world", 10
msg_len equ $-msg
section .text
global _start
_start:
my_syscall 4, 1, msg, msg_len
my_syscall 1, 0
```

1.2. Параметри командного рядка

При запуску програм в ОС Linux, крім імені програми, можуть задаватися і параметри командного рядка. Наприклад, вони вказуються при запуску NASM

```
nasm –f elf prog.asm –o pro
```

При запуску програми, ОС Linux відводить у її адресному просторі спеціальну область пам'яті, в якій розміщаються рядки, що складають командний рядок. Інформація про адреси цих рядків, разом із загальною їх кількістю поміщається у стек запускаємої задачі і після цього їй передається керування. Таким чином, в момент, коли програма починає виконуватися з позначки `_start`, на верхівці стеку (тобто за адресою [esp]) розміщується 4-байтне ціле, рівне кількості елементів командного рядка (включно з іменем програми). В наступній комірці стеку [esp+4] розміщується адреса пам'яті з іменем програми, за яким її викликали. В наступних комірках з [esp+8] розміщуються адреси першого і наступного параметрів. Кожний елемент командного рядка зберігається у пам'яті як рядок, що закінчується нульовим байтом.

```
extern printf
segment .data
fmt: db "%s",15,0
segment .text
global main
main:
  mov ecx,[esp+4]   ; argc
  mov edx,[esp+8]   ; argc
top:
  push ecx  ; збереження регістрів, які використовує printf
  push edx
  push dword [edx]
  push dword fmt
  call printf
  add esp,8    ;  вилучення 2-х параметрів

  pop edx     ; відновлення регістрів для printf
  pop ecx
  add edx,4   ; вказівник на наступний аргумент
  dec ecx     ; лічильник аргументів
  jnz top
  ret
```

Запитання.

1. Які є типи системних викликів.
2. Передача параметрів у системні виклики через регістри загального призначення.

3. Системний виклик для читання з консолі.
4. Системний виклик для виведення на екран.
5. Системні виклики для роботи з файлами. Права доступу до файлу.
6. Структура програми асемблера для запису даних у файл.
7. Структура програми асемблера для читання даних з файлу.
8. Макровизначення для роботи із системними викликами із різним числом параметрів.
9. Параметри командного рядка і їх розміщення у стеку.

Завдання.

1. Написати програму на NASM, яка виводить на екран дві стрічки
   `Привіт всім`
   `студентам, які вивчають Intel асемблер.`

2. Написати програму на NASM, яка зчитує з консолі стрічку ”Привіт світ” і виводить на екран.

3. Написати програму на NASM, яка записує у файл стрічку ”Привіт світ”.

4. Написати програму на NASM, яка читає вміст текстового файлу і виводить на екран.

5. Написати програму на NASM, яка читає вміст текстового файлу і виводить на екран з використанням макросу `my_syscall`.

6. Написати програму на NASM, яка читає текстовий файл з одного рядка “Привіт всім” і замість нього записує “Привіт студентам”.

7. Написати програму на NASM, яка читає текстовий файл з одного рядка “Привіт всім” і добавляє у кінець наступний рядок “Привіт студентам”.

8. Написати програму на NASM, яка копіює файл `file1` у файл `file2`.

9. Написати програму на NASM, яка копіює файл `file1` у файл `file2` записує у файл з використанням макросу `my_syscall`.

10. Написати програму на NASM, яка виводить параметри командного рядка на екран.

11. Написати програму на NASM, яка записує параметри командного рядка у файл з використанням макросу `my_syscall`.

12. Написати программу на NASM, яка виводить поточний час в годинах, хвилинах, секундах.

2. Приклади для самостійної роботи

1. helloworld.asm – програма виведення повідомлення на екран

```
SECTION .data
msg db 'Hello World!', 0Ah  ; 0Ah => \n

SECTION .text
global  _start

_start:

    mov     edx, 13
    mov     ecx, msg
    mov     ebx, 1
    mov     eax, 4
    int     80h

    mov     ebx, 0   ; повернення 0 на виході - 'No Errors'
    mov     eax, 1   ; виклик SYS_EXIT (kernel opcode 1)
    int     80h
```

```
$ nasm -f elf helloworld.asm
$ ld -m elf_i386 helloworld .o -o helloworld
$ ./helloworld
Hello World!
```

2. helloworld-len.asm – обчисленням довжини повідомлення і виведення його на екран

```
SECTION .data
msg db  'Hello, new world!', 0Ah

SECTION .text
global _start

_start:

  mov ebx, msg  ; переслати адресу msg у EBX
  mov eax, ebx  ; скопіювати адресу з EBX в EAX

nextchar:
    cmp byte [eax], 0   ; порівняти byte адресований EAX з 0 (0 – кінець стрічки)
    jz  finished    ; умовний перехід по 0 на позначку ’finished’
    inc eax         ; збільшення на 1 адреси в EAX
    jmp nextchar    ; безумовний перехід на позначку 'nextchar'

finished:
    sub eax, ebx    ; різниця адрес в EBX і EAX

    mov  edx, eax   ; EAX містить число байтів стрічки msg

    mov  ecx, msg   ; виведення стрічки на екран
    mov  ebx, 1
    mov  eax, 4
    int  80h

    mov  ebx, 0     ; завершення програми з кодом 0
    mov  eax, 1
    int  80h
```

```
$ nasm -f elf helloworld-len.asm
$ ld -m elf_i386 helloworld-len.o -o helloworld-len
$ ./helloworld-len1
Hello, new world!
```

3. helloworld-len1.asm – обчисленням довжини повідомлення за допомогою підпрограми і виведення його на екран

```
SECTION .data
msg db 'Hello, new world!', 0Ah

SECTION .text
global  _start

_start:

    mov  eax, msg    ; переслати адресу msg у EAX
    call strlen      ; виклик підпрограми обчислення довжини стрічки

    mov  edx, eax    ; підпрограма повертає результат в EAX
    mov  ecx, msg    ; виведення стрічки на екран
    mov  ebx, 1
    mov  eax, 4
    int  80h

    mov  ebx, 0      ; завершення програми з кодом 0
    mov  eax, 1
    int  80h

strlen:              ; оголошення підпрограми
    push  ebx        ; збереження EBX у стеку
```

```
    mov   ebx, eax    ; скопіювати адресу msg з EAX у EBX

nextchar:             ; визначення довжини msg
    cmp   byte [eax], 0
    jz    finished
    inc   eax
    jmp   nextchar

finished:
    sub   eax, ebx
    pop   ebx          ; відновлення значення EBX із стеку
    ret                ; повернення з підпрограми
```

```
$ nasm -f elf helloworld-len1.asm
$ ld -m elf_i386 helloworld-len1.o -o helloworld-len1
$ ./helloworld-len1
Hello, new world!
```

4. helloworld-inc.asm – друк повідомлення на екран з використання директиви include

```
; functions.asm – бібліотека підпрограм
;-----------------------------------------
; int slen(String message)
; підпрограма обчислення довжини стрічки
slen:
    push    ebx
    mov     ebx, eax

nextchar:
    cmp     byte [eax], 0
    jz      finished
    inc     eax
    jmp     nextchar

finished:
    sub     eax, ebx
    pop     ebx
    ret


;-----------------------------------------
; void sprint(String message)
; підпрограма виведення на екран стрічки
sprint:
    push    edx
    push    ecx
    push    ebx
    push    eax
    call    slen

    mov     edx, eax
    pop     eax

    mov     ecx, eax
    mov     ebx, 1
    mov     eax, 4
    int     80h

    pop     ebx
    pop     ecx
    pop     edx
    ret
```

```
;-----------------------------------------
; void exit()
; Підпрограма виходу із програми
quit:
    mov     ebx, 0
    mov     eax, 1
    int     80h
    ret
```

; helloworld-inc.asm

```
;-----------------------------------------------------------
%include  'functions.asm'  ; підключення зовнішнього файлу

SECTION .data
msg1  db   'Hello, new world!', 0Ah               ; перша стрічка
msg2  db   'This is how we recycle in NASM.', 0Ah ; друга стрічка

SECTION .text
global  _start

_start:

    mov   eax, msg1   ; переслати адресу 1-ї стрічки в EAX
    call  sprint      ; виклик підпрограми

    mov   eax, msg2   ; переслати адресу 1-ї стрічки в EAX
    call    sprint    ; виклик підпрограми

    call  quit        ; виклик підпрограми завершення програми
```

```
$ nasm -f elf helloworld-inc.asm
$ ld -m elf_i386 helloworld-inc.o -o helloworld-inc
$ ./helloworld-inc
Hello, new world!
This is how we recycle in NASM.
This is how we recycle in NASM.
```

Поміняти у основній програмі оголошення стрічок, створити виконуваний файл і запустити програму на виконання:

```
msg1  db  'Hello, new world!', 0Ah, 0h    ; добавлено null байт завершувач
msg2  db  'This is how we recycle in NASM.', 0Ah, 0h
```

```
$ ./helloworld-inc
Hello, new world!
This is how we recycle in NASM.
```

5. helloworld-lf.asm – друк повідомлення на екран з використання директиви include і підпрограми друку стрічок з добавленням символу 0Ah (line feed) друку з нового рядка.

; functions.asm

```
;-----------------------------------------
; int slen(String message)
; Підпрограма обчислення довжини стрічки
slen:
    push    ebx
    mov     ebx, eax

nextchar:
    cmp     byte [eax], 0
    jz      finished
    inc     eax
    jmp     nextchar
```

```
finished:
    sub     eax, ebx
    pop     ebx
    ret


;-----------------------------------------
; void sprint(String message)
; Підпрограма друку стрічки
sprint:
    push    edx
    push    ecx
    push    ebx
    push    eax
    call    slen

    mov     edx, eax
    pop     eax

    mov     ecx, eax
    mov     ebx, 1
    mov     eax, 4
    int     80h

    pop     ebx
    pop     ecx
    pop     edx
    ret


;-----------------------------------------
; void sprintLF(String message)
; Підпрограма друку стрічки з добавленням символу друку з нового рядка (OAh)
sprintLF:
    call    sprint

    push  eax       ; збереження eax у стек
    mov   eax, 0Ah  ; запис 0Ah у eax
    push  eax       ; запис eax у стек
    mov   eax, esp  ; збереження esp у eax для sprint
    call  sprint    ; виклик підпрограми sprint
    pop   eax       ; вилучення символу OAh із стеку
    pop   eax       ; відновлення eax у стан перед викликом
    ret             ; вихід із підпрограми


;-----------------------------------------
; void exit()
; Підпрограми виходу з програми
quit:
    mov     ebx, 0
    mov     eax, 1
    int     80h
    ret
```

; **helloworld-lf.asm**

```
;-----------------------------------------
%include        'functions.asm'

SECTION .data
msg1  db 'Hello, new world!', 0h               ; увага, вилучено символ 0Ah
msg2  db 'This is how we recycle in NASM.', 0h  ; увага, вилучено символ 0Ah
```

```
SECTION .text
global  _start

_start:

    mov  eax, msg1
    call sprintLF   ; виклик нової підпрограми друку

    mov  eax, msg2
    call sprintLF   ; виклик нової підпрограми друку

    call    quit
```

```
$ nasm -f elf helloworld-lf.asm
$ ld -m elf_i386 helloworld-lf.o -o helloworld-lf
$ ./helloworld-lf
Hello, new world!
This is how we recycle in NASM.
```

6. helloworld-args – передача аргументів через командний рядок

```
%include  'functions.asm'

SECTION .text
global  _start

_start:

    pop   ecx        ; перше значення у стеку – це число аргументів

nextArg:
    cmp   ecx, 0h    ; перевірка, чи є ще аргументи зліва
    jz    noMoreArgs  ; якщо нема, то перейти на позначку noMoreArgs
    pop   eax         ; прочитати наступний аргумент із стеку
    call  sprintLF    ; виклик підпрограми друку
    dec   ecx         ; зменшити на 1 ecx (число аргументів зліва)
    jmp   nextArg     ; безумовно перейти на позначку nextArg

noMoreArgs:
    call    quit
```

```
$ nasm -f elf helloworld-args.asm
$ ld -m elf_i386 helloworld-lf.o -o helloworld-args
$ ./helloworld-args "This is one argument" "This is another" 101
$ ./helloworld-args
This is one argument
This is another
101
```

7. helloworld-input.asm – зчитування даних, введених з консолі і їх друк

```
%include 'functions.asm'

SECTION .data
msg1  db  'Please enter your name: ', 0h   ; повідомлення до введення
msg2  db  'Hello, ', 0h                    ; повідомлення після введення

SECTION .bss
sinput:  resb  255  ; буфер 255 байт для стрічок повідомлення

SECTION .text
global  _start
```

```
_start:

    mov   eax, msg1
    call  sprint

    mov   edx, 255        ; число байтів для читання
    mov   ecx, sinput     ; адресу буфера в ecx
    mov   ebx, 0          ; write у STDIN файл
    mov   eax, 3          ; виклик SYS_READ (kernel opcode 3)
    int   80h

    mov   eax, msg2
    call  sprint

    mov   eax, sinput     ; адресу буфера в eax (input містить символ '\n')
    call  sprint          ; виклик підпрограми друку

    call  quit
```

```
$ nasm -f elf helloworld-input.asm
$ ld -m elf_i386 helloworld-input.o -o helloworld-input
$ ./helloworld-input
Please enter your name: Ivanenko Ivan
Hello, Ivanenko Ivan
```

7. helloworld-10.asm – друк ascii символів

```
%include  'functions.asm'

SECTION .text
global  _start

_start:

    mov   ecx, 0   ; ecx ініціалізувати в 0

nextNumber:
    inc     ecx    ;збільшити на 1 ecx

  mov  eax, ecx    ; переслати адресу числа integer у eax
  add   eax, 48    ; додати 48 до числа, щоб перетворити int у код ascii
  push  eax        ; зберегти eax у стек
  mov   eax, esp   ; отримати адресу символу із стеку
  call  sprintLF   ; виклик підпрограми друку

  pop   eax        ; очистити стек від непотрібних байтів
  cmp   ecx, 10    ; порівняння ecx і 10
  jne   nextNumber ; безумовний перехід на позначку nextNumber

  call quit
```

```
$ nasm -f elf helloworld-10.asm;
$ ld -m elf_i386 helloworld-10.o -o helloworld-10
$ ./helloworld-10
1
2
3
4
5
6
7
8
9
:
```

8. helloword-itoa.asm - друк цілих чисел

```
; functions.asm
;-----------------------------------------
; void iprint(Integer number)
; Підпрограма друку цілих чисел (itoa)
iprint:
    push  eax    ; зберігання eax в стеку
    push  ecx    ; зберігання ecx в стеку
    push  edx    ; зберігання edx в стеку
    push  esi    ; зберігання esi в стеку
    mov   ecx, 0 ; лічильник байтів, які необхідно в кінці роздрукувати

divideLoop:
    inc  ecx       ; лічильник числа символів
    mov  edx, 0    ; обнулити edx
    mov  esi, 10   ; переслати 10 у esi
    idiv esi       ; поділити eax на esi
    add  edx, 48   ; перетворити edx в ascii код, edx містить залишок від ділення
    push edx       ; помістити edx (ascii код) у стек
    cmp  eax, 0    ; чи можна ціле integer ще поділити?
    jnz  divideLoop  ; перейти по не нуль на позначку divideLoop

printLoop:
    dec  ecx       ; лічильник байтів, які поміщені у стек
    mov  eax, esp  ; переслати вказівник стеку в eax для друку
    call sprint    ; виклик функції друку
    pop  eax       ; вилучення останнього символу із стеку (збільшення esp)
    cmp  ecx, 0    ; чи вибрано і надруковано всі символи із стеку?
    jnz  printLoop ; безумовний перехід по не нуль на позначку printLoop

    pop  esi       ; відновлення esi із стеку
    pop  edx       ; відновлення edx із стеку
    pop  ecx       ; відновлення ecx із стеку
    pop  eax       ; відновлення eax із стеку
    ret


;-----------------------------------------
; void iprintLF(Integer number)
; Підпрограма друку цілих чисел із символом "\n", linefeed (itoa)
iprintLF:
    call   iprint  ; виклик підпрограми iprint

    push  eax      ; збереження eax у стеку
    mov   eax, 0Ah ; переслати 0Ah у eax, 0Ah ascii символ "\n"
    push  eax      ; переслати eax у стек, щоб отримати адресу
    mov   eax, esp ; переслати esp у eax для sprint
    call  sprint   ; виклик підпрограми sprint
    pop   eax      ; вилучити символ "\n" із стеку
    pop   eax      ; відновити оригінальне значення eax
    ret


;-----------------------------------------
; int slen(String message)
; Підпрограма обчислення довжини стрічки
slen:
    push    ebx
    mov     ebx, eax

nextchar:
    cmp     byte [eax], 0
```

```
    jz      finished
    inc     eax
    jmp     nextchar

finished:
    sub     eax, ebx
    pop     ebx
    ret


;-----------------------------------------
; void sprint(String message)
; Підпрограма друку стрічки
sprint:
    push    edx
    push    ecx
    push    ebx
    push    eax
    call    slen

    mov     edx, eax
    pop     eax

    mov     ecx, eax
    mov     ebx, 1
    mov     eax, 4
    int     80h

    pop     ebx
    pop     ecx
    pop     edx
    ret


;-----------------------------------------
; void sprintLF(String message)
; Підпрограма друку стрічки із символом ”\n”
sprintLF:
    call    sprint

    push    eax
    mov     eax, 0AH
    push    eax
    mov     eax, esp
    call    sprint
    pop     eax
    pop     eax
    ret


;-----------------------------------------
; void exit()
; Підпрограма виходу із програми
quit:
    mov     ebx, 0
    mov     eax, 1
    int     80h
    ret

; helloword-itoa.asm
;-----------------------------------------
%include  'functions.asm'
```

```
SECTION .text
global  _start

_start:

    mov   ecx, 0

nextNumber:
    inc   ecx
    mov   eax, ecx
    call  iprintLF      ; виклик нової підпрограми друку цілих чисел (itoa)
    cmp   ecx, 10
    jne   nextNumber

    call  quit
```

```
$ nasm -f elf helloworld-itoa.asm
$ ld -m elf_i386 helloworld-itoa.o -o helloworld-itoa
$ ./helloworld-itoa
1
2
3
4
5
6
7
8
9
10
```

9.asm – виведення на екран параметрів командного рядка з нового рядка

```
segment .data
nwl db 10
segment .text
global _start

strlen:         ; rag1 – адреса командного рядка
  push ebp
  mov ebp,esp
  push esi
  xor eax,eax
  mov esi,[ebp+8]   ; arg1
.lp:
  cmp byte [esi],0
  jz .quit
  inc esi
  inc eax
  jmp short .lp
.quit:
  pop esi
  pop ebp
  ret

newline:
  pushad
  mov eax,4
  mov ebx,1
  mov ecx,nwl
  mov edx,1
  int 0x80
  popad
  ret
```

```
_start:
  mov ecx,[esp]
  mov esi,esp
  add esi,4
again:
  push dword [esi]
  call strlen
  add esp,4
  push esi
  push ecx

  mov eax,4
  mov ebx,1
  mov ecx,[esi]
  mov edx,eax
  int 0x80

  call newline
  pop ecx
  pop esi
  add esi,4
  loop again

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

```
$ nasm -f elf 9.asm
$ ld -m elf_i386 9.o -o 9
$ ./9 abc 123 xyz
./9
abc
123
Xyz
```

10. execute.asm – виклик Linux команд з програми асемблера

```
%include  'functions.asm'

; SECTION .data
; command  db  '/bin/ls', 0h       ; команда для виконання
; arg1     db  '-l', 0h

; SECTION .data
; command  db  '/bin/sleep', 0h    ; команда для виконання
; arg1     db  '5', 0h

SECTION .data
command   db  '/bin/echo', 0h      ; команда для виконання
arg1      db  'Hello World!', 0h
arguments dd  command
          dd arg1  ;аргументи, які передаються через командний рядок
                   ;(у даному випадку один)
          dd  0h   ; кінець структури
environment  dd 0h ; аргументи, які передаються, як змінні середовища
                   ; (у цьому випадку відсутні)

SECTION .text
global  _start

_start:
```

```
    mov  edx, environment  ; адреса змінних середовища
    mov  ecx, arguments    ; адреса аргументів для передачі через командний рядок
    mov  ebx, command      ; адреса файлу для виконання
    mov  eax, 11           ; виклик SYS_EXECVE (kernel opcode 11)
    int  80h

    call    quit           ; виклик підпрограми виходу
```

```
$ nasm -f elf execute.asm
$ ld -m elf_i386 execute.o -o execute
$ ./execute
Hello World!
```

11. fork.asm – запуск дочірнього процесу із батьківського

```
%include  'functions.asm'

SECTION .data
childMsg    db  'This is the child process', 0h     ; стрічка повідомлення
parentMsg   db  'This is the parent process', 0h    ; стрічка повідомлення

SECTION .text
global  _start

_start:

    mov  eax, 2   ; виклик SYS_FORK (kernel opcode 2)
    int  80h

    cmp  eax, 0   ; якщо eax == 0, то це дочірний процес
    jz   child    ; перехід, якщо eax==0, до позначки child

parent:
    mov  eax, parentMsg ; в середині батьківського процесу переслати
                        ; адресу parentMsg у eax
    call sprintLF       ; виклик підпрограми друку стрічок з добавленням '\n'

    call quit           ; вихід з батьківського процесу

child:
    mov  eax, childMsg  ; в середині дочірнього процесу переслати
                        ; адресу childMsg у eax
    call sprintLF       ; виклик підпрограми друку стрічок з добавленням '\n'

    call quit           ; вихід з дочірнього процесу
```

```
$ nasm -f elf fork.asm
$ ld -m elf_i386 fork.o -o fork
$ ./fork
This is the parent process
This is the child process
```

13. time.asm – виведення часового інтервалу

```
%include 'functions.asm'

SECTION .data
msg  db     'Seconds since Jan 01 1970: ', 0h    ; стрічка повідомлення

SECTION .text
global  _start

_start:

    mov   eax, msg  ; переслати адресу стрічки msg у eax для друку
```

```
    call  sprint     ; виклик підпрограми друку стрічки

    mov   eax, 13    ; виклик SYS_TIME (kernel opcode 13)
    int   80h        ; звернення до ядра

    call  iprintLF   ; виклик підпрограми друку цілих чисел + символ '\n'
    call  quit       ; виклик пыдпрограми виходу
```

```
$ nasm -f elf time.asm ~$
$ ld -m elf_i386 time.o -o time
$ ./time
Seconds since Jan 01 1970: 1374995660
```

Лабораторна робота № 9.

9. Структури даних

Мета роботи: вивчення і робота із структурами даних на асемблері

1. Короткі теоретичні відомості

В асемблері відсутнє поняття «структура», але є «блок пам'яті», його адреса і зміщення в цьому блоці. Нехай є блок пам'яті розміром 4 байти розміщений за адресою `0x00010000`:

0x21	0x72	0x43	0x17

Це значить, що адреса байту `0x21` дорівнює `0x00010000`, адреса байту `0x72` `0x00010001`. Кажуть, що байт `0x72` розміщений із зміщенням 1 від початку блоку пам'яті, байт `0x43` – із зміщенням 2, а байт `0x17` — із зміщенням 3. Таким чином, `адреса елементу = базовий адрес + зміщення`.

Подібно так в асемблері організована робота із структурами: до базової адреси структури додається зміщення, за яким знаходиться потрібний елемент. Як визначається зміщення? В Сі компілятор керується наступними правилами:

- Вся структура повинна бути вирівняна так, як вирівняний її елемент з найбільшим розміром.
- Кожний елемент знаходиться за найменшою наступною адресою з відповідним вирівнюванням. Якщо потрібно, для цього в структуру включається потрібне число байт-заповнювачів.
- Розмір структури має бути кратним її вирівнюванню. Якщо потрібно, для цього в кінець структури включається потрібне число байтів-заповнювачів.

Приклади (зміщення елементів в байтах; заповнювачі позначені `XX`):

```
struct     Вирівнювання структури: 1, розмір: 1
{          +----+
  char c;  | c  |
};         +----+
           0

struct     Вирівнювання структури: 2, розмір: 4
{          +----+----+----+----+
  char c;  | c  | XX |    s    |
  short s; +----+----+----+----+
};          0         2

struct     Вирівнювання структури: 4, розмір: 8
{          +----+----+----+----+----+----+----+----+
  char c;  | c  | XX   XX   XX |         i         |
  int i;   +----+----+----+----+----+----+----+----+
};          0                   4

struct     Вирівнювання структури: 4, розмір: 8
{          +----+----+----+----+----+----+----+----+
  int i;   |         i         | c  | XX   XX   XX |
  char c;  +----+----+----+----+----+----+----+----+
};          0                   4

struct     Вирівнювання структури: 4, розмір: 12
{          +----+----+----+----+----+----+----+----+----+----+----+----+
  char c;  | c  | XX   XX   XX |         i         |    s    | XX   XX |
  int i;   +----+----+----+----+----+----+----+----+----+----+----+----+
```

```
  short s;  0                   4                   8
};

struct     Вирівнювання структури: 4, розмір: 8
{          +----+----+----+----+----+----+----+----+
  int i;   |         i         | c  | XX |    s    |
  char c;  +----+----+----+----+----+----+----+----+
  short s;  0                   4         6
};
```

В двох останніх прикладах елементи структур одні і ті ж, але розміщені в різному порядку, тому розмір структур різний!

Нехай потрібно написати програму, яка виводить розмір файлу. Для цього потрібно буде викликати функцію `stat(2)` і прочитати дані з структури, яку вона заповнить.

Довідка `man 2 stat`:

```
STAT(2)                  Системні виклики    STAT(2)
 ІМ'Я
       stat, fstat, lstat - отримати статус файла

КОРОТКЕ ОПИСАННЯ
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <unistd.h>

       int stat(const char *file_name, struct stat *buf);

ОПИСАННЯ
       stat повертає інформацію про файл, заданий параметром
       file_name, і заповнює буфер buf.

       Всі ці функції повертають структуру stat, яка містить
       такі поля:

           struct stat {
               dev_t         st_dev;     /* пристрій                   */
               ino_t         st_ino;     /* індексний дескриптор       */
               mode_t        st_mode;    /* режим доступу              */
               nlink_t       st_nlink;   /* кількість жорстких посилань */
               uid_t         st_uid;     /* ідентифікатор
                                            користувача-власника       */
               gid_t         st_gid;     /* ідентифікатор
                                            групи-власника             */
               dev_t         st_rdev;    /* тип пристрою (якщо це
                                            пристрій)                  */
               off_t         st_size;    /* загальний розмір в байтах */
               unsigned long st_blksize; /* розмір блоку вводу-виводу */
                                         /* у файловій системі         */
               unsigned long st_blocks;  /* кількість виділених
                                            блоків                     */
               time_t        st_atime;   /* час останнього доступу     */
               time_t        st_mtime;   /* час останньої
                                            зміни                      */
               time_t        st_ctime;   /* час останньої зміни
                                            стану                      */
           };
```

Потрібно обчислити зміщення поля `st_size`, але для цього потрібно визначити розміри типів – `dev_t`, `ino_t`. Їх можна знайти в заголовковому файлі і взнати, що там позначено за допомогою `typedef`:

```
>cpp /usr/include/sys/types.h | less
typedef __dev_t dev_t;
```

```
__extension__ typedef __u_quad_t __dev_t;
__extension__ typedef unsigned long long int __u_quad_t;
```
Значить, розмір `sizeof(dev_t)` = 8.

Так шукаються і розміри інших елементів, але структура реалізовано інакше. Якщо подивитися на визначення `struct stat` (`cpp /usr/include/sys/stat.h | less`), можна побачити поля з іменами `__pad1`, `__pad2`, `__unused4` і інші (залежить від системи). Ці поля не використовуються, вони потрібні для сумісності, і тому в `man` не описані. Тому простіше доручити компілятору Сі порахувати це зміщення (віднімання з адреси поля адреси структури, дає зміщення):

```
/* 1 - struct.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
  struct stat t;
  /* визначення розміру структури і зміщення поля st_size */
  printf("sizeof = %zu, offset = %td\n",
         sizeof(t), ((void *) &t.st_size) - ((void *) &t) );
  return 0;
}
>gcc struct.c
>./a.out
sizeof = 144, offset = 48
```

В Suse 11.3 x86_64 отримано `sizeof = 144, offset = 48`. В інших системах це значення може відрізнятися згідно вказаних вище причин. Отримане значення `st_size` використовується в асемблерній програмі:

```
/* 2 -  struct.s  - програма визначення розміру файла */
/* >gcc struct.s -o struct  */
.data
str_usage:
        .string "usage: %s filename\n"

printf_format:
        .string "%u\n"
.text
.globl main
main:
        pushl %ebp
        movl  %esp, %ebp
        subl  $144, %esp         /* виділити 144 байти під struct stat  */
        cmpl  $2, 8(%ebp)       /* argc == 2?                          */
        je    args_ok
                                 /* програмі передали не 2 аргументи,
                                    вивести usage                       */
        movl  12(%ebp), %ebx     /* помістити в %ebx адрес масиву argv */
        pushl (%ebx)             /* argv[0]                             */
        pushl $str_usage
        call  printf

        movl  $1, %eax           /* вийти з кодом 1                     */
        jmp   return

args_ok:
        leal  -144(%ebp), %ebx   /* помістити адрес структури в
```

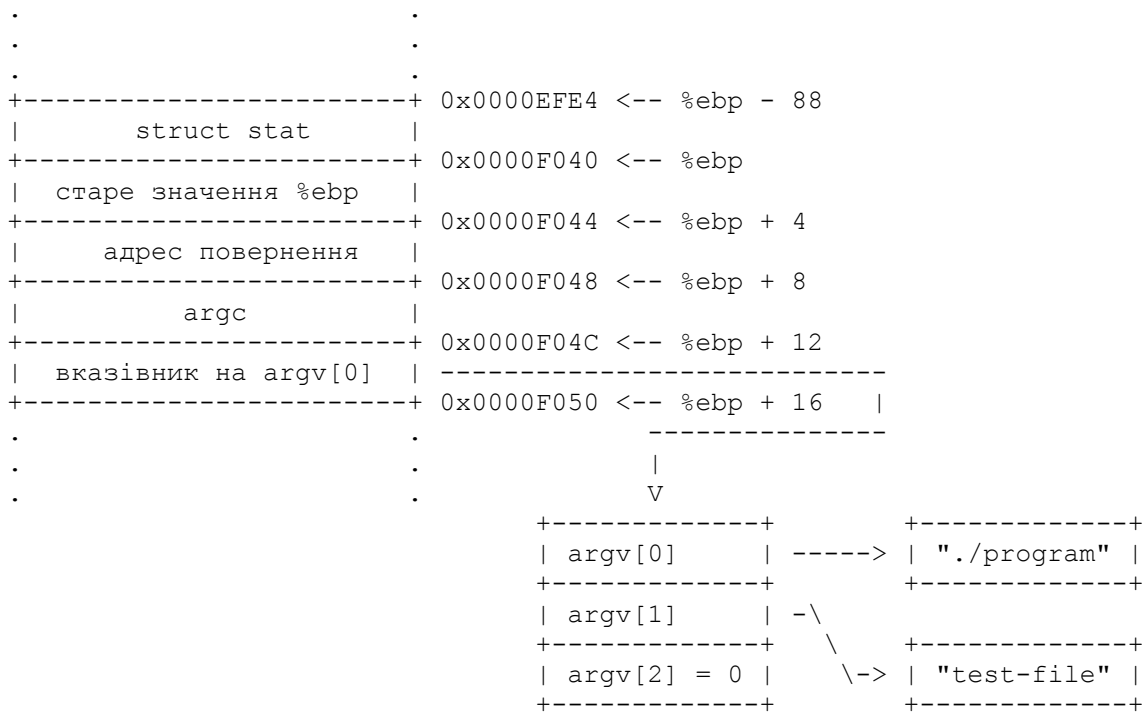
```
                                регістр %ebx                        */
        pushl %ebx

        movl  12(%ebp), %ecx    /* помістити в %ecx адрес масиву argv */
        pushl 4(%ecx)           /* argv[1] — ім'я файлу               */
        call  stat
        cmpl  $0, %eax          /* stat() повернув 0?                 */
        je    stat_ok
        /* stat() повернув помилку, потрібно викликати perror(argv[1]) і
           завершити програму */
        movl  12(%ebp), %ecx
        pushl 4(%ecx)
        call  perror
        movl  $1, %eax
        jmp   return

stat_ok:
        pushl 48(%ebx)          /* потрібне поле із зміщенням 48    */
        pushl $printf_format
        call  printf

        movl  $0, %eax          /* вийти з кодом 0                   */

return:
        movl  %ebp, %esp
        popl  %ebp
        ret
```

Оброблення помилок: якщо передано не 2 аргументи — виводиться описання запуску програми и вихід, якщо `stat(2)` повернув помилку — виводиться повідомлення про помилку і вихід.

Розміщення `argc` і `argv` в стеку. Припустимо, що програма запущена так:

```
>./program test-file
```

Тоді стек буде виглядати так:



Таким чином, в стек поміщаються два параметри: `argc` і вказівник на перший елемент масиву `argv[]`. Десь в пам'яті розміщений блок з трьох вказівників: вказівник на рядок

"./program", вказівник на рядок "test-file" і вказівник NULL. В стек передається адреса цього блоку пам'яті.

1.1. Макроси для роботи зі структурами

В NASM асемблері добавлено макроси для роботи із структурами STRUC, ENDSTRUC. Приклад визначення структури типу mytype:

```
STRUC mytype
   mt_long: resd 1  ; 4 байти
   mt_word: resw 1  ; 2
   mt_byte: resb 1  ; 1 байт
   mt_str: resb 32  ; 32 байти
ENDSTRUC
```

Вказаний код визначає наступні символи: mytype із зміщенням 0, mt_long із зміщенням 0, mt_word із зміщенням 4, mt_byte із зміщенням 6, mt_str із зміщенням 7, mytype_size розміром 39 байтів.

Після визначення структури її можна оголосити в асемблерній програмі використовуючи макроси istruc, at, iend:

```
mystruc:
  istruc mytype
     at mt_long, dd 123456
     at mt_word, dw 1024
     at mt_byte, db 'x'
     at mt_str,  db "Привіт світ!"
  iend
```

Макрос at використовує префікс times для забезпечення зміщення змінних структури відповідно до визначення даних, тому поля структури мають бути оголошені у тому самому порядку як і у визначенні структури.

Якщо дані в полях структури не поміщаються на один рядок, то їх можна продовжити у наступному рядку:

```
     at mt_str, db 123,134,145,156,167,178,189
                db 190,100,0
```

Для звернення до значення поля структури використовується наступний синтаксис [mystruc+mt_long].

Приклад використання макросів для роботи зі структурою:

```
; struct_1.asm
STRUC mytype        ; макрос визначення структури
mt_long:  resd 1
mt_word:  resw 1
mt_byte:  resb 1
mt_str:   resb 32
ENDSTRUC

section .data
mystruct:          ; оголошення структури
 istruc mytype
   at mt_long, dd 123456
   at mt_word, dw 1024
   at mt_byte, db 'x'
   at mt_str,  db 'Привіт світ!',10,0   ; new line
 iend
section .bss
msg resb 16
len equ $-msg
global _start
```

```
section .text
_start:
   mov eax,4   ; виведення на екран
   mov ebx,1
   mov ecx,mystruct+mt_str
   mov edx,24
   int 0x80
   ;
   mov eax,1   ; вихід в Linux
   mov ebx,mytype_size
   int 0x80
   ret
```

```
>./struct_1
echo $?
39
```

Макроси `align`, `alignb` дозволяють вирівнювати код або дані на межу `word`, `dword`, `qword`, `paragraph` або інші значення 2^n. Макрос `align` використовується у секціях `.data`, `.text`, а `alignb` – в секції `.bss`.

```
align 4        ; вирівняти на 4-байтну межу
align 16       ; вирівняти на 16-байтну межу
align 8,db 0   ; вирівняти і заповнити нулями, а не NOP
align 4,resb 1 ; вирівняти на 4-байтну межу у секції BSS
alignb 4       ; еквівалентно до попереднього рядка
```

Приклад використання макросів для вирівнювання елементів структури:

```
; struct_2.asm
; macros defined structure
STRUC mytype
mt_long:  resd 1
mt_word:  resw 1
mt_byte:  resb 2
mt_str:   resb 32
ENDSTRUC
section .data
mystruct:
 istruc mytype
   at mt_long, dd 123456
   at mt_word, dw 1024
   at mt_byte, db 'x'
               align 2
   at mt_str,  db 'Привіт світ!',10,0   ; new line
 iend
section .bss
msg resb 16
len equ $-msg
global _start
section .text
_start:
   mov eax,4   ; виведення на екран
   mov ebx,1
   mov ecx,mystruct+mt_str
   mov edx,24
   int 0x80
   ;
   mov eax,1   ; вихід в Linux
   mov ebx,mytype_size
   int 0x80
   ret
```

```
>./struct_2
>echo $?
```

40

Запитання.

1. Правила вирівнювання структур, прийняті в компіляторі GCC.
2. Визначення зміщення елемента структури в асемблері GNU AS.
3. Розміщення `argc, argv` в стеку при передачі параметрів програмі з консолі.
4. Особливості компіляції файлів з розширенням `*.s`.
5. Макроси асемблера NASM для роботи із структурою.
6. Макроси асемблера NASM для вирівнювання елементів структури.
7. Особливості використання в асемблері GNU AS позначок-чисел.

Завдання.

1. Написати асемблерну програму, яка виводить на екран значення елементів структури аналогічної до Сі структури

```
struct  student {
        char number[2];
        char name[16];
};
```

2. Написати асемблерну програму, яка записує значення елементів 5-х структур у файл. Структура аналогічна до Сі структури завдання 1.

2. Приклади для самостійної роботи

Робота із списками

Програма друку файлу у зворотному порядку

Програма читає із стандартного вводу все до кінця файлу, а потім виводить введені рядки в зворотному порядку. Під час читання рядка елементи поміщаються у зв'язаний список, який проходиться у зворотному порядку з друком елементів.

```
/* 3 - reverse.s */
.data
printf_format:
        .string "<%s>\n"

#define READ_CHUNK 128

.text

/* char *read_str(int *is_eof) */
read_str:
        pushl %ebp
        movl  %esp, %ebp

        pushl %ebx              /* зберегти регістри                */
        pushl %esi
        pushl %edi

        movl  $0,   %ebx        /* прочитано байт                    */
        movl  $READ_CHUNK, %edi /* розмір буфера                     */
        pushl %edi
        call  malloc
        addl  $4, %esp          /* забрати аргументи                  */
        movl  %eax, %esi        /* вказівник на початок буфера        */
        decl  %edi              /* в кінці має бути нульовий байт,
                                   зарезервувати для нього місце      */
```

```
        pushl stdin             /* fgetc() завжди буде викликатися з
                                   цим аргументом                    */

1: /* read_start */
        call  fgetc             /* прочитати 1 символ                */
        cmpl  $0xa, %eax        /* новий рядок '\n'?                 */
        je    2f                /* read_end                          */
        cmpl  $-1, %eax         /* кінець файлу?                     */
        je    4f                /* eof_yes                           */
        movb  %al, (%esi,%ebx,1)  /* записати прочитаний символ в
                                   буфер                             */
        incl  %ebx              /* інкрементувати лічильник
                                   прочитаних байтів                 */
        cmpl  %edi, %ebx        /* буфер заповнений?                 */
        jne   1b                /* read_start                        */

        addl  $READ_CHUNK, %edi /* збільшити розмір буфера           */
        pushl %edi              /* розмір                            */
        pushl %esi              /* вказівник на буфер                */
        call  realloc
        addl  $8, %esp          /* забрати аргументи                  */
        movl  %eax, %esi        /* результати в %eax — новий вказівник
                                                                      */
        jmp   1b                /* read_start                        */
2: /* read_end */

3: /* eof_no */
        movl  8(%ebp), %eax     /* *is_eof = 0                       */
        movl  $0, (%eax)
        jmp   5f                /* eof_end                           */
4: /* eof_yes */
        movl  8(%ebp), %eax     /* *is_eof = 1                       */
        movl  $1, (%eax)
5: /* eof_end */

        movb  $0, (%esi,%ebx,1) /* записати в кінець буфера '\0'      */

        movl  %esi, %eax        /* результат в %eax                  */

        addl  $4, %esp          /* забрати аргумент fgetc()           */

        popl  %edi              /* відновити регістри             */
        popl  %esi
        popl  %ebx

        movl  %ebp, %esp
        popl  %ebp
        ret

/*
struct list_node
{
  struct list_node *prev;
  char *str;
};
*/

.globl main
main:
        pushl %ebp
        movl  %esp, %ebp
```

```
        subl  $4, %esp          /* int is_eof;                          */

        movl  $0, %edi          /* в %edi буде зберігатися вказівник на
                                   попередню структуру                  */

1: /* read_start */
        leal  -4(%ebp), %eax    /* %eax = &is_eof;                      */
        pushl %eax
        call  read_str
        movl  %eax, %esi        /* вказівник на прочитаний рядок
                                   помістити в %esi                     */

        pushl $8                /* виділити 8 байт під структуру        */
        call  malloc

        movl  %edi, (%eax)      /* вказівник на попередню структуру */
        movl  %esi, 4(%eax)     /* вказівник на рядок                   */

        movl  %eax, %edi        /* тепер ця структура — попередня */

        addl  $8, %esp          /* забрати аргументи                     */

        cmpl  $0, -4(%ebp)      /* is_eof == 0?                         */
        jne   2f
        jmp   1b
2: /* read_end */


3: /* print_start */
                                /* продивлятися список в зворотному
                                   порядку, так що в %edi адрес
                                   поточної структури                   */
        pushl 4(%edi)           /* вказівник на рядок з поточної
                                   структури                            */
        pushl $printf_format
        call  printf            /* вивести на екран                     */
        addl  $4, %esp          /* забрати із стеку тільки
                                   $printf_format                       */
        call  free              /* звільнити пам'ять, яку займає
                                   рядок                                */

        pushl %edi              /* вказівник на структуру для
                                   звільнення пам'яті                   */
        movl  (%edi), %edi      /* замінити вказівник в %edi на
                                   наступний                            */
        call  free              /* звільнити пам'ять, яку займає
                                   структура                            */

        addl  $8, %esp          /* забрати аргументи                    */

        cmpl  $0, %edi          /* адрес нової структури == NULL?    */
        je    4f
        jmp   3b
4: /* print_end */

        movl  $0, %eax          /* вийти з кодом 0                      */

return:
        movl  %ebp, %esp
        popl  %ebp
        ret
```

Примітка. Повідомлення клавіатурі про кінець файлу – Cttl-D

```
>gcc print.S -o print
```

```
>./print
aaa
bbbb
ccccc
^D<>
<ccccc>
<bbbb>
<aaa>
```

Введено 4 рядки: `"aaa"`, `"bbbb"`, `"ccccc"`, `""`.

В цій програмі використано деякий новий синтаксис. Є директива препроцесора `#define`. Препроцесор Сі (cpp) може бути використаний для обробки початкового коду на асемблері: для цього потрібно використати розширення `.S`. Файли з таким розширенням gcc попередньо обробляє препроцесором cpp, після чого він компілюється як звичайно.

У програмі використовуються позначки-числа, причому деякі з них повторюються в двох функціях. Можна використати звичайні позначки, але вони повинні бути унікальними в програмі. Наприклад, якщо б визначити позначку `read_start` в функції `read_str i` в функції `main`, то GCC видав би помилку при компіляції:

```
>gcc print.S
print.S: Assembler messages:
print.S:85: Error: symbol `read_start' is already defined
```

При використанні текстових позначок, потрібно використовувати унікальні імена. При використанні позначки-числа, компілятор перетворює їх в унікальні імена сам. Як позначка-число, використовується любе позитивне число як ім'я. Щоб зіслатися на вище розміщену позначку, використовується число з суфіксом **`b`** (мнемонічне значення – `backward`), а на нище розміщену позначку, використовується число з суфіксом **`f`** (`forward`).

Лабораторна робота № 10.

10. Модульне програмування

Мета роботи: вивчення модульного програмування.

1.1. Короткі теоретичні відомості

Невелика програма на асемблері звичайно поміщається в один файл. Однак програма може розміщуватися у декількох файлах. Об'єднання цих файлів в один звичайно виконується директивою `%include`. В результаті отримується одна одиниця трансляції, яку можна асемблювати за один раз. Такий підхід не можна застосувати до багатомодульної програми написаної на різних мовах програмування. При значній кількості модулів приходиться їх всіх транслювувати навіть без внесення змін у їх коди. Крім того, при використанні бібліотек налагоджених програм їх також приходиться кожний раз транслювати, що очевидно є недоцільним.

Усі ці проблеми дозволяє розв'язати розділена трансляція. Суть її в тому, що програма створюється як множина окремих підпрограм, кожна з яких транслюється окремо. Такі частини називаються одиницями трансляції або модулями. Кожний модуль транслюється окремо і в результаті трансляції отримують об'єктний файл з розширенням `.o`. Потім, за допомогою компонувача (редактора зв'язків), з набору об'єктних файлів отримують виконуваний файл.

Важливою особливістю модуля є наявність власного простору імен. Позначки, використані в модулі, видимі тільки всередині модуля. Для того, щоб зробити їх видимими в інших модулях використовується директива `global`.

Асемблер NASM підтримує такі поняття як глобальні і внутрішні позначки. Позначки оголошені директивою `global` відрізняються від звичайних тим, що інформація про них включається в об'єктний файл і стає доступною системному редактору зв'язків.

Для отримання доступу до позначок (імен програм або глобальних змінних) в інших модулях використовується директива `extern`. Наприклад, якщо потрібно звернутися до процедури `myproc` в іншому модулі, то потрібно написати `extern myproc`.

1.2. Створення багатомодульної програми

Приклад багатомодульної програми, яка запитує у користувача його ім'я і вітає його. Головна програма буде залежати від підпрограм виведення повідомлення `putstr` і зчитування імені `getstr`. Признаком кінця стрічки повідомлень прийнято нульовий байт. Підпрограма `putstr` використовує функцію `strlen` для визначення довжини повідомлення. Програма закінчується викликом підпрограми `quit`. Імена модулів прийняті як імена відповідних підпрограм `putstr.asm`, `getstr.asm`, `strlen.asm`, `quit.asm`. Системні виклики викликаються макросом `my_syscall`.

Модуль `putstr.asm`:

```
%include "my_syscall.inc"
global putstr
extern strlen
section .text
putstr:
    push ebp      ; стандартний початок підпрограми
    mov ebp,esp
    push dword [ebp+8]
    call strlen   ; виклик strlen, результат в eax
    add esp,4
    my_syscall 4,1,[ebp+8],eax   ; виклик write
```

```
    mov esp,ebp    ; стандартне завершення
    pop ebp        ; підпрограми
    ret
```

Модуль `getstr.asm`:

```
%include "my_syscall.inc"
global getstr
section .text
    ; [ebp+8]  - адреса буфера
    ; [ebp+12] - довжина буфера
getstr:
    push ebp       ; стандартний початок
    mov ebp,esp    ; підпрограми
    sub esp,4      ; місце під змінну
    xor eax,eax    ; eax=0
    mov [ebp-4],eax  ; i=0
.again:              ; початок головного циклу
    mov eax,[ebp+8]  ; занесення адреси в eax
    add eax,[ebp-4]  ; добавлення до неї istrlen, arg1
    my_syscall 3,0,eax,1  ; виклик read
    cmp eax,1          ; чи повернув виклик 1
    jne .endstr        ; ні - вийти з циклу
    mov eax,[ebp+8]    ; занесення адреси в eax
    add eax,[ebp-4]    ; добавлення до неї одиниці
    mov bl, [eax]      ; прочитаний байт в bl
    cmp bl,10          ; дорівнює 10?
    jne .noeol         ; ні, перехід
    dec dword [ebp-4]  ; так, зменшення i
    jmp .endstr        ; і вихід з циклу
.noeol:
    mov eax,[ebp-4]    ;завантаження i
    inc eax            ; i+1
    cmp eax,[ebp+12]   ; не перевищує arg2?
    jae .endstr        ; так, вихід з циклу
    inc dword [ebp-4]  ; i+1
    jmp .again         ; продовження циклу
.endstr:
    mov eax,[ebp+8]    ; завантаження адреси в eax
    add eax,[ebp-4]    ; збільшення на I
    inc eax            ; збільшення на 1
    xor bl,bl          ; обнулення bl
    mov [eax],bl       ; занесення 0 в кінець стрічки
    mov esp,ebp        ; стандартний вихід з
    pop ebp            ; підпрограми
    ret
```

Модуль `strlen.asm`:

```
global strlen
strlen:         ; addr of string [ebp+8]
  push ebp
  mov ebp,esp
  push esi
  xor eax,eax
  mov esi,[ebp+8]
.lp:
  cmp byte [esi],0
  jz .quit
  inc esi
  inc eax
  jmp short .lp
.quit:
  pop esi
```

```
  pop ebp
  ret
```

Модуль `quit.asm`:

```
%include "student.inc"
global quit
section .text
quit: my_syscall 1,0
```

Основний модуль `main.asm`:

```
global _start            ; головний модуль
extern putstr
extern getstr
extern quit
section .data
msg1 db 'Привіт, як тебе звати',10,0
msg2 db 'Приємно познайомитися з тобою',10,0
msg3 db '!',10,0
section .bss      ; пам'ять під буфер
buf resb 512
buflen equ $-buf
section .text
_start:
    push dword msg1
    call putstr      ; виведення привітання
    add esp,4
    push dword buflen
    push dword buf
    call getstr      ; виклик getstr з параметрами buf, buflen
    add esp,8
    push dword msg2  ; виклик putstr для msg2
    call putstr
    add esp,4
    push dword buf   ; виклик putstr для
    call putstr      ; стрічки введеної
    add esp,4        ; користувачем
    push dword msg3  ; виклик putstr для
    call putstr      ; msg3
    add esp,4
    call quit        ; виклик quit
```

Окреме асемблювання модулів:

```
$ nasm -f elf32 getstr.asm -o getstr.o
$ nasm -f elf32 putstr.asm -o putstr.o
$ nasm -f elf32 strlen.asm -o strlen.o
$ nasm -f elf32 guit.asm -o guit.o
$ nasm -f elf32 main.asm -o main.o
```

Компонування модулів у виконуваний файл `main`:

```
ld -m elf_i386 -Map $1.map getstr.o putstr.o strlen.o quit.o main.o -o main
```

Запуск на виконання:

```
$ ./main
Привіт, як тебе звати
Петро
Приємно познайомитися з тобою
Петро!
```

1.3. Бібліотека об’єктних модулів

Об’єктні модулі можна помістити в бібліотеку для подальшого використання. Створюється така бібліотека програмою **ar**. Довідка для роботи з програмою:

```
$ ar –h
```

Створення бібліотеки `mylib.a`:

```
$ ar crs mylib.a getstr.o putstr.o strlen.o quit.o –target=elf32-i386
```

Виведення вмісту бібліотеки:

```
$ ar t mylib.a
getstr.o
putstr.o
strlen.o
quit.o
```

Після цього можна скомпонувати основну програму з використанням бібліотеки `mylib.a` за допомогою системного редактора зв’язків:

```
$ ld –m elf_i386 main.o mylib.a –o main
```

Запитання.

1. В чому суть модульного програмування.
2. Що оголошують директиви `global` і `extern`.
3. Послідовність створення багатомодульної програми.
4. Для чого використовуються бібліотеки об’єктних модулів.
5. Створення і використання бібліотеки об’єктних модулів.

Завдання.

1. Написати програму друку параметрів консольного рядка на основі модульного програмування.

2. Написати програму, яка виконує додавання, віднімання, множення і ділення двох цілих чисел на основі модульного програмування. Числа вводити з консолі. При компонуванні головної програми використати бібліотеку об’єктних модулів.

2. Приклади для самостійної роботи

1. readt.c - тестування 32-бітової асемблерної функції read_doubles()
яка читає числа doubles з stdin *

```
; read.asm – читання масиву doubles з файлу
segment .data
format  db      "%lf", 0        ; format for fscanf()
segment .bss
segment .text
        global  read_doubles
        extern  fscanf
%define SIZEOF_DOUBLE   8
%define FP              dword [ebp + 8]
%define ARRAYP          dword [ebp + 12]
%define ARRAY_SIZE      dword [ebp + 16]
%define TEMP_DOUBLE     [ebp - 8]
;
; function read_doubles
; C прототип: int read_doubles( FILE * fp, double * arrayp, int array_size );
; EOF або масив заповнений
; Параметри:
;   fp         - вказівник на FILE
;   arrayp     - вказівник на масив double
;   array_size – число елементів у масиві
; Значення повернення:
;   число doubles записаних у масив (повертається в EAX)
```

```
read_doubles:
        push    ebp
        mov     ebp,esp
        sub     esp, SIZEOF_DOUBLE      ; визначення одного double у стеку
        push    esi                     ; збереження esi
        mov     esi, ARRAYP             ; esi = ARRAYP
        xor     edx, edx                ; edx = індекс масиву (початково 0)
while_loop:
        cmp     edx, ARRAY_SIZE         ;  edx < ARRAY_SIZE?
        jnl     short quit              ; якщо ні, quit loop

; виклик fscanf() для читання double у TEMP_DOUBLE
; fscanf() може змінити edx так що збережемо його
        push    edx                     ; збереження edx
        lea     eax, TEMP_DOUBLE
        push    eax                     ; push &TEMP_DOUBLE
        push    dword format            ; push &format
        push    FP                      ; push file pointer
        call    fscanf
        add     esp, 12
        pop     edx                     ; відновлення edx
        cmp     eax, 1                  ; чи fscanf повернув 1?
        jne     short quit              ; якщо ні, quit loop

; копіювання TEMP_DOUBLE у ARRAYP[edx]
; ( 8-байт double копіюється у два 4-байти)
        mov     eax, [ebp - 8]
        mov     [esi + 8*edx], eax      ; спершу копіюється молодші 4 байти
        mov     eax, [ebp - 4]
        mov     [esi + 8*edx + 4], eax  ; наступними копіюються старші 4 байти
        inc     edx
        jmp     while_loop
quit:
        pop     esi                     ; відновлення esi
        mov     eax, edx                ; збереження return значення у eax
        mov     esp, ebp
        pop     ebp
        ret
```

$ nasm -f elf read.asm -o read.o

```
/* readt.c */
#include <stdio.h>
#include "cdecl.h"
extern int PRE_CDECL read_doubles( FILE *, double *, int ) POST_CDECL;

#define MAX 100
int main() {
  int i,n;
  double a[MAX];
  n = read_doubles(stdin, a, MAX);
  for( i=0; i < n; i++ )
    printf("%3d %g\n", i, a[i]);
  return 0;
}
```

```
$ gcc readt.c read.o -o readt
$ ./readt
1.234
123.456
12345.6789
  0 1.234
  1 123.456
```

```
2 12345.7
```

Список використаних джерел

1. Столяров А.В. Программирование на языке асемблера NASM для ОС UNIX. Уч. пособие. – 2-е изд. – М.: Макс-пресс, 2011. – 188 с.

2. Аблязов Р.З. Программирование на ассемблера на платформе x86-64. – М.: ДМК Пресс, 2011. – 304 с.

3. Магда Ю.С. Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006. — 410 с.

4. Харт Джонсон М. Системное программирование в среде Windows / Джонсон М. Харт ; пер. с англ. — М. : Издательский дом «Вильямс», 2005.

5. Кип Р. Ирвин. Язык ассемблера для процесоров Intel. – 4-е изд., Пер. с англ. – М.: Издательский дом ”Вильямс”, 2005. – 912 с.: ил.

6. Юров В.И. Assembler. Учебник для вузов. 2-е изд. – СПб.: Питер, 2003. – 637 с.

7. Голубь Н.Г. Исскуство программирования на ассемблере. Лекции и упражнения. – 2-е изд., испр. и доп., – СПб.: ДиаСофт, 2002. – 656 с.: ил.

8. Електронний ресурс: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.