МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
«Прикарпатський національний університет
імені Василя Стефаника»

Фізико-технічний факультет

Голота В. І.

“СИСТЕМНЕ ПРОГРАМУВАННЯ. АСЕМБЛЕР x86-64”

Методичні вказівки до виконання лабораторних робіт

Електронне мережеве навчальне видання

Івано-Франківськ, 2024

УДК 004.432.2
Г 61

Рекомендовано до друку Вченою радою фізико-технічного факультету Прикарпатського національного університету імені Василя Стефаника (протокол № 2 від 24 жовтня 2024 року)

Рецензенти:

Когут Ігор Тимофійович, завідувач кафедри комп'ютерної інженерії та електроніки Прикарпатського національного університету імені Василя Стефаника, професор, доктор технічних наук

Никируй Любомир Іванович, завідувач кафедри фізики і хімії твердого тіла Прикарпатського національного університету імені Василя Стефаника, професор, кандидат фізико-математичних наук

Методичні вказівки до виконання лабораторних робіт з дисципліни "Системне програмування. Асемблер x86-64. [Електронний ресурс] / Прикарпатський національний університет імені Василя Стефаника; уклад. Голота В. І. – Електронні текстові дані (1 файл: 8 Мбайт). – Івано-Франківськ: Прикарпатський національний університет імені Василя Стефаника, 2024. – 208 с. – Назва з екрану.

Електронне мережеве навчальне видання

У методичних вказівках викладено теоретичні відомості та практичні завдання для виконання лабораторних робіт з дисципліни "Системне програмування". Методичні вказівки розроблено з метою вивчення програмної моделі, типів даних, способів адресації, базових команд мікропроцесорів з архітектурою x86-64 та команд розширень MMX, SSE, AVX. У методичних вказівках для розроблення програм використовується ОС Linux і Nasm асемблер.

Лабораторні роботи містять приклади програмування, запитання та практичні завдання. Методичні вказівки призначені для здобувачів ступеня бакалавра галузі знань 12 "Інформаційні технології" спеціальності 123 "Комп'ютерна інженерія".

УДК 004.432.2

ЗМІСТ

Лабораторна робота 1

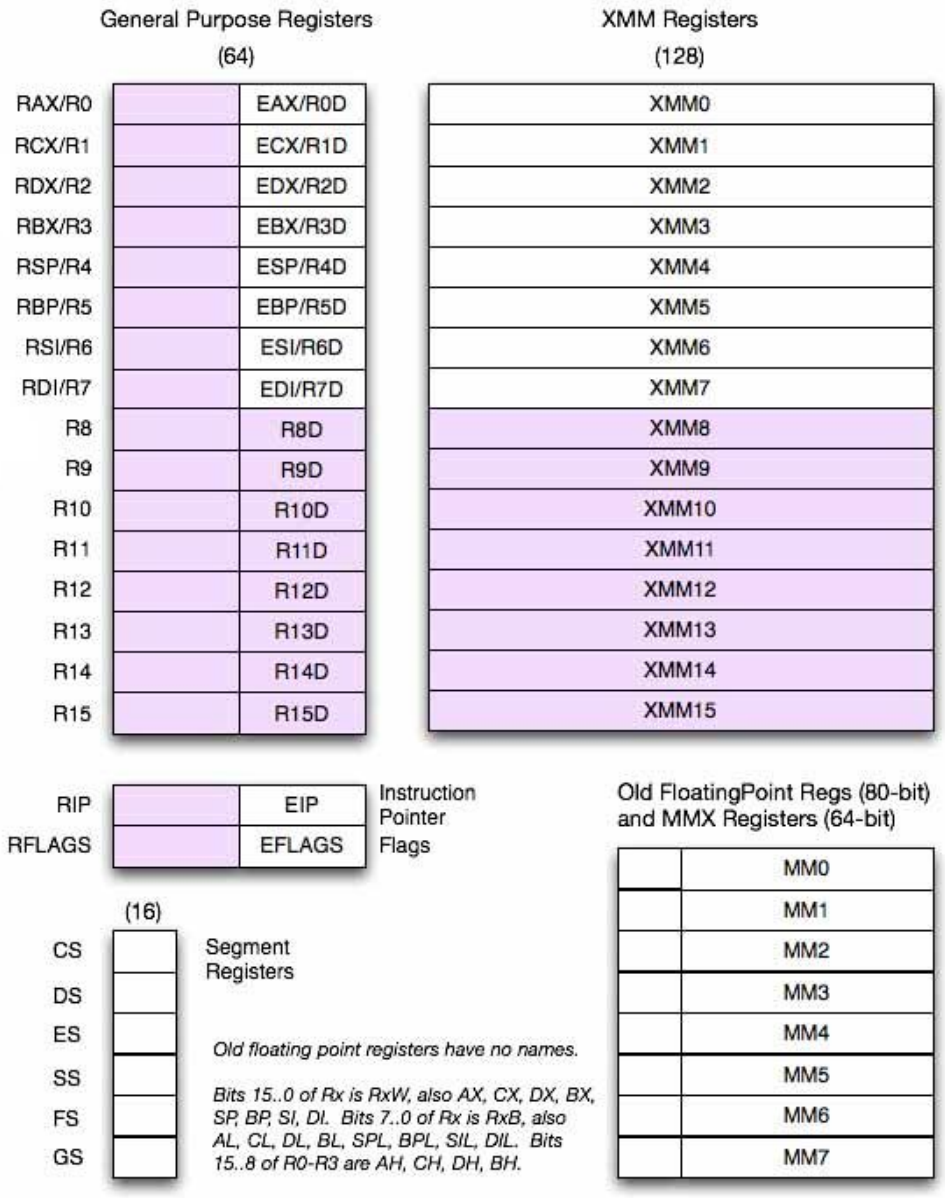
Програмна модель МП x86-64

Мета роботи: вивчення програмної моделі мікропроцесорів IA-32, x86-64 та синтаксису асемблера Nasm.

Теоретичні відомості

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 (64-біт) мають наступні набори регістрів:



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

2. Регістри загального призначення

В МП Intel x86_64 є 16 регістрів загального призначення, які використовуються для зберігання:

- операндів арифметичних і логічних виразів;
- компонентів адрес;
- вказівників на комірки пам’яті.

Регістри загального призначення можуть використовуватися як 64-, 32-, 16- і 8-бітові регістри.

64-біти	32-біти	16-бітів	Молодші 8-біт	Старші 8-біт
rax	eax	ax	al	ah
rbx	ebx	bx	bl	bh
rcx	ecx	cx	cl	ch
rdx	edx	dx	dl	dh
rsi	esi	si	sil	-
rdi	edi	di	dil	-
rbp	ebp	bp	bpl	-
rsp	esp	sp	spl	-
r8	r8d	r8w	r8b	-
r9	r9d	r9w	r9b	-
r10	r10d	r10w	r10b	-
r11	r11d	r11w	r11b	-
r12	r12d	r12w	r12b	-
r13	r13d	r13w	r13b	-
r14	r14d	r14w	r14b	-
r15	r15d	r14w	r15b	-

Позначення перших 8-ти регістрів історично визначилося їх призначенням:

Регістри даних **rax** (Accumulator register) – акумулятор, використовується для зберігання даних проміжних обчислень;	Регістри вказівники для роботи зі стеком **rsp** (Stack pointer register) – регістр вказівник стеку. Містить адрес верхівки стеку.
rbx (Base register) – базовий регістр, використовується для зберігання базової адреси деякого об’єкта в пам’яті;	**rbp** (Base pointer register) – регістр вказівник бази кадру стека. Призначений для організації довільного доступу до даних всередині стеку.
rcx (Counter register) – регістр лічильник, неявно використовується в деяких командах для організації циклів;	Індексні регістри для підтримки ланцюжкових операцій **rsi** (Source index register) – індекс джерела, в ланцюгових командах містить поточну адресу елемента джерела;

rdx (Data register) – регістр даних, використовується для зберігання результатів проміжних обчислень і введення-виведення;	**rdi (**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 однобайтові половини, до яких можна звернутися як до `ah` (старшого), `al` (молодшого) байту.

Інструкції, в яких використовуються регістри з префіксом “`r`” дають доступ до байтових регістрів.

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

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

- **cf** (*Carry flag*) – прапор перенесення/позики у беззнаковій цілочисельній арифметиці:

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** (*adjust flag, 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 адрес до молодших;

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

• **of** (*Overflow flag*) – прапор втрати значущого біта у знаковій цілочисельній арифметиці (у доповняльному коді):

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-бітову 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 мегабітовими сторінками (PSE);

• 5-біт, включає 36-бітовий 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 rax,0200  ; десяткове
mov rax,0200d ; явне десяткове
mov rax,0d200 ; явне десяткове
mov rax,0c8h  ; шістнадцяткове
mov rax,$0c8  ; шістнадцяткове: перед буквою потрібний 0
mov rax,0xc8  ; шістнадцяткове
mov rax,0hc8  ; шістнадцяткове
mov rax,310q  ; вісімкове
mov rax,310o  ; вісімкове
mov rax,0o310 ; вісімкове
mov rax,0q310 ; вісімкове
mov rax,11001000b   ; двійкове
mov rax,1100_1000b  ; двійкове
mov rax,1100_1000y  ; двійкове
mov rax,0b1100_1000 ; двійкове
```

```
mov rax,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 ліво нахилені апострофи (`…`).

У стрічки, з ліво нахиленими лапками, можна поміщати esc-послідовності у С-стилі:

```
\’ одинарні лапки (’)
\" подвійні лапки (")
\‘ одинарні ліво нахилені лапки (‘)
\\ 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 чотирна 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 транслюється 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

Асемблер YASM підтримує синтаксис асемблерів GAS, GNU, NASM, TASM.

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, чотирне 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-байтова 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 розглядаються 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,

чотирне 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  1   ; 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 адрес 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 псевдоінструкція `.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-бітовому 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. Місце знаходження 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 rax, value+2 ; скопіювати адресу value+2 у регістр rax
mov rax, 45h    ; скопіювати безпосереднє значення 45h у регістр rax
```

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  rax, rcx     ; скопіювати значення з регістра rcx в регістр rax
mov rdx, tax_rate ; скопіювати адресу комірки пам'яті у регістр
mov count, rcx    ; 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 rax,addr1   ; скопіювати в регістр eax адресу addr1
add rax,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 rax,[num]     ; завантажити в регістр eax значення за адресою num
mov rbx,[num+2]   ; nasm - завантажити в регістр ebx значення,
                  ; яке знаходиться за адресою num+2
mov rcx,[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-бітове 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`.

Контрольні запитання.

1. Програмна модель МП 80x86 і x86-64.
2. Регістри загального призначення. Сегментні регістри.
3. Регістри спеціального призначення `rip, rflags`.
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`.

Практична частина

Завдання.

1. Записати ціле число 123 у 2-, 8-, 10- і 16-системі числення з використання суфіксів і префіксів.

2. Зарезервувати пам'ять для цілих чисел 1, 123, 512, 64000, 256000.

3. Зарезервувати пам'ять для чисел з плаваючою крапкою 1.23, -25y5, 126e-9, 123456.789101112.

4. Зарезервувати пам'ять для BCD констант 12345678, -87654321.

5. Показати порядок розміщення батів little-endian і big-endian для чисел
m1 db 0x12, 0x34, 0x56

6. Показати порядок розміщення батів little-endian і big-endian для чисел
m1 dw 0x12, 0x34, 0x56

7. Показати порядок розміщення батів little-endian і big-endian для чисел
m1 dd 0x12, 0x34, 0x56

8. Показати порядок розміщення батів little-endian і big-endian для чисел
m1 dq 0x12, 0x34, 0x56

9. Показати порядок розміщення батів little-endian і big-endian для чисел
m1 dt 0x12, 0x34, 0x56

10. Показати порядок розміщення батів little-endian і big-endian для чисел
m1 do 0x12, 0x34, 0x56

11. Зарезервувати 1000 комірок пам'яті із значенням 1.

12. Зарезервувати 1000 комірок пам'яті із значенням 1024.

13.Скопіювати командами область пам’яті `DATA1` типу `byte` із значеннями 1,2,3 у регістри ax, bx, cx.

14. Скопіювати командами область пам’яті `DATA1` типу `word` із значеннями 111, 222, 333 у регістри eax, ebx, ecx.

15. Скопіювати значення з комірки пам'яті з адресою mem1 у регістр rax.

16. Скопіювати значення з регістра al у комірку пам'яті з адресою mem1.

17. Скопіювати значення з регістра ah у комірку пам'яті з адресою mem1.

18. Скопіювати значення з першої комірки пам'яті у сегменті даних CS у регістр eax.

19. Скопіювати значення з четвертої комірки пам'яті у сегменті даних DS у регістр ebx.

20. Записати значення з регістра eax у останню комірку сегменту даних ES.

Лабораторна робота 2

Асемблери NASM і GAS

Мета роботи: вивчення синтаксису і форматів команд асемблерів NASM і GAS, отримання навичок асемблювання, компонування і налагодження програм асемблера.

Теоретичні відомості

1. Асемблери

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
  * bin       flat-form binary files (e.g. DOS .COM, .SYS)
    ith       Intel hex
    srec      Motorola S-records
    aout      Linux a.out object files
    aoutb     NetBSD/FreeBSD a.out object files
    coff      COFF (i386) object files (e.g. DJGPP for DOS)
    elf32     ELF32 (i386) object files (e.g. Linux)
    elf64     ELF64 (x86_64) object files (e.g. Linux)
    elfx32    ELFX32 (x86_64) object files (e.g. Linux)
    as86      Linux as86 (bin86 version 0.3) object files
    obj       MS-DOS 16-bit/32-bit OMF object files
    win32     Microsoft Win32 (i386) object files
    win64     Microsoft Win64 (x86-64) object files
    rdf       Relocatable Dynamic Object File Format v2.0
    ieee      IEEE-695 (LADsoft variant) object file format
    macho32   NeXTstep/OpenStep/Rhapsody/Darwin/MacOS X (i386) object files
    macho64   NeXTstep/OpenStep/Rhapsody/Darwin/MacOS X (x86_64) object files
    dbg       Trace of all info passed to output stage
    elf       ELF (short name for ELF32)
    macho     MACHO (short name for MACHO32)
    win       WIN (short name for WIN32)
```

1.1. Асемблювання і компонування програм на асемблері

Асемблювати програми для x86, x86-64 архітектур можна за допомогою різних асемблерів NASM, GAS (GNU AS є back-end компілятора gcc), YASM, 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 elf32 program.asm –o program.o (64-розрядна ОС, об’єктний файл 32-роз.)
$nasm -f elf64 program.asm -o program.o (64-розрядна ОС, об’єктний файл 64-роз.)
$nasm –f elf64 –g –F dwarf program.asm ( -g вставити налагоджувальну інформацію
для налагоджувача, -F dwarf 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 налагоджувальної інформації

```
$nasm -f elf32 Debugging With Arbitrary Record Formats 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

$nasm -f elf64 -y
valid debug formats for 'elf64' output format are ('*' denotes default):
    dwarf     ELF64 (x86-64) dwarf debug format for Linux/Unix
    stabs     ELF64 (x86-64) stabs debug format for Linux/Unix
```

Формат stabs для налагоджувальної інформації в основному є великим масивом записів фіксованого розміру. Він має один перелік, кілька невеликих цілих аргументів і один рядок. Отже, коли додаються розширення, рядок стає усе складнішим і складнішим. Приклад:

```
.stabs "__1cEtfex3CTACTB_6Fp10_i_:YTfA:tYC(0,19);B:tYC(0,20)
;@;;__1cEtfex3CTACTB_6Fp10_i_:T(0,3);(0,21)=
*(0,20);(0,19);@;x:p(0,21);y:p(0,19);@;1;1;",N_GSYM,0x0,0x0,0x0
```

Формат `dwarf` (Debugging With Arbitrary Record Formats) більш структурований і не залежить від формату об’єктного файлу. Результат роботи `dwarf` можна побачити за допомогою команди `dwarfdump`. Ось приклад виводу dwarfdump:

```
$ dwarfdump -a a.out
...
<1><  405>      DW_TAG_base_type
                DW_AT_name                  int
                DW_AT_encoding              DW_ATE_signed
                DW_AT_byte_size             4
...
```

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 за допомогою ld

```
$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.)
$ld -m elf_x86_64 program.o -o program (64-розрядна ОС, виконуваний файл 64-роз.)
     де program.o – об’єктний файл
     program – виконуваний (бінарний) файл формату ELF
```

Компонування програм за допомогою gcc

```
$gcc -o program program.o -no-pie
```

Останні версії компонувальника і компілятора GCC створюють незалежні від позиції виконувані файли (PIE) за замовчуванням. Це зроблено для того, щоб хакери не досліджували використання пам'яті програмою. у та, зрештою, втручання у виконання програми. Для навчальної мети немає необхідності створювати позиційно-незалежні виконувані файли, тому в `gcc` додається параметр `-no-pie`.

Запуск програми на виконання

```
$./program
```

Bash сценарій отримання бінарного файлу для налагоджувача GDB

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

```
nasm –f elf64 –g –F dwarf –l $1.lst $1.asm –o $1.o
ld –m elf_x86_64 –Map –$1.map $1.o –o $1
```

GNU утиліта make, файл makefile

GNU утиліта `make` керує трансляцією і компонуванням групи програм (проектів). Завдання на виконання `make` записується у текстовому файлі `makefile` (розміщується в поточному каталозі). `Makefile` може містити:

коментарії;
правила;
макровизначення.

Приклад:

```
build:
    nasm -f elf32 –g –F dwarf -l main.lst main.asm
    ld -Map main.map -o main main.o
clean:
    rm -f main main.o main.lst main.map
    # тут можуть бути додаткові команди
```

У прикладі створено два правила з іменами («цілями») `build i clean`. Перше правило виконує асемблювання і компонування програми, а друге вилучає всі файли створені першим правилом. Якщо набрати в командному рядку `make build`, то буде створено виконуваний файл. Якщо набрати `make clean`, то проект буде очищений.

Примітка. У файлі `makefile` команди правил мають починатися з відступу, який обов'язково має створюватися символом табуляції (Tab), а не символами пропуску (у редакторі `mc` подвійним натискання Tab).

Приклад `makefile` із суфіксами:

```
#Записати наступні рядки з іменем 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)
```

Більш повну інформацію про утиліту `man` можна отримати в `man make`.

Сценарій асемблювання і компонування

Асемблювання і компонування програм зручно виконувати запуском одного Bash сценарію:

```
#------------------------------------------------------------
# Сценарій асемблювання і компонування програм Nasm асемблера
# Copyright (C) Голота В.І., 2020
#------------------------------------------------------------
#!/bin/bash
rm *.o *.map *.lst
echo "------------------------------------------------------"
echo "Сценарій: $0      Дата-Час: `date +%d.%m.%y-%T`"
echo "------------------------------------------------------"

declare -i code

if [ ! -f $1 ]
then
   echo "Файл $1 відсутній"
   exit 1
fi

s=$1
name=${s%.*m}
nasm -f elf64 -g -F dwarf  $1 -l $name".lst" -o $name".o"
code=$?
if [ $code -eq 0 ]
then
    echo "Асемблювання $1 успішне - $code"
else
    echo "Асемблювання $1 не успішне - $code"
    exit $code
fi

#ld -m elf_x86_64 -M -Map $name".map" $name".o" -o $name
gcc -o $name  $name".o" -no-pie
code=$?
if [ $code -eq  0 ]
then
   echo "Компонування $1 успішне - $code"
   rm $name".o"
else
   echo "Компонування $1 не успішне - $code"
   exit $code
fi
```

Сценарій записати під іменем `asm_ld.sh` у каталог де знаходяться асемблерні програми. Сценарій зробити виконуваним

```
$ chmod u+x asm_ld.sh
```

Асемблювання і запуск програм на виконання:

```
$ ./asm_ld.sh program.asm
```

Налагоджувач GDB

Інсталяція GDB в ОС Ubuntu:

```
$ sudo apt install gdb
```

Для виявлення логічних помилок у виконуваному файлі асемблера використовується консольний GDB або графічний налагоджувач (`debugger`). Запуск консольного налагоджувача на виконання `gdb <program>`. Довідка про налагоджувач `gdb --help`, довідка для роботи з 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 main
main:
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  rax,60    ; номер системного виклику
mov  rdi,code ; код виходу для системного виклику
syscall    ; 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 elf64 -g -F dwarf $1
```

Запуск на виконання

```
$./debug.sh 1.asm
```

Сценарій компонування (`load.sh`):

```
#!/bin/bash
ld -m elf_x86_64 $1 -o test
```

Запуск на виконання

```
$./load.sh 1.o
```

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

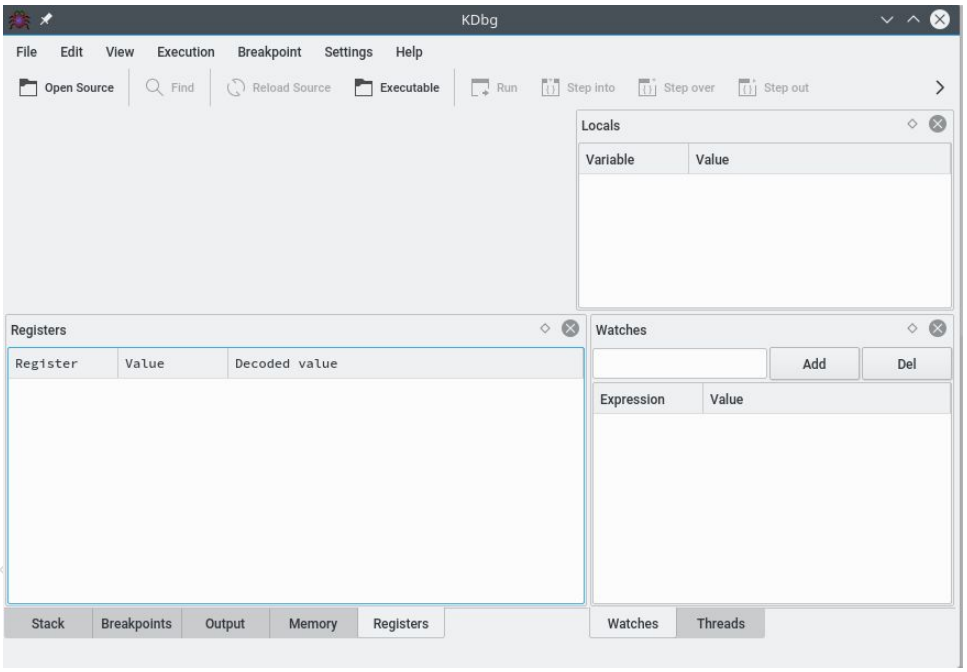


Рисунок 1 – Графічне середовище налагоджувача KDbg

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**.

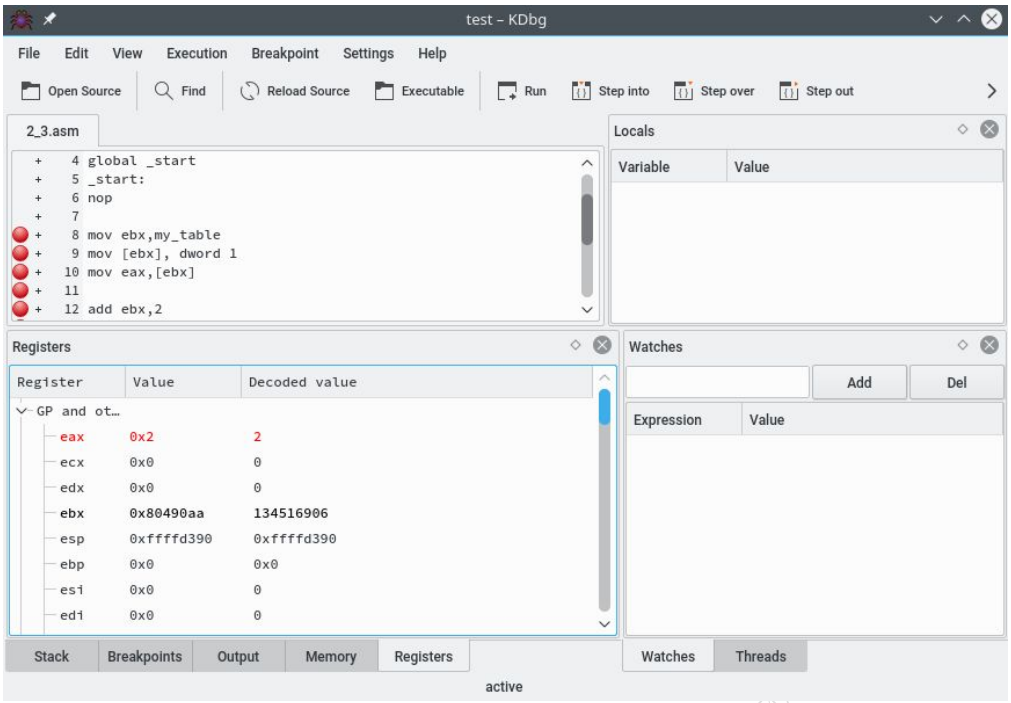


Рисунок 2 – Налагодження програми

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.

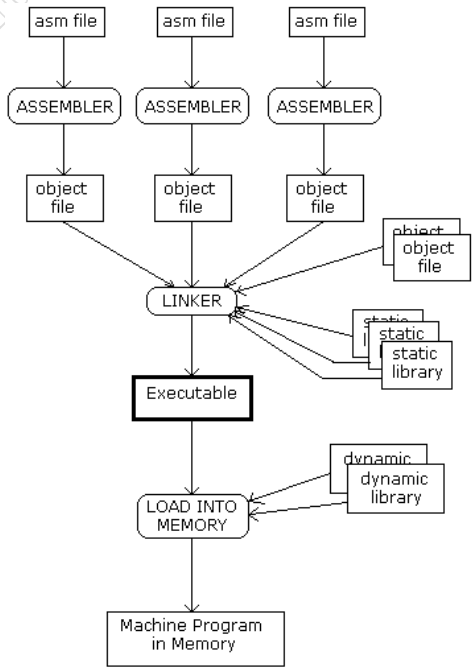


Рисунок 3 – Послідовність створення виконуваних файлів асемблера

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,	невідомо	movl 10, %ecx

	[10]		
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 gas.

Переслати значення за адресою 8 в регістр %eax : `movl 8, %eax (mov eax, [8]).`

Переслати значення 8 в регістр %eax : `movl $8, %eax (move ax, 8)`

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

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

Додати 4 до значення вказівника верхівки стеку і значення по цьому адресу переслати в регістр %eax : `movl 4(%esp), %eax ( у NASM: move eax,[4+esp]).`

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

2.3. Синтаксис і формати операндів в командах GAS асемблера

Кожна 32-бітова інструкція має своє 64-бітове розширення, яке вказується суфіксом. Для запису 32-бітових інструкцій використовують суфікс "l", а 64-бітових – "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 байтова інструкція
movq 1, %rax             - 7-ми байтова інструкція
movl 0xffff ffff, %eax  - 5-ти байтова інструкція
movq 0xffff ffff, %rax   - 10-ти байтова інструкція
```

Можна використовувати символьні вирази в інструкціях:

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

Якщо символьний вираз займає 32-розряди то можна використати 32-бітові інструкції. Якщо потрібно завантажити символьний вираз як 64-бітове 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 адресою символу ”symbol”.
```

2.4. Команди асемблера NASM

Команда додавання add

```
add destreg, constant     ;destreg := destreg + constant
                          ;destreg, регістр ЗП n=8, 16, 32, 64 біт
                          ; constant, n=8, 16, 32

add destmem, constant     ;destmem := destmem + constant
```

```
                          ;destmem, комірка пам’яті розміром n=8, 16, 32, 64
                          ; constant, n=8, 16, 32

add destreg, srcreg       ;destreg := destreg + srcreg
                          ;destreg і srcreg, регістр ЗП n=8, 16, 32, 64 біт
                          ;регістри мають бути одного розміру

add destreg, [srcmem]     ;destreg := destreg + srcmem
                          ;destreg, регістр ЗП n=8, 16, 32, 64 біт
                          ;srcmem, комірка пам'яті такого розміру як регістр

add [destmem], srcreg     ;destmem := destmem + srcreg
                          ;src, регістр ЗП n=8, 16, 32, 64 біт
                          ;destmem, комірка пам'яті такого розміру як регістр
```

Команда логічного and

```
and destreg, constant     ;destreg := destreg AND constant
                          ;destreg, регістри ЗП n=8, 16, 32, 64 біт
                          ;constant, n=8, 16, 32

and [destmem], type constant ;destmem := destmem AND constant
                             ;destmem,  комірка пам'яті n=8, 16, 32, 64
                             ;type := byte, word, dword

and destreg, srcreg      ;destreg := destreg AND srcreg
                         ;destreg і srcreg, регістри n=8, 16, 32, 64 біт
                         ;Регістри мають бути одного розміру

and destreg, [srcmem]    ;destreg := destreg AND srcmem
                         ;destreg, регістри ЗП n=8, 16, 32, 64 біт
                         ;srcmem,  комірка пам'яті такого ж розміру

and [destmem], srcreg    ;destmem := destmem AND srcreg
                         ;srcreg, регістр ЗП n=8, 16, 32, 64 біт
                         ;destmem,  комірка пам'яті такого ж розміру
```

Команда виклику підпрограм call

```
call label  ;виклик підпрограми, яка задана позначкою (label) в програмі
call reg    ;виклик підпрограми за адресою заданою в регістрі
            ;reg ЗП n=8, 16, 32, 64 біт
call mem    ;виклик підпрограми за адресою заданою в комірці
            ;пам'яті mem n=8, 16, 32, 64 біт
```

Команди чистки прапорів cli, cld, clc, cmc, clac,

```
cli  ; чистити прапор interrupt
cld  ; чистити прапор direction
clc  ; чистити прапор carry
cmc  ; інвертувати прапора carry
clac ; чистити прапор alignment
```

Команда порівняння cmp

```
cmp regn, constant    ;порівняти reg з константою
                      ;regn, регістр ЗП n=8, 16, 32, 64 біт
                      ; constant, n=8, 16, 32

cmp [mem], constant   ;порівняти комірку пам’яті mem з константою
                      ;mem, комірка пам'яті n=8, 16, 32, 64 біт
                      ; constant, n=8, 16, 32

cmp leftreg, rightreg ;порівняти регістри leftreg і rightreg .
                      ;leftreg і rightreg регістр n=8, 16, 32, 64 біт
                      ;регістри мають бути одного розміру

cmp reg, [mem]      ;порівняти reg і mem
                    ;reg, регістр ЗП n=8, 16, 32, 64 біт
                    ;mem, комірка пам'яті розміром n=8, 16, 32 біт

cmp [mem], reg      ;порівняти mem і reg
                    ;reg, регістр ЗП n=8, 16, 32, 64 біт
                    ;mem, комірка пам'яті розміром n=8, 16, 32, 64 біт
```

Команда інкременту dec

```
dec reg             ;reg := reg - 1
                    ;regn, регістр ЗП n=8, 16, 32, 64 біт

dec type [mem]      ;mem := mem - 1
                    ;memn, комірка пам’яті розміром n=8, 16, 32, 64 біт
                    ;type : = byte, word, dword, qword
```

Команди ділення беззнакового div і знакового idiv

```
div reg8            ;al := ax div reg8   - частка
                    ;ah := ax mod reg8   - залишок
                    ;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 reg64           ;rax := rdx:rax div reg64
                    ;rdx := rdx:rax mod reg64
                    ;reg64, регістр ЗП 64-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

div qword [mem64]   ;rax := rdx:rax div mem64
                    ;rdx := rdx:rax mod mem64
                    ;mem64, комірка пам'яті 64-біт
```

Команда налаштування стеку при вході в процедуру enter

```
enter imm16, 0  ; imm16 – безпосереднє значення
                ;push ebp
                ;sub esp, imm16
```

Команда інкременту inc

```
dec reg         ;reg := reg + 1
                ;regn, регістр ЗП n=8, 16, 32, 64 біт

dec [type] mem  ;mem := mem + 1
                ;memn, комірка пам’яті розміром n=8, 16, 32, 64 біт
                ;type=byte, word, dword, qword
```

Команди 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
```

Команда безумовного переходу jmp

```
jmp label ;перехід на позначку label
jmp reg   ;перехід на адресу з регістра reg n=8,16,32,64
jmp [reg] ;перехід на адресу, прочитаної з пам'яті, адреса якої міститься в
          ;регістрі reg ЗП n=8,16,32, 64
jmp [mem] ;перехід на адресу, прочитаної з пам'яті, адреса якої міститься в
          ; комірці mem 8,16,32,64
```

Команда завантаження ефективної адреси lea

```
lea reg, [mem]  ;reg - регістр ЗП 8, 16, 32, 64 біт
                ;mem, комірка пам'яті того ж розміру як регістр
```

Команда налаштування стеку при виході з підпрограми leave

```
leave  ; mov rsp, rbp
       ; pop rbp
```

Команда копіювання даних mov

```
mov destreg, constant          ;destreg, регістр ЗП n=8, 16, 32, 64 біт
                               ;constant, n=8, 16, 32, 64
mov type [destmem], constant   ;destmem, комірка пам'яті n=8,16, 32, 64 біти
                               ;type=byte, word, dword, qword
                               ;constant, n=8,16,32
mov destreg, srcreg            ;destreg := srcreg
                               ;destreg і srcreg,  регістр ЗП n=8, 16, 32, 64
біт
                               ;регістри мають бути одного розміру
mov destreg, [srcmem]          ;destreg := srcmem
                               ;destreg, регістр ЗП n=8, 16, 32, 64 біт
                               ;srcmemn, комірка пам'яті того ж розміру
mov destmem, [srcreg]          ;desttmem := srcreg
                               ;srcreg, регістр ЗП n=8, 16, 32 біт, 64
                               ;destmemn, комірка пам'яті того ж розміру
```

Команда копіювання даних заданих розмірів 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,
movsq           ; [rdi] := [rsi], rdi+=8, rsi+=8

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 в рег. ecx
```

```
rep movsw
rep movsd
rep movsq
```

Команда беззнакового mul і знакового множення imul

```
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 reg64        ;rdx:rax := rax * reg64
                 ;reg64, регістр ЗП 64-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 dword [mem32]  ;edx:eax := eax * mem32
                   ;mem32, комірка пам'яті 32-біт
mul qword [mem64]  ;edx:eax := eax * mem64
                   ;mem64, комірка пам'яті 64-біт
```

Команда зміни знаку (доповнення до 2) neg

```
neg reg        ;reg := -reg
               ;reg, регістр загального призначення n=8, 16, 32, 64-біт

neg type [mem] ; mem := -memn
               ; mem, комірка пам'яті розміром n=8, 16, 32, 64-біт,
               ; type=byte,word,dword,qword
```

Команда інвертування бітів (доповнення значення до 1) not

```
not reg        ;reg := not reg
               ;reg, регістр загального призначення n=8, 16, 32, 64-біт

not type [mem] ;mem := not memn
               ;mem, комірка пам'яті розміром n=8, 16, 32, 64-біт,
               ; type=byte,word,dword,qword
```

Помістити дані у стек

```
push type constant   ;помістити константу 8,16,32, 64-біт в стек
push srcreg          ;помістити регістр ЗП 8,6,32, 64-біт в стек
push type [srcmem]   ;помістити 8,16,32, 64-бітову комірку пам'яті в стек
pushf                ;помістити молодші 16-бітів регістра EFLAGS в стек
(i386)
pushfd               ;помістити копію регістра EFLAGS в стек (i386)
pushfq               ;помістити копію регістра RFLAGS в стек (x86_64)
push reg_es | reg_cs | reg_ss | reg_ds | reg_fs | reg_gs
```

Перемістити дані із стеку

```
    pop srcreg            ;перемістити із стеку дані 8,16,32, 64-біт регістр
    pop type [srcmem]     ;перемістити із стеку дані 8,16,32, 64-біт комірку
пам'яті
    popf                  ;перемістити дані із стеку у молодші 16-бітів EFLAGS
    popfd                 ;перемістити дані із стеку у регістр EFLAGS
    popfq                 ;перемістити дані із стеку у регістр RFLAGS
    pop reg_es | reg_cs | reg_ss | reg_ds | reg_fs | reg_gs
```

Команда повернення керування з підпрограми ret

```
    ret        ;взяти із стеку адрес повернення із підпрограми і передати
керування за
               ; адресою повернення
    ret imm16  ;взяти із стеку адрес повернення із підпрограми, додати 16-біт
                    ;константу до ESP регістра і передати керування за адресою
повернення
```

Команда віднімання sub

```
    sub destreg, constant       ;destreg := destreg - constant
                                ;destreg, регістр ЗП n=8, 16, 32, 64 біт
                                ;constant, n=8, 16, 32
    sub destmem, constant       ;destmem := destmem - constant
                                ;destmem, комірка пам’яті розміром n=8, 16, 32,
64 біт
                                ;constant, n=8, 16, 32
    sub destreg, srcreg         ;destreg := destreg - srcreg
                                ;destreg і srcreg, регістр ЗП n=8, 16, 32, 64
біт.
                                ;Регістри мають бути одного розміру

    sub destreg, [srcmem]       ;destreg := destreg - srcmem
                                ;destreg, регістр ЗП=8, 16, 32, 64 біт
                                ;srcmem, комірка пам'яті такого ж розміру

    sub [destmem], srcreg       ;destmem := destmem - srcreg
                                ;src, регістр ЗП 8, 16, 32, 64 біт
                                ;destmem, комірка пам'яті такого ж розміру
```

Контрольні запитання.

1. Асемблювання і компонування асемблерних програм на NASM і GNU AS.
2. Шаблони програм на асемблері.
3. Послідовність створення виконуваних файлів асемблера.
4. Утиліта `make`, файл `makefile`.
5. Призначення налагоджувача GDB і основні його команди.
6. На які групи поділяються команди асемблера;
7. Синтаксис і формати команд асемблера NASM і GAS.
8. Синтаксис і формати команд асемблера GAS.
9. Синтаксис і формати арифметичних команд;
10. Синтаксис і формати команд для роботи з бітами
11. Синтаксис і формати команд для роботи з прапорами
12. Синтаксис і формати логічних команд;

13. Синтаксис і формати команд передачі і галуження;
14. Синтаксис і формати команд копіювання даних;
15. Синтаксис і формати команд для роботи із стеком.

Практична частина

Завдання.

№	Зміст завдання	Асемблер	Значення			
			a	b	c	d
1	Написати програму, яка обчислює значення y=a+b+c+d. Змінні і результат розмістити в секції даних. Результат вивести в консоль.	NASM	1	2	3	4
2	Написати програму, яка обчислює значення y=a+b+c+d. Змінні розмістити в секції даних, а результат в секції буферів. Результат вивести в консоль.	NASM	2	3	4	5
3	Написати програму, яка обчислює значення y=a+b+c+d. Змінні і результат розмістити в секції даних. Змінні вводяться з консолі. Результат вивести в консоль.	NASM	3	4	5	6
4	Написати NASM програму, яка: обчислює значення y=a*b+c*d. Змінні розмістити в секції даних, а результат в секції буферів. Результат вивести в консоль.	NASM	1	2	3	4
5	Написати програму, яка обчислює значення y=a*b+c*d. Змінні розмістити в секції буферів. Результат вивести в консоль.	NASM	2	3	4	5
6	Написати NASM програму, яка: обчислює значення y=a*b+c*d. мінні і результат розмістити в секції даних. Змінні вводяться з консолі. Результат вивести в консоль.	NASM	3	4	5	6
7	Написати програму, яка обчислює значення y=a/b+c/d. Змінні і результат розмістити в секції даних. Результат вивести в консоль.	NASM	10	2	8	4
8	Написати програму, яка обчислює значення y=a/b+c/d. Змінні розмістити в секції буферів.	NASM	12	3	15	5
9	Написати програму, яка обчислює значення y=a/b+c+d. Змінні розмістити в	NASM	8	4	9	3

	секції буферів. Змінні вводяться з консолі. Результат вивести в консоль.					
10	Написати програму, яка обчислює мінімальне значення чисел a, b, c, d. Змінні розмістити в секції даних, а результат в секції буферів. Результат вивести в консоль.	NASM	1	2	3	4
11	Написати програму, яка обчислює максимальне значення чисел a, b, c, d. Змінні розмістити в секції даних, а результат в секції буферів. Результат вивести в консоль.	NASM	2	3	4	5
12	Написати програму, яка обчислює суму парних чисел a, b, c, d. Змінні і результат розмістити в секції даних. Змінні вводяться з консолі.	NASM	3	4	5	6
13	Написати програму, яка обчислює суму непарних чисел a, b, c, d. Змінні і результат розмістити в секції даних. Результат вивести в консоль.	NASM	3	4	5	6
14	Написати програму, яка: переписує значення чисел a, b, c, d із секції даних у секцію буферів у відсортованому за зростанням значень порядку. Змінні розмістити в секції даних, а результат в секції буферів. Результат вивести в консоль.	NASM	1	35	4	77
15	Написати програму, яка: переписує значення чисел a, b, c, d із секції даних у секцію буферів у відсортованому за спаданням значень порядку. Змінні розмістити в секції даних, а результат в секції буферів. Результат вивести в консоль.	NASM	35	4	77	1

Примітка. Номер варіанта завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

1. Завдання до роботи.
2. Короткий опис теоретичної частини.
3. Блок схема алгоритму програми виконана згідно стандартів.
4. Текст програми із поясненнями.
5. Роздруки екранів стану пам’яті і регістрів з результатами виконання програми у налагоджувачі `KDbg` або `dbg`.

Приклади для самостійної роботи

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

```
;ПРИЗНАЧЕННЯ: Проста програма яка повертає
;             код завершення в ОС Linux
;ВХІД: немає
;ВИХІД:   повертає код завершення. Його можна продивитися
;         вводячи в командному рядку консолі echo $?
;         після виконання програми
;ЗМІННІ:
;   rax містить номер системного виклику
;   rdi містить код завершення програми

section .data  ; секція даних
section .text  ; секція коду
global main    ; глобальна позначка точки входу в програму
main:

mov rax,60     ; системний виклик для виходу (sys_exit)
mov rdi,9      ; код завершення програми 0
               ; В ОС код завершення можна вивести командою echo $?
syscall        ; виклик ядра ОС
```

$./asm_ld.sh 2_1.asm

```
; Запуск програми в консолі:
```

$./2_1

$echo $?

```
9
```

2. 2_2.asm – виведення повідомлення на екран

```
;hello.asm
section .data
msg1 db "hello, world",10,0
len1 equ $-msg1-1   ; без Nl
msg2 db "I teatch nasm",10,0
len2 equ $-msg2-1   ; без NL
section .bss
section .text
global main
main:
mov rax, 1 ; 1 = write
mov rdi, 1 ; 1 = to stdout
mov rsi, msg1 ; string to display in rsi
mov rdx, len1 ; length of the string, without 0
syscall ; display the string

mov rax, 1 ; 1 = write
mov rdi, 1 ; 1 = to stdout
mov rsi, msg2 ; string to display in rsi
mov rdx, len2 ; length of the string, without 0
syscall ; display the string

mov rax, 60 ; 60 = exit
mov rdi, 0 ; 0 = success exit code
syscall ; quit
```

$./asm_ld.sh 2_2.asm

```
; Запуск програми в консолі:
$./2_2
Hello, world
I teach asm
```

3. 2_3.asm - цілочисельне множення

```
segment .data
var db 100           ; резервування byte і присвоєння значення 100
segment .text        ; сегмент коду
global main
 main:
dec dword [var]   ; зменшення на 1 значення за адресою var 99=100-1
mov rax,2         ; переслати ціле число 2 в регістр eax
mul qword [var]   ; помножити eax на значення за адресою var, eax=99*2=198
mov rcx,rax       ; переслати rax в rcx

mov rax,60        ; системний виклик для виходу (sys_exit)
mov rdi,rcx       ; код завершення програми 198
syscall
```

```
$./asm_ld.sh 2_3.asm
; Запуск програми в консолі:
$./2_3
$echo $?
198
```

4. 2_4.asm - цикл loop

```
segment .data     ; сегмент даних
x  dq 1,2,3,4,5   ; резервування і ініціалізація пам'яті dword
global main
main:
mov rcx,5          ; переслати в лічильник циклів rcx число 5
mov rsi,qword x    ; переслати в індексний регістр rsi адресу x
mov rax,0          ; переслати в регістр акумулятор rax число 0
lp: add rax,qword[rsi]  ; початок циклу: додати в регістр eax значення [rsi]
add rsi,8          ; збільшити значення адреси rsi на 8
loop lp            ; ecx=ecx-1 і перейти на позначку lp, цикл триває поки rcx
!= 0
mov rcx,rax        ; результат з eax переслати в rcx

mov rax,60       ; системний виклик для виходу (sys_exit)
mov rdi,rcx     ; код завершення програми 15 в rcx
syscall         ; виклик ядра ОС
```

```
$./asm_ld.sh 2_4.asm
; Запуск програми в консолі:
$./2_4
$echo $?
15
```

5. 2_5.asm - команда and

```
segment .data  ; сегмент даних
segment .text  ; сегмент коду
global main
main:
```

```
mov rax,100          ; переслати в регістр eax число     100 -> 0110_0100
mov rcx,0x0000_000f  ; переслати в регістр ecx маску число 15-> 0000_1111
                     ;                                          0000_0100
and rcx,rax          ; побітове AND над eax і ecx, результат 4
mov rax,60    ; системний виклик для виходу (sys_exit)
mov rdi,rcx   ; код завершення 4 програми в rcx
syscall       ; виклик ядра ОС
```

$./asm_ld.sh 2_5.asm

```
; Запуск програми в консолі:
```

$./2_5
$echo $?

```
4
```

6. 2_6.asm – команда test

```
segment .data
segment .text
global main
main:
mov rax,0
mov rcx,0
mov rax,0000_1000b
mov rcx,0000_1111b
test rax,rcx        ; порозрядна операція AND на регістрами ax і cx
jz m0               ; якщо прапор zero = 0, то перейти на позначку m0
mov rcx,1           ; якщо прапор zero = 1 переслати в cx число 1
jmp m1              ; і перейти на позначку m1
m0:
mov rcx,0
m1:

mov rax,60      ; системний виклик для виходу (sys_exit)
mov rdi,rcx     ; код завершення програми в ecx
syscall         ; виклик ядра ОС
```

$./asm_ld.sh 2_6.asm

```
; Запуск програми в консолі:
```

$./2_6
$echo $?

```
1
```

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

```
; максимальне з трьох чисел (32-бітова версія)
segment .data
var1 dq 30
var2 dq 20
var3 dq 40
segment .text
global main
main:
; move вміст змінних
      mov rcx,[var1]    ; rcx <- var1
      cmp rcx,[var2]
      jg  check_var3    ; rcx > var2
```

```
      mov rcx,[var2]    ; var1 <= var2
check_var3:
     cmp rcx,[var3]    ; rcx > var3
     jg    exit
     mov rcx,[var3]    ; rcx <= var3
exit:
      mov rax,60
      mov rdi,rcx
      syscall
```

$./asm_ld.sh 2_7.asm

```
; Запуск програми в консолі:
```

$./2_7

$echo $?

40

8. Структура об'єктного модуля

Команда nm виводить список символів об'єктного модуля

$nasm -f elf64 2_7.asm -o 2_7.o

$nm 2_7.o

```
        1        2   3
000000000000001a t check_var3
000000000000002c t exit
0000000000000000 T main
0000000000000000 d var1
0000000000000008 d var2
0000000000000010 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

```
;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 dq 3,67,34,122,45,75,54,34,44,33,22,11,66,0
segment .text
global main
main:
mov rdi,0       ; завантажити 0 в індексний регістр
mov rax,[data_items+rdi*8] ; переслати в eax перше значення довжиною 8 байтів
mov rbx,rax     ; так як це перший елемент то rax містить найбільше число
start_loop:     ; позначка початку циклу
cmp rax,0       ; перевірка, чи не досягнуто кінець даних 0
je loop_exit    ; якщо 0 то вихід з циклу
```

```
inc rdi          ; збільшити індекс edi
mov rax, [data_items+rdi*8]  ; завантажити наступне значення
cmp rbx,rax      ; порівняти значення
jg start_loop    ; перехід на початок циклу, якщо нове значення менше max
mov rbx,rax      ; оновити найбільше значення
jmp start_loop   ; перехід на початок циклу
loop_exit:
mov rax,60       ; системний виклик для виходу (sys_exit)
mov rdi,rbx      ; в ebx найбільше значення
syscall          ; виклик ядра ОС
```

$./asm_ld.sh 2_9.asm

; Запуск програми в консолі:

$./2_9

>echo $?

122

10. 2_asm – виклик підпрограми printf

```
section .data
msg1 db "Значення ",0
msg2 db "Результат ",0
radius dd 15
pif dd 3.14
pi  dq 3.14
fmtstr db "%s",10,0 ;format for printing a string
fmtflt db "%lf",10,0 ;format for a float
fmtint db "%d",10,0 ;format for an integer
section .bss
section .text
global main
extern printf
main:
  push rbp
  mov rbp,rsp

; print msg1
  mov rax, 0 ; no floating point
  mov rdi, fmtstr
  mov rsi, msg1
  call printf

; print radius
  mov rax, 0 ; no floating point
  mov rdi, fmtint
  mov rsi, [radius]
  call printf

  ; множення
  mov rax,0           ; no float point
  mov eax, [radius]   ; множене
  mov ebx,[pif]       ; множник
  mul ebx             ; результат ax
  mov rsi,rax

; print msg2
```

```
  mov rax, 0 ; no floating point
  mov rdi, fmtstr
  mov rsi, msg2
  call printf

; print результат
  mov rax,0
  mov rdi, fmtflt
  call printf


; print pi
  mov rax, 1 ; 1 xmm register used
  movq xmm0, [pi]
  mov rdi, fmtflt
  call printf

  mov rsp,rbp
  pop rbp
  ret
```

$./asm_ld.sh 2_10.asm

```
; Запуск програми в консолі:
```

$./2_10

>echo $?

```
Значення
15
Результат
0.000000
3.140000
```

Лабораторна робота 3

Налагоджувачі GDB, DDD, KDBG

Мета: вивчення команд консольного і графічного налагоджувача програм на асемблері

Теоретична частина

1. Налагоджувачі

Для виявлення помилок у виконуваних файлах використовуються консольні налагоджувачі (GDB, GNU DEbugger) або налагоджувачі з графічним інтерфейсом (DDD, KDbg, SASM). Основним налагоджувачем (зневадником) в Linux із консольним інтерфейсом є GDB.

GDB підтримує наступні мови: Ada, Assembly, C, C++, D, Forntan, Go, Objective-C, OpenCL, Modula-2, Rust.

DDD – це графічний інтерфейс для налагоджувачів командного рядка, таких як GDB, DBX, WDB, bashdb, pydb. Окрім звичайних функцій, таких як перегляд початкових (сирцевих) кодів, підтримується інтерактивне графічне відображення даних, де структури даних відображаються у вигляді графіків.

KDBG – це програма для налагодження як звичайних програм так і ядер Linux, NetBSD, FreeBSD. При роботі з ядрами потрібні дві машини з'єднані послідовним інтерфейсом. Послідовне з'єднання може бути через інтерфейс RS-232 за допомогою нуль-модемного кабелю або через мережевий протокол UDP.

SASM – Open Source середовище розробки програм на мовах NASM, MASM, GAS, FASM з підсвічуванням синтаксису і налогоджувачем.

Для вивчення GDB використовується програма `hello.asm`:

```
;hello.asm
section .data
msg db "hello, world",10,0  ; 10 => '\n'
msgLen equ $-msg-1 ; $ - поточна адреса
section .bss
section .text
global main
main:
mov rax, 1 ; 1 = write
mov rdi, 1 ; 1 = to stdout
mov rsi, msg ; рядок для виводу у rsi
mov rdx, msgLen ; довжина рядка, без '\n'
syscall ; виведення рядка
mov rax, 60 ; 60 = exit
mov rdi, 0 ; 0 = код успішного виходу
syscall ; quit
```

Для отримання бінарного файлу з додатковою інформацією для налагоджувача потрібно виконати асемблювання і компонування початкової програми:

```
$ ./asm_ld.sh hello.asm
```

Запуск GDB для налагодження бінарного файлу `hello`:

```
$ gdb hello
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
```

```
…
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello...done.
(gdb)
```

GDB завантажує програму в пам’ять і очікує наступну команду:

```
(gdb)list 1, 14
1       ;hello.asm
2       section .data
3       msg db "hello, world",10,0  ; 10 => '\n'
4       msgLen equ $-msg-1 ; $ - поточна адреса
5       section .bss
6       section .text
7       global main
8       main:
9       mov rax, 1 ; 1 = write
10      mov rdi, 1 ; 1 = to stdout
11      mov rsi, msg ; рядок для виведення у rsi
12      mov rdx, msgLen ; довжина рядка, без '\n'
13      syscall ; виведення рядка
14      mov rax, 60 ; 60 = exit
```

GDB вивів текст програми. Якщо текст виведений у форматі GAS асемблера, то потрібно змінити налаштування GDB на формат Intel асемблера:

```
set disassembly-flavor intel
```

Якщо ввести команду `run` то `GDB` виконає програму і результат виведе на екран:

```
(gdb) run
Starting program: /home/victor/Nasm/Lab3/hello
hello, world
[Inferior 1 (process 54) exited normally]
```

Позначка входу в асемблерні програми `main`, тому деасемблювання програми потрібно починати саме з неї:

```
(gdb) disassemble main
  0x00000000004004a0 <+0>:     mov    eax,0x1
  0x00000000004004a5 <+5>:     mov    edi,0x1
  0x00000000004004aa <+10>:    movabs rsi,0x601028
  0x00000000004004b4 <+20>:    mov    edx,0xd
  0x00000000004004b9 <+25>:    syscall
  0x00000000004004bb <+27>:    mov    eax,0x3c
  0x00000000004004c0 <+32>:    mov    edi,0x0
  0x00000000004004c5 <+37>:    syscall
  0x00000000004004c7 <+39>:    nop    WORD PTR [rax+rax*1+0x0]
End of assembler dump.
```

У роздруку перший стовпчик зліва задає адреси комірок пам’яті в яких розміщені машинні інструкції програми `hello`. Другий стовпчик вказує на кількість байтів, які займає команда. Так команда `mov eax, 0x1` займає 5 байтів.

Якщо подивитися на початковий текст програми то видно, що там використовуються 64-розрядні регістри `rax`, `rdi`, `rsi`, `rdx`. Асемблер проводить аналіз і де можливо замінює 64-розрядні регістри на 32-розрядні. Так для збереження одиниць достатньо 32-розрядних регістрів `eax, edi`.

Інструкція `mov rsi, msg` була замінена на інструкцію `movabs rsi, 0x601028`. 0x601028 це адреса де зберігається `msg`.

Вивести вміст комірки за адресою:

```
(gdb) x/s 0x601028
0x601028 <msg>: "hello, world\n"
```

“x” означає перевірити (examine), а “s” – стрічка (string).

Вивести перший символ `msg`:

```
(gdb) x/c 0x601028
0x601028 <msg>: 104 'h'
```

“c” означає стрічка (string).

Вивести задану кількість символів із заданої адреси у 16-му форматі:

```
(gdb) x/14x 0x601028
0x601028 <msg>: 0x68    0x65    0x6c    0x6c    0x6f    0x2c    0x20    0x77
0x601030:       0x6f    0x72    0x6c    0x64    0x0a    0x00
```

Вивести вміст позначки:

```
(gdb) x/s &msg
0x601028 <msg>: "hello, world\n"
```

Вивести вміст двох комірок пам’яті із заданої адреси:

```
(gdb)x/2x 0x4004a0
0x4004a0 <main>:        0xb8    0x01
```

Вийти з GDB:

```
(gdb)q
```

Знову запустити GDB:

```
$ gdb hello
```

Задати точку зупинки програми (break poin)**:**

```
(gdb)break main
Breakpoint 1 at 0x4004a0: file hello.asm, line 9.
```

Запустити програму:

```
(gdb)run
Starting program: /home/victor/Nasm/Lab3/hello
Breakpoint 1, main () at hello.asm:9
9       mov rax, 1 ; 1 = write
```

Налагоджувач зупиниться на заданій точці зупинки і покаже наступну команду. Тобто команда `mov rax, 1` ще невиконана.

Вивести вміст регістрів:

```
(gdb) info registers
rax            0x4004a0 4195488
rbx            0x0      0
rcx            0x4004d0 4195536
rdx            0x7fffffffee1f8   140737488282104
rsi            0x7fffffffee1e8   140737488282088
```

```
rdi            0x1      1
rbp            0x4004d0 0x4004d0 <__libc_csu_init>
rsp            0x7fffffffee108   0x7fffffffee108
r8             0x7fffff3ecd80   140737475693952
r9             0x7fffff3ecd80   140737475693952
r10            0x2      2
r11            0x3      3
r12            0x4003b0 4195248
r13            0x7fffffffee1e0   140737488282080
r14            0x0      0
r15            0x0      0
rip            0x4004a0 0x4004a0 <main>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

Зараз вміст регістрів не є важливим, за винятком лічильника команд `rip`. Регістр rip має значення `0x4004a0`, яке є адресою комірки наступної інструкції до виконання. У роздруку дизасемблера ця адреса вказує на команду `mov eax,0x1`. GDB зупинився перед цією інструкцією і очікую наступну команду. **Важливо пам'ятати, що інструкція, на яку вказує `rip`, ще не виконувалася**.

Виконати одну інструкції програми:

```
(gdb)step
10      mov rdi, 1 ; 1 = to stdout
```

Вивести вміст регістрів:

Для виведення вмісту регістрів замість команди `info registers` можна використати абревіатуру `i r`:

```
(gdb)i r
rax            0x1      1
rbx            0x0      0
rcx            0x4004d0 4195536
rdx            0x7fffffffee1f8   140737488282104
rsi            0x7fffffffee1e8   140737488282088
rdi            0x1      1
rbp            0x4004d0 0x4004d0 <__libc_csu_init>
rsp            0x7fffffffee108   0x7fffffffee108
r8             0x7fffff3ecd80   140737475693952
r9             0x7fffff3ecd80   140737475693952
r10            0x2      2
r11            0x3      3
r12            0x4003b0 4195248
r13            0x7fffffffee1e0   140737488282080
r14            0x0      0
r15            0x0      0
rip            0x4004a5 0x4004a5 <main+5>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
```

```
ss             0x2b      43
ds             0x0       0
es             0x0       0
fs             0x0       0
gs             0x0       0
```

Тепер регістр `rax` містить 0x1, а `rip` містить адресу наступної команди, яка ще не виконувалася.

Послідовно вводити команди `next`, аналізувати вміст регістра `rip` і коли в регістрі `rsi` появиться адреса `msg`.

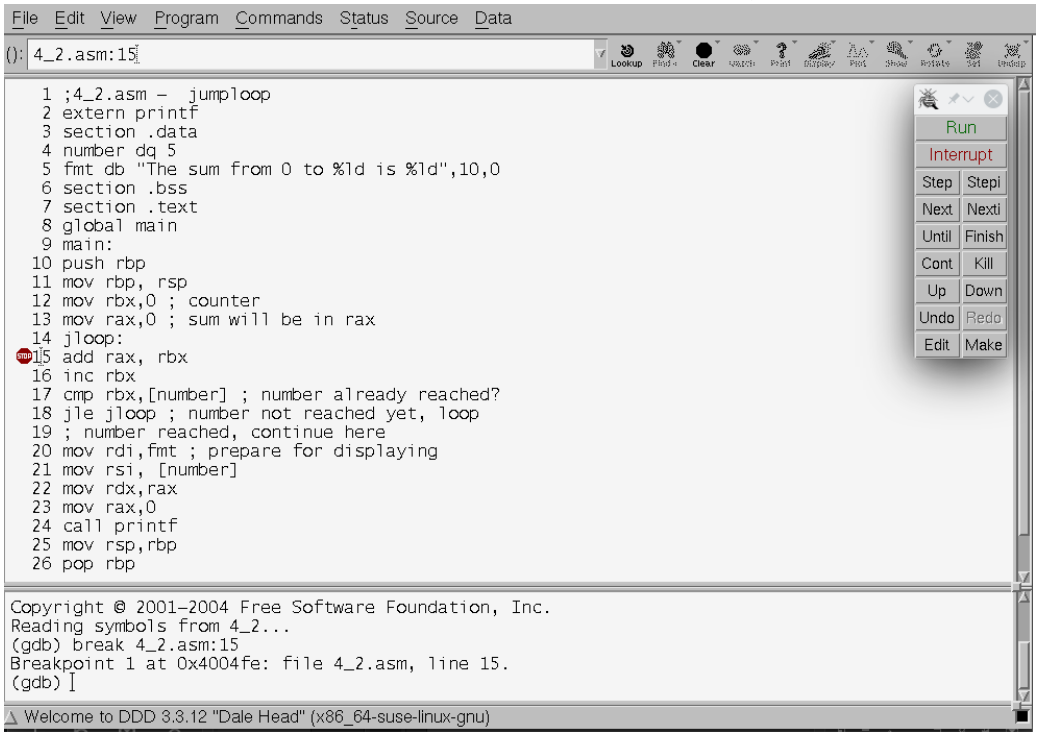
2. Графічний налагоджувач DDD

Налагоджувач даних (DDD) – це інструмент налагодження з графічним інтерфейсом користувача для Linux. Встановлення налагоджувача DDD:

```
$ sudo apt install ddd
```

Запуск на виконання:

```
$ ddd <binfile>
```



Налагоджувач буде використовуватися для дослідження програми, яка не виводить дані.

```
; move.asm
section .data
bNum db 123
wNum dw 12345
dNum dd 1234567890
qNum1 dq 1234567890123456789
qNum2 dq 123456
qNum3 dq 3.14
section .bss
section .text
global main
```

```
main:
push rbp
mov rbp,rsp
mov rax, -1 ; заповнити rax значенням 1s
mov al, byte [bNum] ; не чистити старші біти rax
xor rax,rax ; чистити rax
mov al, byte [bNum] ; тепер rax має коректне значення
mov rax, -1 ; заповнити rax значеннями 1s
mov ax, word [wNum] ; не чистити старші біти rax
xor rax,rax ; чистити rax
mov ax, word [wNum] ; тепер rax має коректне значення
mov rax, -1 ; заповнити rax значеннями 1s
mov eax, dword [dNum] ; чистити старші біти rax
mov rax, -1 ; заповнити rax значеннями 1s
mov rax, qword [qNum1] ; чистити старші біти rax
mov qword [qNum2], rax ; один операнд завжди регістр
mov rax, 123456 ; операнд джерело є безпосереднє значення
movq xmm0, [qNum3] ; команда для плаваючої крапки
mov rsp,rbp
pop rbp
ret
```

Команди асемблювання і отримання бінарного (виконуваного) файлу програми:

```
$ ./asm_ld.sh move.asm
$ ls
asm_ld.sh move move.asm  move.lst
```

Запуск програми у налагоджувачі:

```
$ ddd move
```

2.1. Виконання програми

Для виконання програми натиснути **Run** з меню команд графічного середовища або набрати команду **run** в консолі gdb (внизу вікна).

2.2. Встановлення точок запинки програми

Це можна зробити трьома способами:

- клацнути правою кнопкою миші на номері рядка і вибрати *Set breakpoint*;
- у консолі команд GDB набрати: *break last;*
- у консолі команд GDB набрати: *break 15.*

В результаті, зліва від номера заданого рядка висвітиться червоного кольору іконка STOP.

2.3. Запуск програми на виконання із зупинкою

Натиснути **Run** з меню команд графічного середовища або набрати команду **run** в консолі gdb (внизу вікна). Програма виконається до рядка, на який вказує зелена стрілка. Зелена стрілка вказує на команду, яка буде виконуватися *наступною*.

Після кожного натискання Run програму зупиняється перед першою точкою зупинки. Для продовження виконання програми до наступної точки зупинки, потрібно натиснути **Cont** з меню команд графічного середовища або набрати команду **cont** в консолі gdb.

Для порядкового виконання програми потрібно натиснути **Step** або **Next** з меню команд графічного середовища або набрати команду **step** або **next** в консолі gdb. Команда **next** виконує один рядок команди або всю функцію. Команда **step** виконує один рядок команди і при необхідності заходить у функцію. Для рядків з одної команди між **next** і **step** різниці немає.

2.4. Виведення вмісту регістрів

Вивести вміст регістрів можна вибором меню Status/Registers…. У вікні регістрів висвічується три стовпці — назва регістрів, вміст регістрів у шістнадцятковій системі, вміст регістрів у беззнаковій десятковій системі. Знакові десяткові значення виводяться як беззнакові, що треба мати на увазі.

2.5. Короткий перелік основних команд DDD/GDB

Команди	Описання
quit | q	Вихід з налагоджувача
break <label/addr> | b <label/addr>	Поставити точку зупинки на <lavel> або <addr>
run <args> | r <args>	Виконати програму до першої точки зупинки
continue | c	Продовжити виконання програми до наступної точки зупинки
continue <n> | c <n>	Продовжити виконання програми до наступної точки зупинки виконуючи n-1 ітерацію
step | s	Крок до наступної інструкції (крок в функцію/виклик процедури)
next | n	Наступна інструкція (крок без входження у функцію/виклик процедури)
F3	Рестарт програми і зупинка на першій точці зупинки.
where	Поточна активація (call depth)
x/<n><f><u> $rsp	Виведення вмісту стека
x/<n><f><u> & <variable>	Виведення вмісту комірки пам'яті <variable> <n> число позицій для виведення <f> формат: d – десятковий x – шістнадцятковий u – десятковий беззнаковий c – символ s – стрічка f – плаваюча крапка <u> - розмір: b – byte (8-біт)

	h – halfword (16-біт) w – word (32-біт) g – giant (64-біт)
source <filename>	читання команд з файлу <filename>
set logging file <filename>	gdb.txt за замовчуванням
set loggin on	включити журнал
set loggin on	відключити журнал
set loggin overwrite	Перезаписати журнал

2.6. Приклади використання команд налагоджувача

2.6.1. Комірки пам’яті

Оголошено і ініціалізовані дані:

```
bnum1 db 5
wnum2 dw -2000
dnum3 dd 100000
qnum dq 1234567890
str  db "Assembly", 0
pi dd 3.14
```

Команди для дослідження комірок пам’яті:

```
x/db &bnum1
x/dh &wnum2
x/dw &dnum3
x/dg &qnum
x/s &str
x/f &pi
```

Синтаксис виведення значень масиву комірок в DDD:

```
x/<n><f><u> &<variable>
```

Наприклад:

```
list1 dd 100001, -100002, 100003, 100004, 100005
x/5dw &list1
```

де 5 довжина масиву, d – знакові дані, w – дані 32-біти, &list1 – адреса змінної list1.

Можна вказувати безпосередньо адресу:

```
x/dw 0x600d44
```

2.6.2. Стек

Стек звичайно містить 64-бітові беззнакові значення. Адреса вказується у `rsp` регістрі. Наприклад, виведення значення верхівки стеку та 6-ти елементів від верхівки стеку:

```
x/ug $rsp
x/6ug $rsp
```

2.6.3. Введення команд налагоджувача із файлу (інтерактивний режим)

Налагоджувач може читати команди з файлу (замість ручного введення), наприклад `gdb.txt`. Результати можна відобразити на на екран або перенаправити у вихідний файл.

Наприклад, можна встановлення точки зупинки, запуску програми, відобразити деякі змінні і переспрямувати вивід до файлу журналу:

```
#-----------------------------------
# Debugger Input Script
#-----------------------------------
echo \n\n
break last
run
set pagination off
set logging file out.txt
set logging overwrite
set logging on
set prompt
echo ----------------------------------- \n
echo display variables \n
echo \n
x/100dw &list
x/dw &length
echo \n
x/dw &listMin
x/dw &listMid
x/dw &listMax
x/dw &listSum
x/dw &listAve
echo \n \n
set logging off
quit
```

2.6.4. Файл команд налагоджувача (не інтерактивний режим)

Команда налагоджувача для читання файлу:

```
(gdb) source <filename>
```

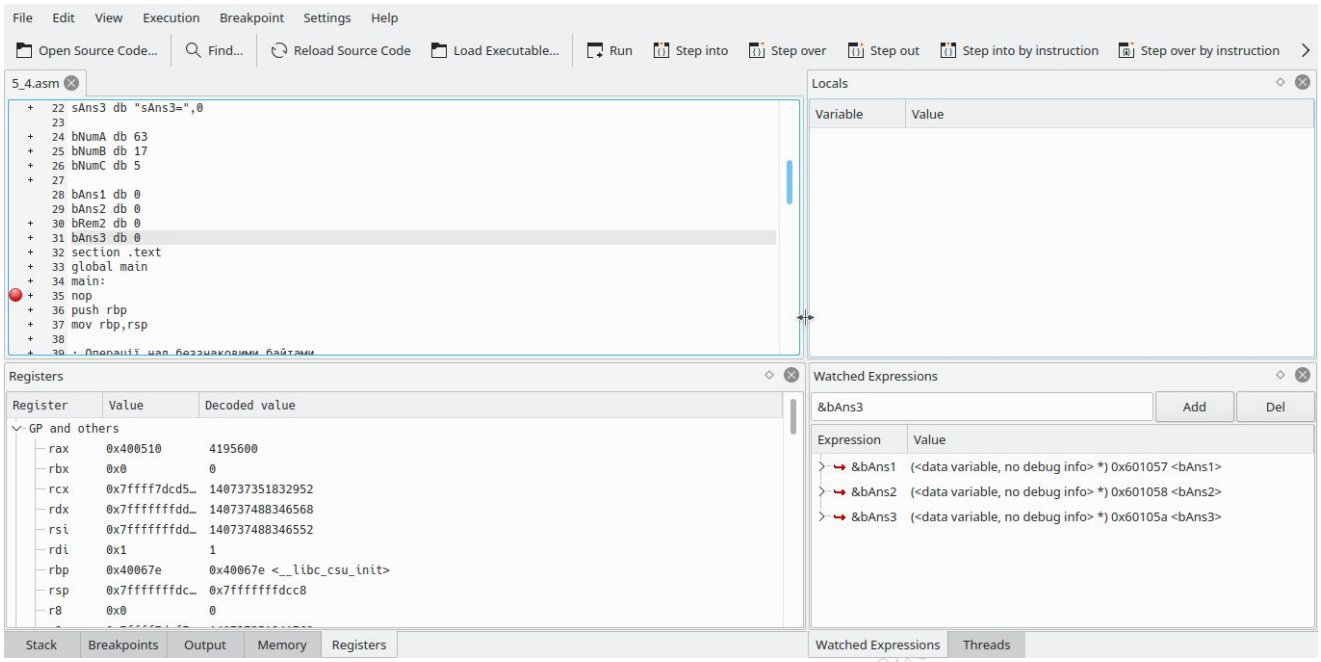
Результат буде записаний у фал `out.txt`.

Кожна програма вимагає спеціальний набір. Файл вхідних команд налагоджувача буде корисним лише тоді, коли програма майже роботоздатна. Збої програми та інші більш значні помилки вимагають інтерактивного налагодження для визначення конкретної помилки.

Вихідний файл можна отримати безпосередньо без інтерактивного сеансу DDD. Наступна команда, введена в командному рядку, виконає команди з вхідного файлу, створить вихідний файл і вийде з програми.

```
gbd <gdb.txt prog
```

3. Графічний налагоджувач KDbg



Можливості налагоджувача KDbg при роботі із звичайними програмами асемблера подібні до можливостей налагоджувача DDD. Відмінність у тому, що для налагодження потрібно завантажувати як початковий код програми, так і бінарний виконуваний файл.

KDbg дозволяє переглядати стан стеку, локальних змінних, регістрів, пам’яті, потоків, результатів, точок переривання, спостережуваних виразів.

2. Практична частина

Контрольні запитання.

1. Для чого призначені налагоджувачі програм.
2. Можливості GDB налагоджувач. Послідовність налагодження програми.
3. Можливості DDD налагоджувач. Послідовність налагодження програми.
4. Можливості KDbg налагоджувача.
5. Можливості SASM налагоджувача.
6. Які основі команди GDB налагоджувача.
7. Як виконати команди DDD налагоджувача із файлу?
8. Для чого призначена точка зупинки програми (break point) і яка інформацію можна отримати у цьому стані програми?
9. Що виконують команди `run` і `continue`.
10. Яка різниця між командами `next` і `step`?
11. Яка інформація зберігається у `rip` регістрі і яку інформацію можна отримати її його вмісту?
12. Якою командою модна вивести вміст стеку і комірки пам’яті?
13. Які формати даних модна використати для виведення вмісту стеку і комірки пам’яті?

Завдання.

Написати і налагодити в GDB програму, яка обчислює арифметичний вираз. Результат обчислень помістити у буферну секцію.

Змінним a, b, c, d присвоїти цілочисельні значення. Вивести значення комірок пам’яті в яких зберігаються значення змінних a, b, c, d і результат sum.

Варіанти.

1. sum=a+b+c+d. Змінні типу byte.
2. sum=a+b-c-d. Змінні типу word.
3. sum=a+b*c*d. Змінні типу dword.
4. sum=a+b/c/d. Змінні типу qword.
5. sum=a-b+c+d. Змінні типу byte.
6. sum=a*b+c-d. Змінні типу word.
7. sum=a+b+c*d. Змінні типу dword.
8. sum=a-b-c/b. Змінні типу qword.
9. sum=a+b*c+d. Змінні типу byte.
10. sum=a+b/c-d. Змінні типу word.
11. sum=a-b+c*d. Змінні типу dword.
12. sum=a*b+c/b. Змінні типу qword.
13. sum=a+b-c/d. Змінні типу byte.
14. sum=a-b+c*d. Змінні типу dword.
15. sum=a*b+c-d. Змінні типу qword.

Лабораторна робота 4

Передача керування, логічні і бітові команди

Мета роботи: вивчення команд передачі керування, логічних і бітових команд, операцій над виразами

Теоретичні відомості

1. Безумовні та умовні переходи

1.1. Безумовні переходи

В системі команд процесора x86_64 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 rax    ; перехід за адресою, яка знаходиться в регістрі rax
jmp [rax]  ; перехід за адресою, яка знаходиться в комірці пам'яті
              з адресою, яка знаходиться у регістрі rax
```

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

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

1.2. Прості умовні переходи за значеннями прапорів

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 rax, rbx     ; 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. Команди умовних переходів за результатами порівняння команди cmp

За результатами команди порівняння `cmp` можуть встановлюватися два прапори, наприклад `sf=1 (знак), of=0 (переповнення)` (результат негативний, переповнення не було,) або `sf=0, of=1` (результат позитивний, 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
знакові, беззнакові операнди				
je	equal	a == b	zf=1	jz
jne	not equal	a != b	zf=0	jnz
знакові операнди				
jl jnge	less not greater or equal	a < b	(sf XOR of) = 1	
jle jng	less or equal not greater	a <= b	((sf XOR of) OR zf) = 1	
jg jnle	greater not less or equal	a > b	((sf XOR of) or zf) = 0	
jge jnl	greater or equal not less	a >= b	(sf XOR of) = 0	
беззнакові операнди				
jb jnae	below not above or equal	a < b	cf=1	jc
jbe jna	below or equal not above	a <= b	cf=1 OR zf=1	
ja jnbe	above not below or equal	a > b	cf OR zf) = 0	
jae jnb	above or equal not below	a >= b	cf=0	jnc

Приклад використання команд:

```
.text
    mov rax,15
    cmp rax,15      ; порівняння
    jne not_equal   ; якщо операнди не рівні, перейти на позначку not_equal
        ; команди для rax = 15
        jmp out
 not_equal:
        ; команди для rax != 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      ; [b]<10 (below)
cmp  [c],0
sete  bl      ; c==0 (equal)
```

При порівнянні беззнакових і знакових чисел встановлюються різні пропори регістра `rflags`, рис.1

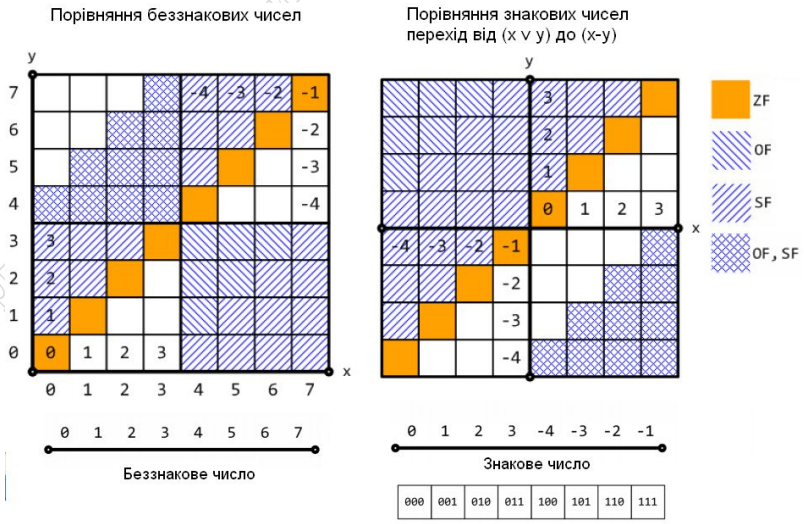


Рисунок 1 – Порівняння беззнакових і знакових чисел

2. Команда циклів `loop`

Команда **`loop`** призначена для організації циклів з наперед відомою кількістю ітерацій, яка задається у регістрі `rcx`/`ecx`/`cx`.

Синтаксис команди `loop`:

```
mov rax,10
позначка:
…
loop позначка
…
```

Принцип роботи:

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

Як лічильник ітерацій команда `loop` використовує регістр **`rcx/ecx/cx`**, в який перед початком циклу записується потрібне число ітерацій. Сама команда `loop` виконує дві дії: зменшує на одиницю значення в регістрі `rcx/ecx/cx` 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 rcx,10     ; кількість ітерацій
   mov rsi,array  ; адреса першого елементу
   mov rax,0      ; початкове значення суми
lp: add rax,[rsi] ; додати число до суми, esi індекс масиву array
   add rsi,8      ; адреса наступного елементу
   loop lp        ; зменшення лічильника і якщо ecx не 0, перехід на lp
```

Замість команди `loop` можна використати дві команди:

```
dec rcx
jnz lp
```

У наступному прикладі сума обчислюється без регістра `rsi`, але у цьому випадку елементи масиву вибираються з кінця:

```
mov rcx,10
mov rax,0
lp: add rax,[array+8*rcx-8]  ; loop зменшує значення ecx
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 `ecx/cx` є дві додаткові команди умовного короткого переходу:

`jecxz` - умовний перехід, якщо в регістрі `ecx` нуль і `jcxz` - умовний перехід, якщо в регістрі `cx` 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 передумовою, подібні до **`while(){}`** в мові програмування Сі. 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:
```

Також на асемблері можна створювати довільні цикли з післяумовою, подібні до **`do{} while()`** в мові програмування Сі. 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:
```

Реалізація циклу `do-while` на мові Сі і асемблера:

```
int pcount_do(unsigned x) {
  int result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
  return result;      // x->rdx, result->rcx
}

   mov rcx, 0     ;   result = 0
.L2:              ; loop:
   mov rax, rdx
   and rax, 1     ;   t = x & 1
   add rcx, rax   ;   result += t
   shr rdx, 1     ;   x >>= 1
   jne .L2        ;   If !0, goto loop
```

3. Логічні команди і бітові операції

3.1. Логічні команди

Для роботи з комірками пам’яті і регістрами використовуються логічні команди **`and`**, **`or`**, **`xor`** (виключне або), **`not`** (інверсія).

```
and   джерело, приймач
or    джерело, приймач
xor   джерело, приймач
```

```
not   операнд
```

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

```
xor rax, rax  ;  розмір команди 3 байти
mov rax, 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 регістра `rflags/eflags/flags`. Тому, якщо потрібно зберегти стан прапорів, використовується команда `mov`.

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

```
sub eax,eax  ; тепер eax == 0
```

Логічні команди побітового `and`, `or` і `xor` подібні до інструкцій мови програмування Сі `&`, `|`, `^`. 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 Сі `~`.

3.2. Команди для роботи з бітами

Для роботи з бітами використовують команди `bt, bts, btr, btc, setcc, test.`

Команда `bt` (`bit test`) переносить значення біта регістра у прапор `cf`.

```
bt операнд, зміщення_біта
bt ax,5   ; перевірити значення біта 5
jnc m1    ; перехід, якщо біт = 0
```

Команда `bts` (`bit test and set`) переносить значення біта регістра у прапор `cf` і потім встановлює перевірюваний біт регістра в 1.

```
bts операнд, зміщення_біта
btc ax,10 ; перевірити і встановити 10-й біт
jc m1     ; перехід, якщо перевірюваний біт = 0
```

Команда `btr` (`bit test and reset`) переносить значення біта регістра у прапор `cf` і потім встановлює перевірюваний біт регістра в 0.

Команда `btc` (`bit test and convert`) переносить значення біта регістра у прапор `cf` і потім інвертує значення цього біту регістра.

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

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

```
test al, 0b0000_1001

 al  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 */
```

Команда `test` може порівнювати значення регістра з нулем:

```
test rax,rax
je is_zero
   ; eax != 0 */
is_zero:
   ; eax == 0
```

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

3.2.1. Бітові операції над виразами в операндах

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
```

Бітова операція над виразом на мові Сі і асемблера:

```
int pierce_arrow(int a, int b) {
   int t = ~(a | b);
   return t;
}

section .text
```

```
global arrow
arrow:
   push  rbp
   mov   rbp, rsp
   mov   rax, qword [rbp+16] ; (1)
   or    rax, qword [rbp+8]  ; (2)
   not   rax                 ; (3)
   pop   rbp
   ret
```

3.3. Обчислення поточних адрес `$, $$`

Асемблер обчислює поточну адресу виразу 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 **`$$`**. Так вираз `$-$$` визначає зміщення поточної команди від початку поточного сегменту.

4. Зміна порядку виконання програми

При виконанні арифметичних команд або перетворенні чисел встановлюються відповідні пропори регістра `rflags/eflags/flags`. За допомогою наступних команд можна перевірити прапори і змінити порядок виконання програми (тобто змінити регістр `rip/eip/ip`):

- `cmp`
  - `reg/mem 8/16/32,64 imm 8/16/32`
  - `reg/mem 8/16/32,64 reg 8/16/32,64`
  - `reg 8/16/32,64 reg/mem 8/16/32,64`
- `test`
  - `reg/mem 8/16/32,64 imm 8/16/32`
  - `reg/mem 8/16/32,64 reg 8/16/32,64`
- `jmp`
  - `reg/mem/imm 8,16,32,64`
- `jcc`
  - `imm 8/16/32`

В табл. 3 показано, як команди змінюють прапори регістра `rflags/eflags`.

Таблиця 3 – Стан регістра `rflags` після виконання команд

	OF	SF	ZF	PF	CF
ADD, SUB, NEG	M	M	M	M	M
INC, DEC	M	M	M		
IMUL, MUL	M	-	-	-	M
IDIV, DIV	-	-		-	-
CBW, CWD, CDQ					
MOV, MOVSX, MOVZX					
CMP	M	M	M	M	M
TEST	0	M	M	M	0

M – команда модифікує прапор;
«-» – дія команди на команди не визначена;
« » – команда не впливає на прапор;
«0» – команда скидає прапор.

5. Операції зсуву

В асемблері операції побітового зсуву реалізовуються командами:
- логічного зсуву;
- арифметичного зсуву;
- простого циклічного зсуву;
- циклічного зсуву через прапор CF.

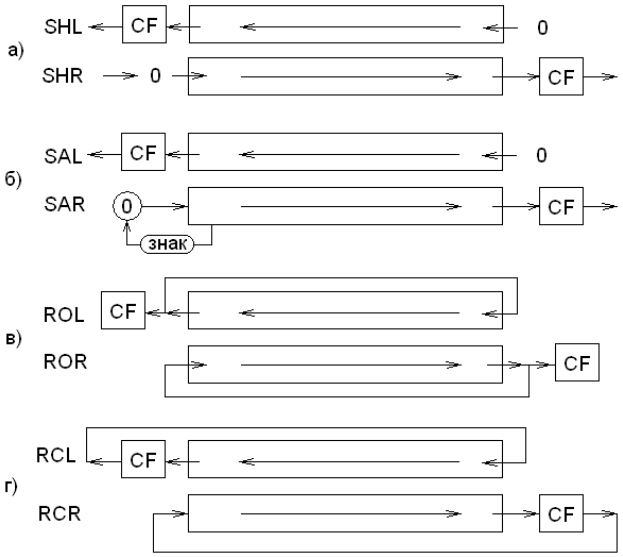


Рисунок – 1. Операції побітового зсуву: а) логічного; б) арифметичного; в) простого циклічного; г) циклічного зсуву через прапор CF

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 зсунутий за розрядну сітку біт попадає в прапор `cf`, причому звільнений біт з другої сторони заповнюється 0. Таким чином, в прапорі `cf` виявляється самий останній

зсунутий 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 діленню 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
```

Команда `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.

Арифметичний зсув регістра на мові Сі і асемблера:

```
char upndown(char x) {
   return (x << 8) >> 8;
}

section .text
global upndown
upndown:
   push   ebp
   mov    ebp, esp
   movsx  eax, byte [ebp+8]
   sal    eax, 8
   sar    eax, 8
   pop    ebp
   ret
```

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
```

Циклічний зсув регістра на мові Сі і асемблера:

```
unsigned sha256_f1(unsigned x) {
   unsigned t;
   t  = ((x >> 2)  | (x << ((sizeof(x) << 3) - 2)));  // (1)
   t ^= ((x >> 13) | (x << ((sizeof(x) << 3) - 13))); // (2)
   t ^= ((x >> 22) | (x << ((sizeof(x) << 3) - 22))); // (3)

   return t;
}

global sha256_f1
sha256_f1:
   push  ebp
   mov   ebp, esp
   mov   edx, dword [ebp+8] ; (1)
   pop   ebp                ; (2)
   mov   eax, edx           ; (3)
   mov   ecx, edx           ; (4)
   ror   eax, 13            ; (5)
   ror   ecx, 2             ; (6)
   xor   eax, ecx           ; (7)
   ror   edx, 22            ; (8)
   xor   eax, edx           ; (9)
   ret
```

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…63. 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.

Контрольні 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.`

Практична частина

Завдання.

1. Написати програму на асемблері в якій обчислюється вираз і результат вивести в консоль:

```
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 і результат вивести в консоль.

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

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
```

і результат виводить в консоль:

5. Написати програму на асемблері в якій реалізувати фрагмент програми на мові С і результат вивести в консоль:

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

6. Написати програму на асемблері в якій реалізувати фрагмент програми на мові С і результат вивести в консоль:

```
int a[5];i=0;
while ( i < 5 )a[i]=i++;
```

7. Написати програму на асемблері в якій реалізувати фрагмент програми на мові С і результат вивести в консоль:

```
int a[5];i=5;
do { a[i]=i--; } while (i==0);
```

8. Написати програму на асемблері в якій реалізувати фрагмент програми на мові Сі результат вивести в консоль:

```
a=0;i=3;
if (i>5) then a=5;
else a=10;
```

9. Написати програму на асемблері в якій реалізувати фрагмент програми на мові Сі результат вивести в консоль:

```
a=0;i=3;j=5;
if (i==3) then a=5;
else {
if (j<5) then a=4;
else a=1;
}
```

10. Написати програму на асемблері в якій реалізувати фрагмент програми на мові Сі результат вивести в консоль:

```
a=0;i=3;j=5;
if (i==3 && j!=5) then a=5;
else {
if (j<5 && i>3) then a=4;
else a=1;
}
```

11. Написати програму на асемблері в якій реалізувати фрагмент програми на мові Сі результат вивести в консоль:

```
a=0;i=3;j=5;
if (i==3 || j!=5) then a=5;
else {
if (j<5 || i>3) then a=4;
else a=1;
}
```

12. Написати програму на асемблері в якій реалізувати фрагмент програми на мові Сі результат вивести в консоль:

```
char ch='b';
switch (ch){
  case='a':
      putchar('a');
      break;
  case='b':
      putchar('b');
      break;
  default:
      putchar('x');
}
```

13. Написати програму на асемблері в якій реалізувати фрагмент програми на мові Сі результат вивести в консоль:

```
int a[5];
for(i=0;i<5;i++) {
  a[i]=i;
}
```

14. Написати програму на асемблері в якій реалізувати фрагмент програми на мові Сі результат вивести в консоль:

```
int a[5];
for(i=0;i<5;i++) {
  a[i]=i;
if (i==3) break;
}
```

15. Написати програму на асемблері в якій реалізувати фрагмент програми на мові Сі результат вивести в консоль:

```
int a[5];
for(i=0;i<5;i++) {
```

```
  a[i]=i;
if (i==3) continue;
}
```

Примітка. Номер варіанта завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

1. Назва групи, П.І.Б. студента, завдання до роботи.
2. Короткий опис теоретичної частини.
3. Блок схема алгоритму програми виконана згідно стандартів.
4. Текст програми із поясненнями.
5. Роздруки екранів стану пам’яті і регістрів з результатами виконання програми у консолі або налагоджувачі `KDbg/dbg`.

Приклади для самостійної роботи

1. 4-1.asm – програма для порівняння двох чисел (jump)

```
extern printf
section .data
number1 dq 42
number2 dq 41
fmt1 db "NUMBER1 > = NUMBER2",10,0
fmt2 db "NUMBER1 < NUMBER2",10,0
section .bss
section .text
global main
main:
push rbp
mov rbp,rsp
mov rax, [number1] ; переслати числа у регістри
mov rbx, [number2]
cmp rax,rbx ; порівняти rax і rbx
jge greater ; якщо rax більше або дорівнює перейти на greater:
mov rdi,fmt2 ; якщо rax менше, продовжити тут
mov rax,0 ; xmm не використовується
call printf ; вивести fmt2
jmp exit ; перейти на позначку exit:
greater:
mov rdi,fmt1 ; rax є більше
mov rax,0 ; xmm не використовується
call printf ; вивести  fmt1
exit:
mov rsp,rbp
pop rbp
ret
```

```
./asm_ld.sh 4_1.asm
./4_1
NUMBER1 > = NUMBER2
```

2. 4_2.asm - програма для обчислення суми чисел від 0 до 5 (jumploop)

```
extern printf
section .data
```

```
number dq 5
fmt db "The sum from 0 to %ld is %ld",10,0
section .bss
section .text
global main
main:
push rbp
mov rbp, rsp
mov rbx,0 ; лічильник
mov rax,0 ; сума буде в rax
jloop:
add rax, rbx
inc rbx
cmp rbx,[number] ; чи досягнуто number?
jle jloop ; number ще не досягнуто, loop
; number досягнуто, продовжувати тут
mov rdi,fmt ; підготовка до виведення
mov rsi, [number]
mov rdx,rax
mov rax,0
call printf
mov rsp,rbp
pop rbp
ret
```

```
./asm_ld.sh 4_2.asm
./4_2
The sum from 0 to 5 is 15
```

```
; 4_3.asm – сума чисел від 0 до 5 (покращений цикл, betterloop)
extern printf
section .data
number dq 5
fmt db "The sum from 0 to %ld is %ld",10,0
section .bss
section .text
global main
main:
push rbp
mov rbp,rsp
mov rcx,[number] ; ініціалізація rcx значенням number
mov rax, 0
bloop:
add rax,rcx ; додати rcx до sum
loop bloop ; цикл із зменшенням поки rcx досягне 1
; поки rcx = 0
mov rdi,fmt ; rcx = 0, продовжувати тут
mov rsi, [number] ; sum буде виведена
mov rdx, rax
mov rax,0 ; без плаваючої крапки
call printf ; виведення
mov rsp,rbp
pop rbp
ret
```

```
./asm_ld.sh 4_3.asm
./4_3
The sum from 0 to 5 is 15
```

```
;4_4.asm – робота з бітами
extern printb
extern printf
section .data
    msg1 db "No bits are set:",10,0
    msg2 db 10,"Set bit #4, that is the 5th bit:",10,0
    msg3 db 10,"Set bit #7, that is the 8th bit:",10,0
    msg4 db 10,"Set bit #8, that is the 9th bit:",10,0
    msg5 db 10,"Set bit #61, that is the 62nd bit:",10,0
    msg6 db 10,"Clear bit #8, that is the 9th bit:",10,0
    msg7 db 10,"Test bit #61, and display rdi",10,0
    bitflags    dq  0
section .bss
section .text
    global main
main:
    mov rbp, rsp; для коректного налагодження
    push rbp
    mov rbp,rsp
    mov rdi, msg1
    xor rax,rax
    call printf

    mov rdi, [bitflags]
    call printb

;set bit 4 (=5th bit)
    mov rdi, msg2
    xor rax,rax
    call printf

    bts qword [bitflags],4  ; встановити біт 4
    mov rdi, [bitflags]
    call printb

;set bit 7 (=8th bit)
    mov rdi, msg3
    xor rax,rax
    call printf

    bts qword [bitflags],7  ; встановити біт 7
    mov rdi, [bitflags]
    call printb

;set bit 8 (=9th bit)
    mov rdi,msg4
    xor rax,rax
    call printf

    bts qword [bitflags],8  ; встановити біт 8
    mov rdi, [bitflags]
```

```
    call printb

;set bit 61 (=62nd bit)
    mov rdi, msg5
    xor rax,rax
    call printf

    bts qword [bitflags],61  ; встановити біт 61
    mov rdi, [bitflags]
    call printb

;clear bit 8 (=9th bit)
    mov rdi, msg6
    xor rax,rax
    call printf

    btr qword [bitflags],8  ; очистити біт 8
    mov rdi, [bitflags]
    call printb

    btr qword [bitflags],8  ; очистити біт 8
    mov rdi, [bitflags]
    call printb

; тестувати біт 61 (буде встановлено прапор CF якщо 1)
    mov rdi, msg7
    xor rax,rax
    call printf

    xor rdi,rdi
    mov rax,61          ; біт 61 буде тестуватися
    xor rdx,rdx         ; переконатися, що всі біти є 0
    bt  [bitflags],rax  ; тест біту
    setc    dil    ; встановити dil (=low rdi) у 1 якщо CF є встановленим
    call printb    ; вивести rdi

leave
ret

// printb.c
#include <stdio.h>

void printb(long long n){
        long long s,c;
        for (c = 63; c >= 0; c--)
        {
                s = n >> c;
                // простір після кожного 8th біту
                if ((c+1) % 8 == 0) printf(" ");

                if (s & 1)
                        printf("1");
                else
                        printf("0");
```

```
        }
        printf("\n");
}
```

makefile (відступи 2, 4, 6 рядків зробити за допомогою Tab):

```
4_4 : 4_4.o printb.o
    gcc -g -o 4_4 4_4.o printb.o -no-pie
4_4.o :
    nasm -f elf64 -g -F dwarf 4_4.asm -l 4_4.lst
printb : printb.c
    gcc -c printb.c
```

$ make
$./4_4

```
No bits are set:
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Set bit #4, that is the 5th bit:
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00010000

Set bit #7, that is the 8th bit:
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 10010000

Set bit #8, that is the 9th bit:
 00000000 00000000 00000000 00000000 00000000 00000000 00000001 10010000

Set bit #61, that is the 62nd bit:
 00100000 00000000 00000000 00000000 00000000 00000000 00000001 10010000

Clear bit #8, that is the 9th bit:
 00100000 00000000 00000000 00000000 00000000 00000000 00000000 10010000

Test bit #61, and display rdi
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
```

5. 4_5.asm - команди зсуву бітів

```
extern printf
section .data

     msgn1     db   "Number 1 is = %d",0
     msgn2     db   "Number 2 is = %d",0
     msg1 db   "SHL 2 = OK multiply by 4",0
     msg2 db   "SHR 2 = wrong divide by 4",0
     msg3 db   "SAL 2 = correctly multiply by 4",0
     msg4 db   "SAR 2 = correctly divide by 4",0
     msg5 db   "SHR 2 = OK divide by 4",0
number1 dq 8
number2 dq  -8
     result  dq   0

section .bss
section .text
     global main
```

```
main:
     push rbp
     mov rbp,rsp
;SHL
;positive number
        mov rsi, msg1
        call printmsg       ;друк heading
        mov rsi, [number1]
     call printnbr      ;друк number1
      mov rax,[number1]
        shl rax,2            ;множення на 4 (логічне)
        mov rsi, rax
     call printres
;negative number
        mov rsi, msg1
        call printmsg       ;друк heading
        mov rsi, [number2]
        call printnbr       ;друк number2
        mov rax,[number2]
        shl rax,2            ;множення на 4 (логічне)
        mov rsi, rax
     call printres
;SAL
;positive number
        mov rsi, msg3
        call printmsg       ;друк heading
        mov rsi, [number1]
     call printnbr      ;друк number1
        mov rax,[number1]
        sal rax,2            ;множення на 4 (арифметичне)
        mov rsi, rax
     call printres
;negative number
        mov rsi, msg3
        call printmsg       ;друк heading
        mov rsi, [number2]
        call printnbr       ;друк number2
        mov rax,[number2]
        sal rax,2           ;множення на 4 (арифметичне)
        mov rsi, rax
     call printres
;SHR
;positive number
        mov rsi, msg5
        call printmsg       ;друк heading
        mov rsi, [number1]
     call printnbr      ;друк number1
        mov rax,[number1]
        shr rax,2            ;ділення на 4 (логічне)
        mov rsi, rax
     call printres
;negative number
        mov rsi, msg2
        call printmsg       ;друк heading
        mov rsi, [number2]
```

```
        call printnbr        ;друк number2
        mov rax,[number2]
        shr rax,2            ;ділення на 4 (логічне)
        mov [result], rax
        mov rsi, rax
     call printres
;SAR
;positive number
        mov rsi, msg4
        call printmsg        ;друк heading
        mov rsi, [number1]
     call printnbr          ;друк number1
        mov rax,[number1]
        sar rax,2           ;ділення на 4 (арифметичне)
        mov rsi, rax
     call printres
;negative number
        mov     rsi, msg4
        call printmsg        ;друк heading
        mov rsi, [number2]
        call printnbr        ;друк number2
        mov rax,[number2]
        sar rax,2           ;ділення на 4 (арифметичне)
        mov rsi, rax
     call printres
leave
ret
printmsg:
     section .data
               .fmtstr   db   10,"%s",10,0 ;format for a string
     section .text
          mov  rdi,.fmtstr
          mov  rax,0
          call      printf
     ret
printnbr:
     section .data
     .fmtstr   db   "The original number is %lld",10,0 ;format for an int
     section .text
          mov  rdi,.fmtstr
          mov  rax,0
          call      printf
     ret
printres:
     section .data
     .fmtstr   db   "The resulting number is %lld",10,0 ;format for an int
     section .text
          mov  rdi,.fmtstr
          mov  rax,0
          call      printf
     ret
```

makefile (відступи 2, 4 рядків зробити за допомогою Tab):

```
4_5: 4_5.o
     gcc -g -o 4_5 4_5.o -no-pie
```

```
4_5.o: 4_5.asm
     nasm -f elf64 -g -F dwarf 4_5.asm -l 4_5.lst

$ make
$ ./4_5

SHL 2 = OK multiply by 4
The original number is 8
The resulting number is 32

SHL 2 = OK multiply by 4
The original number is -8
The resulting number is -32

SAL 2 = correctly multiply by 4
The original number is 8
The resulting number is 32

SAL 2 = correctly multiply by 4
The original number is -8
The resulting number is -32

SHR 2 = OK divide by 4
The original number is 8
The resulting number is 2

SHR 2 = wrong divide by 4
The original number is -8
The resulting number is 4611686018427387902

SAR 2 = correctly divide by 4
The original number is 8
The resulting number is 2

SAR 2 = correctly divide by 4
The original number is -8
The resulting number is -2
```

Лабораторна робота 5

Цілочислові арифметичні команди

Мета роботи: вивчення команди пересилання даних, обчислення виконавчої адреси, цілочислових арифметичних команд та ланцюгових команд.

Теоретичні відомості

1. Команда `mov`

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

```
mov отримувач, джерело
mov eax, 100 ; eax = 0x00000064
mov rcx, -1  ; rcx = 0xffffffffffffffff
mov ecx, eax ; ecx = 0x00000064

mov al, 50
mov rbx, 0  ; занулення регістру перед пересиланням значення меншого розміру
mov bl, al
```

Приклади використання команди `mov`:

```
section .data
    count dd 0
    var1  dd 0x72
    var2  dd 0x01, 0x02, 0x03
var3  dq 0x04, 0x05, 0x06
section .text
global main
main:
   mov bl,ch        ; операнди 8-біт
   add di,ax        ; операнди 16-біт
   mov  eax,0x48    ; операнди 32-бітові, переслати число 0x48 в eax
   mov  eax,var1    ; скопіювати в eax адресу позначки var1
   mov  eax,[var1]  ; скопіювати в eax значення за адресою var1, тобто 0x72
   mov  eax,[var2+4]     ; за адресою var2 міститься значення 0x01 типу dword
                          ; розміром 4 байта, отже за адресою var2 + 4  число
0x02
   mov  ecx, 1           ; переслати в ecx число 1
   mov  eax,[var2+ecx*4] ; скопіювати в eax перший (нумерація з нуля) елемент
0x02
                              ; масиву var2, використовуючи ecx як індексний
регістр
   mov  rbx,var3         ; помістити в rbx адресу масиву var3
   mov  [count],eax      ; скопіювати дані з рег. eax в область пам'яті count
   mov [var1],byte 'G'   ; скопіювати у комірку пам'яті var1 байтовий символ
'G'
   mov  rax,[ebx+ecx+16] ; скопіювати в rax значення з адреси ebx+ecx+16
   mov  rax,[rbx+8]       ; скопіювати в eax значення за адресою ebx+8, тобто
0x05
   mov  rax, var3+16     ; скопіювати в rax адресу третього елементу
   mov  [0x01c615], rax  ; скопіювати за адресою 01c615 вміст eax
```

Приклади запису у буфер масиву однобайтних чисел:

```
section .bss
   array resb 256 ;  неініціалізований масив розміром 256 байт
section .text
; ...
   mov rcx, 256     ; кількість елементів масиву -> у лічильник rcx
   mov rdi, array   ; адреса масиву array в EDI
   mov al, '@'      ; символ заповнювач в AL
again: mov [rdi],al ; занесення коду в черговий елемент масиву
   inc rdi          ; збільшення адреси масиву
   dec rcx ;        ; зменшення лічильника елементів
jnz again           ; якщо не нуль, то повторення циклу
```

2. Команди xchg

Команда обміну значеннями двох операндів operand1 і operand2Ж

```
xchg operand1,operand2
```

xchg ax,r16	xchg r8, r/m8
xchg r16, ax	xchg r8*, r/m8*
xchg eax, r32	xchg r/m16, r16
xchg rax, r64	xchg r16, r/m16
xchg r32, eax	xchg r/m32, r32
xchg r64, rax	xchg r/m64, r64
xchg r/m8, r8	xchg r32, r/m32
xchg r/m8*, r8*	xchg r64, r/m64

***Примітка.**

`r/m8` – байтовий операнд, який може бути байтом регістра загального призначення (`al, cl, dl, bl, ah, ch, dh, bh, bpl, spl, dil and sil`) або байтом пам'яті. Байтові регістри `r8l - r15l` доступні у командах з префіксом REX у 64-бітовому режимі.

У 64-бітовому режимі `r/m8` не можуть використовувати `ah, bh, ch, dh`. Доступ до `до ah, bh, ch, dh` можливий у команд без REX префікса.

3. Команди додавання `add` і віднімання `sub`

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

Принцип роботи:

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

Варіанти команд:

```
  add al,imm8   ; додати imm8 до al
  add ax,imm16  ; додати imm16 до ax
  add eax,imm32 ; додати imm32 до eax
  add rax,imm32 ; додати imm32 з розширенням знаку до 64-біт до rax
  add r/m8, imm8 ; додати imm8 до r/m8.
```

```
add r/m8*, imm8 ; додати imm8 з розширенням знаку до r/m64.
add r/m16, imm16 ; додати imm16 до r/m16
add r/m32, imm32 ; додати imm32 до r/m32.
add r/m64, imm32 ; додати imm32 з розширенням знаку до 64-біт до r/m64.
add r/m16, imm8  ; додати imm8  з розширенням знаку до r/m16.
add r/m32, imm8  ; додати add imm8 з розширенням знаку до r/m32
add r/m64, imm8  ; додати add imm8 з розширенням знаку до r/m64.
add r/m8, r8     ; додати r8 до r/m8.
add r/m8, r8     ; додати r8 до r/m8
add r/m16, r16   ; додати r16 до r/m16
add r/m32, r32   ; додати r32 до r/m32
add r/m64, r64   ; додати r64 до r/m64
add r8, r/m8     ; додати r/m8 до r8
add r8, r/m8     ; додати r/m8 до r8.
add r16, r/m16   ; додати r/m16 до r16.
add r32, r/m32   ; додати r/m32 до r32
add r64, r/m64   ; додати r/m64 до r64
```

```
add cx, word [wVvar]
add rax, 42
add dword [dVar], eax
add qword [qVar], 300
```

Примітка. Особливості команд для роботи байтовими розмірами:

```
mov r/m8,r8
mov r8,r/m8
mov r8,imm8
mov moffs8,al
mov r/m8,imm8
```

У 64-розрядному режимі операнди `r/m8` не можуть бути використані у командах з префіксом REX до наступних регістрів: `ah, bh, ch, dh`. У цьому випадку в операндах `r/m8` старші байти `ah, bh, ch, dh` перевизначені як `dil, sil, bpl, spl`.

Команди з префіксом REX мають доступ до всіх 64-розрядних регістрів, до регістрів `r8-r15`, до регістрів `dil, sil, bpl, spl`.

Приклади арифметичних команд:

```
add rdx,12        ; збільшити на 12 вміст рег. rdx
add rax,ebx       ; до значення рег. rax додати значення rbx
add qword [x],12  ; до значення 8-ми байтової комірки x додати 12
sub [x],rcx       ; від значення 8-ми байтової комірки x відняти значення rcx
```

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 прапору

OF для беззнакових чисел (результат не помістився в розмір операнда або отримано негативний). Для знакових чисел прапор CF не має змісту.

Наявність прапора перенесення 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 128-бітові числа, перше записано в пару регістрів rdx:rax, а друге – в rbx:rcx. Тоді додати ці два числа можна командами

```
add rax, rcx ; додавання молодших частин
adc rdx, rbx ; додавання старших частин з врахуванням перенесення
```

якщо потрібно їх відняти то використовуються команди

```
sub rax, rcx ; віднімання молодших частин
sbb rbx, rcx ; віднімання старших частин з врахуванням позики
```

Додавання 64-розрядних чисел на 32-розрядних регістрах (порівняння мов С і асемблера):

```
long long f1(long long a, long long b) {
   long long c;
   c = a + b;
   return c;
}

; початок функції пропущений
   mov  eax, dword [ebp+16] ; (1)
   mov  edx, dword [ebp+20] ; (2)
   add  eax, dword [ebp+8]  ; (3)
   adc  edx, dword [ebp+12] ; (4)
; кінець функції пропущени1
```

Віднімання 64-розрядних чисел на 32-розрядних регістрах (на мові С і реалізація на асемблері):

```
long long f1(long long a, long long b) {
   long long c;
   c = a - b;
   return c;
}

; початок функції пропущений
   mov  eax, dword [ebp+16] ; (1)
   mov  edx, dword [ebp+20] ; (2)
   sub  eax, dword [ebp+8]  ; (3)
   sbb  edx, dword [ebp+12] ; (4)
; кінець функції пропущений
```

3.1. Команди інкременту, декременту

Принцип роботи команд *інкременту* inc, *декременту* dec:

```
- inc операнд  ; збільшує операнд на 1.
- dec операнд  ; зменшує операнд на 1.
```

```
inc word [wVvar]
inc rax
inc dword [dVar]
inc qword [qVar]
```

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

```
dec rax
inc qword [count]   ; необхідно вказати розмір операнда
```

Приклад організації циклу з використанням команди декременту:

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

3.2. Команди інвертування і зміни знаку

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.

```
not rax   ; інверсія бітів регістра rax
neg al    ; зміна знаку значення в регістрах
neg dx
neg ecx
neg rcx
neg byte [ax]   ; зміна знаку значення у комірці пам’яті [ax] довжиною byte
neg word [dx]   ; зміна знаку значення у комірці пам’яті [dx] довжиною word
neg dword [ecx] ; зміна знаку значення у комірці пам’яті [ecx] довжиною dword
neg qword [rcx] ; зміна знаку значення у комірці пам’яті [rcx] довжиною qword
```

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

```
section .text
global main
main:
  mov rcx,0
  mov cl,0000_0101b   ;   5
  not cx ; 1111_1010b -> 250
  ;neg cx  ; 1111_1011b -> 251 -> -5
 mov rax,60  ; 60-exit
 mov rdi,0   ; 0-код завершення програми
syscall
echo $?
250
```

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

```
mov rax,42
neg rax
add rax,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	-32 768	8000h	32767	7FFFh
32	−2 147 483 648 (-2^{31})	8000 0000h	2 147 483 647 (2^{31}-1)	7FFFFFFFh
64	256 Тбайт адрес (-2^{48})	8000 0000 0000 0000h	256 Тбайт - 1 (адрес) (-2^{48})-1	7FFF FFFF FFFF FFFFh

4. Команди збільшення розрядності із поширенням знаку або нуля

Команда збільшення розрядності числа із поширення знакового біту `movsx`:

```
movsx reg16, reg8/mem8         ; 8-й біт розширюється  до 16-го біту
movsx reg32, reg16/reg8        ; 16-й біт розширюється  до 32-го біту
movsx reg64, reg32/reg16/reg8 ; 32-й біт розширюється до 64-го біту
```

Приклад копіювання знакового числа без і з поширенням знаку:

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

Команда збільшення розрядності числа із поширення нуля `movzx`:

```
mov al,0x7F
movzx ebx,al   ; ebx = 0x0000007F
movsx ebx,al   ; ebx = 0x0000007F

mov al,0x80
movzx ebx,al   ; ebx = 0x00000080
movsx ebx,al   ; ebx = 0xFFFFFF80
```

5. Команда порівняння операндів

Команда `cmp arg1, arg2` (від слова “compare” – “порівняти”) порівнює значення двох операндів. Команда здійснює такі ж обчислення, як і команда `sub`, але значення змінних не

міняються. В результаті виконання команди встановлюються прапори, звичайно за командою слідує команда умовного переходу, наприклад `je`.

`mov ax,5`
`mov bx,8`
`cmp ax,bx`
`"ZF = 0", "CF = 1"`
	`mov ax,8`
	`mov bx,5`
	`cmp ax,bx`
	`"ZF = 0", "CF = 0"`
		`mov ax,5`
		`mov bx,5`
		`cmp ax,bx`
		`"ZF = 1", "CF = 0"`

6. Команди множення і ділення

Команди цілочислового *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, яким 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. 2 показано розміщення неявного операнду і результату операцій цілочислового 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/mem8	ax	dx:ax	dx:ax	ax	dx
32	reg32/mem32	eax	edx:eax	edx:eax	eax	edx
64	reg64/mem64	rax	rdx:rax	rdx:rax	rax	rdx

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:

```
section .text
global main
main:
   mov rax,0
   mov al,5
   mov bl,6
   mul bl   ; ax=5x6=30
   mov rcx,rax
mov rax,60  ; 60-exit
mov rdi,rcx   ; 0-код завершення програми
syscall
echo $?
30
```

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 264 – 256 = 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.

Множення 64-розрядних чисел на 32-розрядних регістрах (порівняння мов С і асемблера):

```
long long f1(long long a, long long b) {
   long long c;
   c = a * b;
   return c;
}

global f2
f2:
   push ebp
   mov  ebp, esp
   push ebx
   mov  eax, dword [ebp+8]
   mov  edx, dword [ebp+16]
   mov  ecx, dword [ebp+20]
   mov  ebx, dword [ebp+12]
   imul ecx, eax
   imul ebx, edx
   mul  edx
   add  ecx, ebx
```

```
add  edx, ecx
pop  ebx
pop  ebp
ret
```

7. 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 dq 0x0000072
segment .text
        lea rax,var     ; аналогічно mov rax, var
...
        lea rax, rsp+8 ; помістить в rax адресу попереднього елемента в стеку
```

Команду `lea` можна використати для виконання деяких арифметичних операцій Вона обчислює адресу свого операнда-джерела і поміщає цю адресу в операнд-отримувач. Адреса операнда формується так

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

Обчислена адреса 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
```

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.

8. Ланцюгові команди

Асемблер має спеціальні команди для роботи з масивами. Їх називають ланцюговими командами або командами для обробки рядків символів. У даному випадку під рядком символів розуміють послідовність байтів. Ланцюгові команди дозволяють виконувати дії над блоками даних розміром байт, слово, подвійне слово, чотирне слово, вісімкове слово. Ланцюгові команди використовують індексні регістри **`rsi`** (індекс джерела, англ. source index) і **`rdi`** (індекс приймача, англ. destination index) для виконання операцій і автоматично збільшують або

зменшують їх значення. Прийнято, що в регістрі `rsi` знаходиться вказівник на наступний необроблений елемент ланцюга-джерела, а в регістрі `rdi` – вказівник на наступний елемент ланцюга-приймача. Напрям переміщення по ланцюгу задається прапором напряму **`df`** у регістрі `rflags`, 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.

8.1. Читання і записування пам'яті

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

Команда `lods` копіює елемент ланцюга-джерела з `[ds:rsi/esi]` в регістр `rax/eax/ax/al`. Після цього значення регістра `rsi/esi` збільшується або зменшується (в залежності від прапора напряму) на значення, яке дорівнює розміру елемента ланцюга.

Команда `stos` записує вміст регістра `rax/eax/ax/al` в ланцюг-приймач `[es:rdi/edi]`. Після цього значення регістра `rdi/edi` збільшується або зменшується (в залежності від прапора напряму) на значення, яке дорівнює розміру елемента ланцюжка.

`loads`

`lodsb` `al<=[ds:esi], esi=esi±1`	`stosb` `[es:edi]<=al edi=edi±1`
`lodsw` `ax<=[ds:esi], esi=esi±2`	`stosw` `[es:edi]<=ax, edi=edi±2`
`lodsd` `eax<=[ds:esi], esi=esi±4`	`stosd` `[es:edi]<=eax, edi=edi±4`
`lodsd` `rax<=[ds:rsi], rsi=rsi±8`	`stosd` `[es:rdi]<=rax, rdi=rdi±8`

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`
`movsq` `qword [es:rdi] => dword [ds:rsi], rsi=rsi±8, rdi=rdi±8`

8.2. Команди `cmps`, `scas` і префікс інструкцій `rep`

Для порівняння і пошуку елементів ланцюгів використовуються команди:

`cmps` – порівняння елемента ланцюга-джерела і ланцюга-приймача.

`scas` – пошук заданого елемента у ланцюгу.

Команда `cmps` порівнює елементи ланцюгів `[ds:rsi/esi] i [es:rdi/edi]`.

Команда `scas` порівнює вміст регістра `rax/eax/ax/al` з елементами, які адресуються регістром [`es:rdx/edi]` (виконує віднімання `rax/eax/ax/al – [es:rdi/edi]`, результат не записується, але прапори встановлюються). Адреса ланцюга має бути поміщена 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], byte [es:edi], esi=esi±1, edi=edi±1
cmpsw compares word [ds:esi], word [es:edi], esi=esi±2, edi=edi±2
cmpsd compares dword [ds:esi], dword [es:edi], esi=esi±4, edi=edi±4
cmpsd compares qword [ds:rsi], qword [es:rdi], rsi=esi±8, rdi=edi±8
scasb compares al, [es:edi], edi=edi±1
scasw compares ax, [es:edi], edi=edi±2
scasd compares eax, [es:edi], edi=edi±4
scasq compares rax, [es:rdi], rdi=rdi±8

Вказані команди обробляють тільки один елемент ланцюга. Для оброблення всіх елементів ланцюга використовуються *префікси повторень* інструкцій:

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

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

Префікси повторень вказуються перед ланцюговою командою. Розміщення префікса `rep` перед ланцюговою 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 ланцюгової команди приймається на основі стану регістра `rcx/ecx/cx` (`repe/repne`) або прапора `zf` (`repz/repnz`).

Так команда `rep stosw` заміняє блок команд:

```
Clear:  mov [es:di],ax ; копіювання AX в es:di
inc di                 ; збільшення індексу на 2 для word у буфері,
inc di
dec cx                 ; зменшення cx на 1
jnz Clear              ; цикл поки cx не 0
```

Префікси 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 ланцюг 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 rsi, block1   ; адреса першого блоку
  mov rdi, block2   ; адреса другого блоку
  mov rcx, size     ; розмір блоку в байтах
  repe cmpsq        ; повторювати поки ZF=1
  je equal          ; якщо ZA=1, блоки однакові
  ; код, якщо блоки не однакові
  jmp onward
equal:
  ; код, якщо блоки однакові
onward:
```

Контрольні запитання.

1. Варіанти використання команди `mov`.
2. Використання команд `xchg, lea`.
3. Команди цілочислової арифметики `add, adc`. Додавання багатобайтових чисел.
4. Команди цілочислової арифметики `sub, sbb`. Віднімання багатобайтових чисел.
5. Команди цілочислової арифметики `inc, dec, neg, not, cmp`. Доповнення до "1" і доповнення до "2" знакового числа.
6. Команди поширення знакового біту і нуля `movsx, movzx`.
7. Команди цілочислової арифметики `mul, imul`
8. Команди цілочислової арифметики `div, idiv`.
9. Додавання і віднімання незапакованих BCD чисел.
10. Додавання і віднімання запакованих BCD чисел.
11. Використання команди `lea` для обчислення арифметичних виразів в операндах.
12. Логічні команди.
13. Команди для роботи з бітами.
14. Команди зсуву.
15. Команди для роботи з пам'яттю `lods, stos`.
16. Команди порівняння і пошуку елементів ланцюгів `cmps scans`.

Практична частина

Завдання.

№	Зміст завдання	Асемблер	Значення			
			a	b	c	d
1	Оголосити і ініціалізувати вектор типу byte значеннями від a до b. Скопіювати з вектора у секцію .bss тільки ті числа, залишок від ділення яких на c дорівнює d.	NASM	1	20	-	-

2	Оголосити і ініціалізувати вектор типу `word` значеннями від `a` до `b`. Обчислити суму елементів вектора. Результат вивести у консоль.	NASM	-5	-10		
3	Оголосити і ініціалізувати вектор типу dword значеннями від a до b. Обчислити добуток парних елементів вектора. Результат вивести у консоль.	NASM	33	45	-	-
4	Оголосити і ініціалізувати вектор чисел типу `qword` значеннями від `a` до `b`. Знайти середнє арифметичне цих елементів і вивести у консоль.	NASM	255	270	-	-
5.	Оголосити неініціалізований вектор типу qword розміром a у секції .bss. Заповнити комірки послідовними числами. Обчислити суми парних чисел і суми непарних чисел. Результат ділення сум вивести у консоль.	NASM	16	-	-	-
6.	Оголосити і ініціалізувати довільними числами типу qword дві матриці розміром a x a у секції .data. Результат множення матриць вивести у консоль.	NASM	2	-	-	-
7.	Оголосити два символьні рядки з довільним текстом розміру a і b типу db. Порівняти два рядки і однакові символи вивести у консоль.	NASM	16	24	-	-
8.	Оголосити два символьні рядки з довільним текстом розміру a типу db. Посимвольно об'єднати два рядки в один рядок. Результати записати у секцію .bss і вивести у консоль.	NASM	4		-	-

9.	Завантажити в регістри ebx:eax число a, а в регістри edx:ecx число b. Додати ці два числа використовуючи команди add, adc.	NASM	0x9876_1234: 0x1872_4321	0x5432_1234 : 0x1432_4321	-	-
10.	Завантажити в регістри ebx:eax число a, а в регістри edx:ecx число b. Відняти ці два числа використовуючи команди sub, sbb.	NASM	0x9486_1235: 0x1862_4321	0x4532_1234 : 0x2132_4321	-	-
11.	Обчислити вираз (a+b)*(c-d) з використанням команди lea.	NASM	-8	6	4	2
12.	Обчислити вираз (a/b)+(c*d) з використанням команди lea.	NASM	12	4	6	-4
13.	Обчислити вираз (a+b)*(c-d) з використанням незапакованих двійково-десяткових чисел	NASM	4	3	5	2
14.	Обчислити вираз (a*b)+(c/d) з використанням незапакованих двійково-десяткових чисел. Результат вивести у консоль.	NASM	3	4	8	4
15.	Написати програму для переставлення командою xchg комірок пам'яті типу word із значеннями a, b, c, d. Вміст переставлених комірок вивести вивести у консоль.	NASM	8	7	6	5

Примітка. Номер варіанта завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

1. Завдання до роботи.
2. Короткий опис теоретичної частини.
3. Текст програми із поясненнями.
5. Роздруки екранів стану пам'яті і регістрів з результатами виконання програми у консолі або налагоджувачі `DDD/KDbg/dbg`.

Приклади для самостійної роботи

1. 1.asm – зміна регістру букв стрічки

```
; ./asm_ld.sh 1.asm
section .data
  var db 'HELLO WORD',0
  len equ $-var
section .text
global main
```

```
main:
  nop           ; для gdb
  mov ebx,var   ; адресу var у ebx
  mov eax,len-1 ; кількість 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 rax,1     ; 1-write
  mov rdi,var
  mov rsi,len
  syscall

  mov rax,60  ; 60-exit
  mov rdi,0   ; 0-код завершення програми
  syscall
```

./1

```
hello@word
```

2. 2.asm – цілочислове множення

```
section .data
msg1 db "Значення ",0
msg2 db "Результат ",0
   radius dd 15
   pif dd 3.14
   pi  dq 3.14
fmtstr db "%s",10,0 ;формат для друку string
fmtflt db "%lf",10,0 ;формат для чисел float
fmtint db "%d",10,0 ;формат для чисел integer
section .bss
section .text
global main
extern printf
main:
  push rbp
  mov rbp,rsp

 ; друк msg1
  mov rax, 0 ; без плаваючої крапки
  mov rdi, fmtstr
  mov rsi, msg1
  call printf

; друк radius
  mov rax, 0 ; без плаваючої крапки
  mov rdi, fmtint
  mov rsi, [radius]
  call printf

  ; множення
  mov rax,0           ; без плаваючої крапки
  mov eax, [radius]   ; множене
```

```
  mov ebx,[pif]       ; множник
  mul ebx             ; результат ax
  mov rsi,rax

; друк msg2
  mov rax, 0 ; без плаваючої крапки
  mov rdi, fmtstr
  mov rsi, msg2
  call printf

; друк результат
  mov rax,0
  mov rdi, fmtflt
  call printf

; друк pi
  mov rax, 1 ; використовується xmm регістр
  movq xmm0, [pi]
  mov rdi, fmtflt
  call printf

  mov rsp,rbp
  pop rbp
  ret
```

```
  $ ./2
Значення
15
Результат
0.000000
3.140000
```

3. 3.asm команда lodsb

```
section .data
s_addr  db 'a','b','c','d','e',1
t_addr  db 'A','B','C','D','F',0
section .bss
section .text
global main
main:
        mov   esi, s_addr
        mov   edi, t_addr
next:   lodsb                       ; mov al, [esi]; inc esi
        and   al, 0xdf              ; перетворити source у верхній регістр
        scasb                       ; cmp al, [edi]; inc edi
        je    next
done:


mov rax, 60 ; 60 = exit
mov rdi, 0 ; 0 = код успішного виходу
syscall ; quit
```

4. 4.asm команда movsq

```
section .data
SIZE equ 8
s_addr  dq '01234567'
t_addr  dq '00000000'
section .bss
section .text
global main
main:
        mov   rsi, s_addr
        mov   rdi, t_addr
        mov   cl,SIZE

.next:   movsq              ; [s_addr] => t_addr] inc rsi, rdi
        dec cl
        jnz .next

mov rax, 60 ; 60 = exit
mov rdi, 0 ; 0 = код успішного виходу
syscall ; quit
```

5. 5.asm команда cmpsq

```
section .data
SIZE equ 8
s_addr  dq '01234567'
t_addr  dq '00000500'
section .bss
section .text
global main
main:
        mov   rsi, s_addr
        mov   rdi, t_addr
        mov   cl,SIZE
next:   cmpsq          ; [s_addr] == t_addr] inc rsi, rdi
        dec cl
        jnz next

mov rax, 60 ; 60 = exit
mov rdi, 0 ; 0 = код успішного виходу
syscall ; quit
```

6. 6.asm команда scasq

```
;test4.asm
section .data
SIZE equ 8
s_addr  dq '01234567'
t_addr  dq '00000500'
section .bss
section .text
global main
main:
        mov   rsi, s_addr
        mov   rdi, t_addr
        mov   cl,SIZE
        rep scasq          ; [s_addr] == t_addr] inc rsi, rdi
```

```
mov rax, 60 ; 60 = exit
mov rdi, 0 ; 0 = код успішного виходу
syscall ; quit
```

7. 7.asm – обчислення суми квадратів чисел від 1 до n

```
; ************************************************
; Data declarations
section .data
; -----
; визначення констант
SUCCESS equ 0   ; успішне завершення
SYS_exit equ 60 ; код завершення програми
; Define Data.
n dd 10
sumOfSquares dq 0
; ************************************************************
section .text
global main
extern printf
main:
   push rbp
   mov rbp,rsp
; обчислення суми квадратів чисел від 1 до n (включно).
; подібно до:
; for (i=1; i<=n; i++)
; sumOfSquares += i^2;
   mov rbx, 1 ; i
   mov ecx, dword [n]
sumLoop:
   mov rax, rbx ; get i
   mul rax ; i^2
   add qword [sumOfSquares], rax
   inc rbx
loop sumLoop

; друк результату
   mov rax,0
   mov rdi, fmtint
   mov rsi, [sumOfSquares]
   call printf

; завершення програми
last:
mov rax, SYS_exit ; call code for exit
mov rdi, SUCCESS ; exit з кодом  success
syscall
```

$ ./7
385

8. Арифметичні операції над байтами

```
; macro
extern printf

%ifmacro print 2
```

```
    %error "print 2" помилка виклику макросу
%else
%macro print 2
     mov rax,0   ; не число з плаваючою крапкою
     mov rsi,%1  ; 1-й параметр число або str
     mov rdi,%2  ; 2-й параметр формат
     call printf
%endmacro
%endif

section .data
x     db 0
sFmt  db "%s",0
bFmt  db "%u",10,0
sAns1 db "sAns1=",0
sAns2 db "sAns2=",0
sRem2 db "sRem2=",0
sAns3 db "sAns3=",0

bNumA db 63
bNumB db 17
bNumC db 5

bAns1 db 0
bAns2 db 0
bRem2 db 0
bAns3 db 0
section .text
global main
main:
nop
push rbp
mov rbp,rsp

; Операції над беззнаковими байтами
; bAns1 = bNumA / bNumB
mov al, byte [bNumA]
mov ah,0
mov bl,3
div bl   ; al = ax/3
mov byte [bAns1],al
print sAns1,sFmt
print [bAns1],bFmt


; bAns2 = bNumA / bNumB (unsigned)
mov ax,0
mov al, byte [bNumA]
div byte [bNumB]
mov byte [bAns2],al   ;al=ax / bNumB
;shr ax,8
mov byte [bRem2],ah   ;ah=ax % bNumb

print sAns2,sFmt
print [bAns2],bFmt
```

```
print sRem2,sFmt
print [bRem2],bFmt

; bAns3 = (bNumA * bNumC) / bNumB (unsigned)
mov rax,0
mov al, byte [bNumA]
mul byte [bNumC] ; result in ax
div byte [bNumB] ; al = ax / bNumB
mov byte [bAns3], al
print sAns3,sFmt
print [bAns3],bFmt

mov rsp,rbp
pop rbp

mov rax,60  ; 60-exit
mov rdi,0   ; 0-код завершення програми
syscall
```

$ ./8

```
sAns1=21
sAns2=3075 ?
sRem2=12
sAns3=18
```

9. Арифметичні операції над словами

```
; macro
extern printf

%ifmacro print 2
    %error "print 2" помилка виклику макросу
%else
%macro print 2
     mov rdi,0
     mov rsi,0
     mov rax,0   ; не число з плаваючою крапкою
     mov rdi,%2  ; 2-й параметр формат
     mov rsi,%1  ; 1-й параметр str або число
     call printf
%endmacro
%endif

section .data
xxx  db 0,0
sFmt db "%s",0
wFmt dw "%d",10,0

wNumA dw 4321
wNumB dw 1234
wNumC dw 167
wAns1 dw 0
wAns2 dw 0
wAns3 dw 0
wRem2 dw 0
```

```
swNumA db "swNumA=",0
swNumB db "swNumB=",0
swNumC db "swNumC=",0
swAns1 db "swAns1=",0
swAns2 db "swAns2=",0
swAns3 db "swAns3=",0
swRem2 db "swRem2=",0

section .text
global main
main:

push rbp
mov rbp,rsp

; Операції над беззнаковими словами
; wAns1 = wNumA / 5 (unsigned)
mov ax, word [wNumA]
mov dx, 0
mov bx, 5
div bx ; ax = dx:ax / 5
mov word [wAns1], ax
print swAns1,sFmt
print [wAns1],wFmt

; wAns2 = wNumA / wNumB (unsigned)
mov dx,0
mov ax, word [wNumA]
div word [wNumB] ; ax = dx:ax / wNumB
mov word [wAns2], ax
mov word [wRem2], dx
print swAns2,sFmt
print [wAns2],wFmt
print swRem2,sFmt
print [wRem2],wFmt

; wAns3 = (wNumA * wNumC) / wNumB (unsigned)
mov ax, word [wNumA]
mul word [wNumC] ; результат в dx:ax
div word [wNumB] ; ax = dx:ax / wNumB
mov word [wAns3], ax
print swAns3,sFmt
print [wAns3],wFmt

mov rsp,rbp
pop rbp

mov rax,60  ; 60-exit
mov rdi,0   ; 0-код завершення програми
syscall
```

$ ./9

```
swAns1=864
swAns2=3
swRem2=2004025963 ?
```

```
SwAns3=40567368 ?
```

10. Арифметичні операції над подвійним словами

```
; macro
extern printf

%ifmacro print 2
    %error "print 2" помилка виклику макросу
%else
%macro print 2
     mov rdi,0
     mov rsi,0
     mov rax,0   ; не число з плаваючою крапкою
     mov rdi,%2  ; 2-й параметр формат
     mov rsi,%1  ; 1-й параметр str або число
     call printf
%endmacro
%endif

section .data
xxx  db 0,0
sFmt db "%s",0
dFmt db "%d",10,0

dNumA dd 42000
dNumB dd -3157
dNumC dd -293
dAns1 dd 0
dAns2 dd 0
dAns3 dd 0
dRem2 dd 0

sdNumA db "sdNumA=",0
sdNumB db "sdNumB=",0
sdNumC db "sdNumC=",0
sdAns1 db "sdAns1=",0
sdAns2 db "sdAns2=",0
sdAns3 db "sdAns3=",0
sdRem2 db "sdRem2=",0

section .text
global main
main:

push rbp
mov rbp,rsp

; Операції над знаковими подвійними словами
; dAns1 = dNumA / 7 (signed)
mov eax,dword [dNumA]
cdq   ; eax => edx:eax
mov ebx,7
idiv ebx ; eax = dx:eax / 7
mov dword [dAns1], eax
print sdAns1,sFmt
```

```
print [dAns1],dFmt

; wAns2 = wNumA / wNumB (signed)
mov eax, dword [dNumA]
cdq  ; eax => edx:eax
idiv dword [dNumB] ; eax = edx:ex / dNumB
mov dword [dAns2], eax  ; eax = edx:eax / dNumB
mov dword [dRem2], edx  ; edx = edx:eax % dNumB
print sdAns2,sFmt
print [dAns2],dFmt
print sdRem2,sFmt
print [dRem2],dFmt

; wAns3 = (wNumA * wNumC) / wNumB (unsigned)
mov eax, dword [dNumA]
imul dword [dNumC] ; результат у edx:eax
idiv dword [dNumB] ; eax = edx:eax / dNumB
mov dword [dAns3], eax
print sdAns3,sFmt
print [dAns3],dFmt

mov rsp,rbp
pop rbp

mov rax,60  ; 60-exit
mov rdi,0   ; 0-код завершення програми
syscall
```

$ ./10

```
sdAns1=6000
sdAns2=-13
sdRem2=959
sdAns3=3898
```

11. Арифметичні операції над чотирними словами

```
; macro
extern printf

%ifmacro print 2
    %error "print 2" помилка виклику макросу
%else
%macro print 2
     mov rdi,0
     mov rsi,0
     mov rax,0   ; не число з плаваючою крапкою
     mov rdi,%2  ; 2-й параметр формат
     mov rsi,%1  ; 1-й параметр str або число
     call printf
%endmacro
%endif

section .data
xxx  db 0,0
sFmt db "%s",0
qFmt db "%d",10,0
```

```
qNumA dq 730000
qNumB dq -13456
qNumC dq -1279
qAns1 dq 0
qAns2 dq 0
qAns3 dq 0
qRem2 dq 0

sqNumA db "sqNumA=",0
sqNumB db "sqNumB=",0
sqNumC db "sqNumC=",0
sqAns1 db "sqAns1=",0
sqAns2 db "sqAns2=",0
sqAns3 db "sqAns3=",0
sqRem2 db "sqRem2=",0

section .text
global main
main:

push rbp
mov rbp,rsp

; Операції над знаковими чотирними словами
; qAns1 = qNumA / 9 (signed)
mov rax,qword [qNumA]
cqo   ; rax => rdx:rax
mov ebx,9
idiv rbx ; rax = rdx:rax / 9
mov qword [qAns1],rax
print sqAns1,sFmt
print [qAns1],qFmt

; qAns2 = qNumA / qNumB (signed)
mov rax, qword [qNumA]
cqo  ; rax => rdx:rax
idiv qword [qNumB] ; rax = rdx:rax / qNumB
mov qword [qAns2], rax  ; rax = rdx:rax / qNumB
mov qword [qRem2], rdx  ; rdx = rdx:rax % qNumB
print sqAns2,sFmt
print [qAns2],qFmt
print sqRem2,sFmt
print [qRem2],qFmt

; qAns3 = (qNumA * qNumC) / qNumB (signed)
mov rax, qword [qNumA]
imul qword [qNumC] ; результат в in rdx:rax
idiv qword [qNumB] ; rax = rdx:rax / qNumB
mov qword [qAns3],rax
print sqAns3,sFmt
print [qAns3],qFmt

mov rsp,rbp
pop rbp
```

```
mov rax,60  ; 60-exit
mov rdi,0   ; 0-код завершення програми
syscall
```

```
$ ./11
sqAns1=81111
sqAns2=-54
sqRem2=3376
sqAns3=69386
```

12. Команда xchg

```
extern printf
%ifmacro print 2
    %error "print 2" помилка виклику макросу
%else
%macro print 2
     mov rdi,0
     mov rsi,0
     mov rax,0   ; не число з плаваючою крапкою
     mov rdi,%2  ; 2-й параметр формат
     mov rsi,%1  ; 1-й параметр str або число
     call printf
%endmacro
%endif

arr dd 3,4,5
sFmt db "%s",0
dFmt db "%d",10,0
sres db "sres=",0
section .text
global main
main:

mov REX.R8L,1
mov eax,[arr]    ; eax=3
mov ebx,[arr+4]  ; ebx=4
xchg ebx,[arr]   ; arr=4,4,5
xchg eax,[arr+4] ; arr=4,3,5
print sFmt,sres
print [arr],dFmt
print [arr+4],dFmt
print [arr+8],dFmt

mov rax,60  ; 60-exit
mov rdi,0   ; 0-код завершення програми
syscall
```

```
$ ./12
sRes=4
3
5
```

Лабораторна робота 6

Ланцюгові команди

Мета роботи: отримання практичних навичок роботи з ланцюговими командами

Теоретичні відомості

1. Ланцюгові команди

Асемблер має спеціальні команди для роботи з масивами. Їх називають ланцюговими командами або командами для обробки рядків символів. У даному випадку під рядком символів розуміють послідовність байтів. Ланцюгові команди дозволяють виконувати дії над блоками даних розміром байт, слово, подвійне слово, чотирне слово. Ланцюгові команди використовують індексні регістри **rsi** (індекс джерела, англ. source index) і **rdi** (індекс приймача, англ. destination index) для виконання операцій і автоматично збільшують або зменшують їх значення. Прийнято, що в регістрі `rsi` знаходиться вказівник на наступний необроблений елемент ланцюга-джерела, а в регістрі `rdi` – вказівник на наступний елемент ланцюга-приймача. Напрям переміщення по ланцюгу задається прапором напряму **df** у регістрі rflags, 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 ланцюговими командами є читання **lods** і записування **stos** пам'яті.

Команда `lods` копіює елемент ланцюга-джерела з комірки пам'яті `[ds:rsi]` в регістр `rax`. Після цього значення регістра `rsi` збільшується або зменшується (в залежності від прапора напряму) на значення, яке дорівнює розміру елемента ланцюга.

Команда `stos` записує вміст регістра `rax` в ланцюг-приймач `[es:rdi]`. Після цього значення регістра `rdi` збільшується або зменшується (в залежності від прапора напряму) на значення, яке дорівнює розміру елемента ланцюжка.

lodsb al<=[(r)si], (r)si=(r)si±1	stosb (r|e)di<=al, (r|e)di=(r|d)i±1
lodsw ax<=[(r)si], (r)si=(r)si±2	stosw (r|e)di<=ax, (r|e)di=(r|d)i±2
lodsd eax<=[(r)si], (r)si=(r)si±4	stosd (r|e)di<=eax, (r|e)di=(r|e)di±4
lodsq rax<=[(r)si], (r)si=(r)si±8	stosq (r|e)di<=rax, (r|e)di=(r|e)di±8

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

movsb byte [(r|e)si] => byte [(r|e)di], (r|e)si±1, (r|e)di±1
movsw word [(r|e)si] => word [(r|e)di], (r|e)si±2, (r|e)±2
movsd dword [(r|e)si]=> dword [(r|e)di], (r|e)si±4, (r|e)±4
movsq qword [(r|e)si] => dword [(r|e)di], (r|e)si±8, (r|e)±8

3. Команди `cmps`, `scas` і префікс інструкцій `rep`

Для порівняння і пошуку елементів ланцюгів використовуються команди:

`cmps` – порівнює елементи ланцюга-джерела і ланцюга-приймача.

`scas` – шукає заданий елемент у ланцюгу.

Команда `cmps` порівнює елементи ланцюгів `(r|e)si i (r|e)di.`

Команда `scas` порівнює вміст регістра `rax` з елементами за адресою `[es:(e)di]|rdi` (виконує віднімання `rax – [es:(e)di]|rdi`, результат не записується, але прапори встановлюються). Адреса ланцюга має бути поміщена в регістр `(r|e)di`.

cmpsb compares byte (r|e)si, byte (r|e)di
cmpsw compares word (r|e)si, word (r|e)di
cmpsd compares dword (r|e)si, dword (r|e)di
cmpsq compares qword rsi, qword rdi
scasb compares al, [es:(e)di]|rdi
scasw compares ax, [es:(e)di]|rdi
scasd compares eax, [es:(e)di]|rdi
scasq compares rax, rdi|edi

Вказані команди обробляють тільки один елемент ланцюга. Для оброблення всіх елементів ланцюга використовуються *префікси повторень* інструкцій:

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

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

Префікси повторень вказуються перед ланцюговою командою. Розміщення префікса `rep` перед ланцюговою 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 ланцюгової команди приймається на основі стану регістра rcx (repe/repne) або прапора zf (repz/repnz).

Так команда rep stosw заміняє блок команд:

```
Clear:  mov [es:edi],rax ; копіювання AX в es:edi
  add edi,4              ; збільшення індексу на 4 для dword у буфері,
  dec ecx                ; зменшення ecx на 1
jnz Clear                ; цикл поки cx не 0
```

Префікси повторень обробляють різні прапори:

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

- repne (або repe) виконує інструкцію поки регістр rcx не дорівнює 0.

Фрагмент програми порівняння двох блоків пам'яті.

```
segment .text
  cld
  mov rsi, block1   ; адреса першого блоку
  mov rdi, block2   ; адреса другого блоку
  mov rcx, size     ; розмір блоку в байтах
  repe cmpsq        ; повторювати поки ZF=1
  je equal          ; якщо ZA=1, блоки однакові
  ; код, якщо блоки не однакові
  jmp onward
equal:
  ; код, якщо блоки однакові
onward:
```

Контрольні запитання.

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

Практична частина

Завдання.

1. Написати програму, яка копіює символьний рядок "Hello world" із секції даних .data у секцію буферних даних .bss з використанням ланцюгових команд. Виводити на екран копійовані символи.

2. Написати програму, яка копіює послідовність чисел 1,2,3,4,5 типу байт із області пам'яті mem1 в область пам'яті mem2 з використанням ланцюгових команд. Виводити на екран копійовані числа.

3. Написати програму, яка копіює послідовність чисел 1,2,3,4,5 типу word із області пам'яті mem1 в область пам'яті mem2 з використання ланцюгових команд. Виводити на екран копійовані числа.

4. Написати програму, яка копіює послідовність чисел 1,2,3,4,5 типу dword із області пам’яті mem1 в область пам’яті mem2 з використанням ланцюгових команд. Виводити на екран копійовані числа.

5. Написати програму, яка копіює послідовність чисел 1,2,3,4,5 типу qword із області пам’яті mem1 в область пам’яті mem2 з використанням ланцюгових команд. Виводити на екран копійовані числа.

6. Написати програму, яка переписує з масиву однобайтних чисел 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 типу word у буфер спочатку парні числа, а потім непарні без використанням ланцюгових команд. Вивести на екран початковий і переписаний масив.

7. Написати програму, яка переписує масив чисел 8, 1, 6, 2, 4, 3, 1, 2 типу word у буфер у відсортованому за зростанням порядку. Вивести на екран початковий і відсортований масив.

8. Написати програму, яка переписує масив чисел 8, 3, 6, 2, 4, 3, 1, 2 типу word у буфер у відсортованому за спаданням порядку. Вивести на екран початковий і відсортований масив.

9. Написати програму, яка порівнює два масиви ASCII символів і завершує роботу при першому співпадінні підстрічок мінімальної довжини. Вивести на екран співпавшу підстрічку.

10. Написати програму, яка порівнює два масиви ascii символів і виводить на екран не співпавші символи.

11. Написати програму, яка порівнює два масиви ASCII символів і виводить на екран співпавші символи.

12. Написати програму, яка виводить на екран з масиву ASCII символів, символи у яких співпадають молодші 4-біти.

13. Написати програму пошуку і виведення на екран символів “a” і його індексів у стрічці “Students learn low level language” з використання ланцюгових команд..

14. Написати програму пошуку і виведення на екран послідовності співпадаючих пар символів (наприклад "le") і їх індексів у стрічці байтів “Students learn low level language assembler” з використанням ланцюгових команд.

15. Написати програму підрахунку і виведення на екран частотності букв у стрічці “Students learn low level language assembler” з використанням ланцюгових команд.

Примітка. Номер варіанта завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

1. Назва групи, П.І.Б. студента, завдання до роботи.
2. Короткий опис теоретичної частини.
3. Текст програми із поясненнями.
4. Роздруки результатів виконання програми у консолі або налагоджувачі `Kdbg/DDD/dbg`.

3. Завдання для самостійної роботи

1. 1.asm команда lodsb

```
;test1.asm
section .data
s_addr  db 'abcde,10,0
t_addr  db 'ACDEF’,10,0
section .bss
section .text
global main
main:
```

```
        mov esi, s_addr
        mov edi, t_addr
        mov rbx,0
next:   lodsb               ; mov al, [esi]; inc esi
        and   al, 0xdf      ; перетворити source у великі букви
        scasb               ; cmp al, [edi]; inc edi
        inc bx
        je next
done:

mov rax, 60   ; 60 = exit
mov rdi, rbx  ; rbx = число співпадінь
syscall ; quit
```

$ ./1

```
$ echo $?
5
```

2. 2.asm команда movsq

```
; 2.asm
section .data
SIZE equ 8
s_addr dq '01234567'
d_addr dq '00000000'
section .text
global main
main:
        mov rsi, s_addr
        mov rdi, t_addr
        mov cl,SIZE

.next:   movsq  ; [s_addr] => d_addr] inc rsi, rdi
        dec cl
        jnz .next

mov rax, 60 ; 60 = exit
movzx rdi, cl ; 0 = код успішного виходу
syscall ; вихід
```

$ ./2

```
$ echo $?
0
```

3. 3.asm команда cmpsq

```
; 3.asm
section .data
SIZE equ 8
s_addr  dq '01234567'
t_addr  dq '00000500'
section .text
global main
main:
        mov   rsi, s_addr
        mov   rdi, t_addr
        mov   cl,SIZE
```

```
next:   cmpsq         ; [s_addr] == t_addr] inc rsi, rdi
        dec cl
        jnz next

mov rax, 60 ; 60 = exit
movzx rdi, cl ; 0 = код успішного виходу
syscall ; вихід
```

```
$ ./3
$ echo $?
0
```

4. Пересилання стрічок символів командами `movsb, stosb`

```
; 4.asm
; макрос друкування символьних стрічок
%macro prnt 2
    mov     rax, 1 ; 1 = write
    mov     rdi, 1 ; 1 = to stdout
    mov     rsi, %1     ; адреса стрічки
    mov     rdx, %2     ; довжина стрічки
    syscall
     mov rax, 1
     mov rdi, 1
     mov rsi, NL
     mov rdx, 1
    syscall
%endmacro
section .data
     length    equ 95
     NL db 0xa
     string1 db "my_string of ASCII:"
     string2 db 10,"my_string of zeros:"
     string3 db 10,"my_string of ones:"
     string4 db 10,"again my_string of ASCII:"
     string5 db 10,"copy my_string to other_string:"
     string6 db 10,"reverse copy my_string to other_string:"
section .bss
      my_string    resb length
     other_string resb length
section .text
     global main
main:
push rbp
mov  rbp, rsp
;--------------------------------------------------------------------------
;заповнення стрічки друкованими ascii символами
            prnt string1,18
            mov rax,32
            mov rdi,my_string
            mov rcx, length
str_loop1:    mov byte[rdi], al       ; найпростіший метод
            inc rdi
            inc al
            loop str_loop1
            prnt my_string,length
```

```
    ;-------------------------------------------------------------------------
;заповнення стрічки ascii нулями 0
            prnt string2,20
            mov rax,48
            mov rdi,my_string
            mov rcx, length
    str_loop2:    stosb                 ; непотрібна команда inc rdi
            loop str_loop2
            prnt my_string,length
    ;-------------------------------------------------------------------------
    ;заповнення стрічки ascii одиницями 1
            prnt string3,19
            mov rax, 49
            mov rdi,my_string
            mov rcx, length
            rep stosb          ; непотрібні команди inc rdi і loop
            prnt my_string,length
    ;-------------------------------------------------------------------------
    ;заповнення стрічки знову друкованими ascii символами
            prnt string4,26
            mov rax,32
            mov rdi,my_string
            mov rcx, length
    str_loop3:    mov byte[rdi], al       ; the simple method
            inc rdi
            inc al
            loop str_loop3
            prnt my_string,length
    ;-------------------------------------------------------------------------
    ;копіювання стрічки у іншу стрічку
            prnt string5,32
            mov rsi,my_string             ;rsi джерело
            mov rdi,other_string          ;rdi отримувач
            mov rcx, length
            rep movsb
            prnt other_string,length
    ;-------------------------------------------------------------------------
    ;реверсне копіювання стрічки my_string у стрічку other_string
            prnt string6,40
            mov rax, 48             ;очищення стрічки other_string
            mov rdi,other_string
            mov rcx, length
            rep stosb
            lea rsi,[my_string+length-4]
            lea rdi,[other_string+length]
            mov rcx, 27             ;копіювання тільки десяти символів
            std                ;std встановити прапор DF, cld очистити DF
            rep movsb
            prnt other_string,length
    leave
    ret

    $ ./4
    my_string of ASCII
```

```
     !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

     my_string of zeros:
     0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000

     my_string of ones:
     1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111

     again my_string of ASCII:
     !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

     copy my_string to other_string:
     !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

     reverse copy my_string to other_string:
     0000000000000000000000000000000000000000000000000000000000000000000000000000000abcdefgh
ijklmnopqrstuvwxyz
```

5. Порівняння стрічок символів командою cmpsb і сканування командою scasb

```
; 5.asm strings.asm
extern   printf
section .data
     string1   db "This is the 1st string.",10,0
     string2   db "This is the 2nd string.",10,0
     strlen2   equ $-string2-2
     string21  db "Comparing strings: The strings do not differ.",10,0
     string22  db "Comparing strings: The strings differ, "
               db "starting at position: %d.",10,0
     string3   db "The quick brown fox jumps over the lazy dog.",0
     strlen3   equ $-string3-2
     string33  db "Now look at this string: %s",10,0
     string4   db "z",0
     string44  db "The character '%s' was found at position: %d.",10,0
     string45  db "The character '%s' was not found.",10,0
     string46  db "Scanning for the character '%s'.",10,0
section .bss
section .text
     global main
main:
     push rbp
     mov  rbp,rsp
; друк двох strings
     xor rax,rax
     mov rdi, string1
     call printf
     mov rdi, string2
     call printf
; порівняння двох strings
;-----------------------------------------------------
     lea rdi,[string1]
```

```
     lea rsi,[string2]
     mov rdx, strlen2
     call compare1
     cmp rax,0
     jnz not_equal1
;strings однакові, друк
     mov rdi, string21
     call printf
     jmp otherversion
;strings не однакові, друк
not_equal1:
     mov rdi, string22
     mov rsi, rax
     xor rax,rax
     call printf
; порівняння двох strings, інша версія
;-----------------------------------------------------
otherversion:
     lea rdi,[string1]
     lea rsi,[string2]
     mov rdx, strlen2
     call compare2
     cmp rax,0
     jnz not_equal2
;strings однакові, друк
     mov rdi, string21
     call printf
     jmp scanning
;strings не однакові, друк
not_equal2:
     mov rdi, string22
     mov rsi, rax
     xor rax,rax
     call printf
; scan для символу у string
;-----------------------------------------------------
; first print the string
     mov rdi,string33
     mov rsi,string3
     xor rax,rax
     call printf
; тоді друк шуканого аргументу, якщо це один символ
     mov rdi,string46
     mov rsi,string4
     xor rax,rax
     call printf
scanning:
     lea rdi,[string3]  ;    string
     lea rsi,[string4]  ;    шуканий аргумент
     mov rdx, strlen3
     call cscan
     cmp rax,0
     jz char_not_found
;символ знайдений , друк
     mov rdi,string44
```

```
     mov rsi,string4
     mov rdx,rax
     xor rax,rax
     call printf
     jmp exit
;символ не знайдений, друк
char_not_found:
     mov rdi,string45
     mov rsi,string4
     xor rax,rax
     call printf
exit:
leave
ret
; FUNCTIONS
;=====================================================
; функція порівняння двох strings
compare1: mov rcx, rdx
          cld
cmpr:     cmpsb
          jne notequal
          loop cmpr
          xor rax,rax
          ret
notequal: mov rax, strlen2
          dec rcx
          sub rax,rcx   ;обчислення позиції
          ret
          xor rax,rax
          ret
;-----------------------------------------------------
; функція порівняння двох strings
compare2: mov rcx, rdx
          cld
          repe cmpsb
          je equal2
          mov rax, strlen2
          sub rax,rcx   ;обчислення позиції
          ret
equal2:   xor rax,rax
          ret
;-----------------------------------------------------
; функція пошуку символу у string
cscan:    mov rcx, rdx
          lodsb
          cld
          repne scasb
          jne char_notfound
          mov rax, strlen3
          sub rax,rcx   ;обчислення позиції
          ret
char_notfound:xor rax,rax
          ret
```

$ ./5

```
This is the 1st string.
This is the 2nd string.
Comparing strings: The strings differ, starting at position: 13.
Comparing strings: The strings differ, starting at position: 13.
Now look at this string: The quick brown fox jumps over the lazy dog.
Scanning for the character 'z'.
The character 'z' was found at position: 38.
```

Лабораторна робота 7

Директиви, макроси

Мета роботи: вивчення директив препроцесора

Теоретичні відомості

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
```

2. Однорядкові макроси

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
```

Вирази, визначені макросом **`%define`** є регістрозалежними, а **`%idefine`** – регістронезалежними

```
%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 макросу, 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 назначеного 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.

2.2. Макрозмінна змінна %assign (%iassign)

%assign (**%iassign** – регістронечутлива версія) використовується для визначення однорядкових макросів, які не мають параметрів, а мають числове значення. Значення можна задати як вираз, який оцінюється один раз при обробленні передпроцесорної 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
```

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)
```

2.4. Спеціальні символи %?, %??

Спеціальні символи `%?`, `%??` використовуються 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
```

2.5. Директива %strcat

Директива `%strcat` об'єднує символьні рядки і назначає їх однорядковому макросу:

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

2.6. Директива %strlen

Директива `%strlen` назначає довжину стрічки макросу

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

В результаті `charcnt` буде присвоєно значення 9.

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'
```

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
```

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
```

3.2. Локальні позначки у макросах

Багаторядкові макроси можуть містити локальні позначки, які залишаються локальними при кожному виклику макросу: при багатократному виклику макросу позначки кожний раз будуть змінюватися. Така позначка записується за допомогою префіксу `%%`. Nasm створює імена у формі ..@2345.skip, де число 2345 змінюється при кожному виклику. Префікс @ запобігає перетину імен локальних позначок з механізмом звичайних локальних позначок.

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

3.3. Поглинаючі макропараметри

Іноді потрібно визначити макрос, який виділяє декілька параметрів, а решту параметрів об'єднує як один параметр разом із комами. Для цьому після числа, яке задає кількість параметрів ставиться символ '+'. Параметр, який відповідає цьому числу буде поглинаючим. 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 буде поглинаючим і об'єднує решту параметрів – `"hello, world",13,10` (розміщується після `db`).

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.

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. Він може використовуватися як аргумент для %rep з метою перебору всіх параметрів.

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
```

3.7. Відміна макросів

Для відміни багаторядкових макросів використовується директива **%unmacro**:

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

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 заставляти препроцесор повідомляти 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 буде розгорнутий в інструкцію умовного переходу `je`.

3.9. Зчеплення макросів для утворення таблиць кодів

Виклики макросів можуть об'єднуватися 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
```

макроси розширяться до:

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

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 передпроцесорного стеку контекстів має таке ж ім'я, як один із аргументів.

Для тестування довільних числових виразів використовуються 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

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
```

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'
```

7. Стек контекстів

Іноді потрібно розділити локальні позначки макросу із іншими макросами. Для цього Nasm підтримує стек контекстів, кожен з яких характеризується 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 визначити або отримати доступ до локальних позначок контексту нижче від верхівки стеку можна використати **`%$$label`**, або **`%$$$label`** і так далі.

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-байтову межу
align 16        ; вирівняти на 16-байтову межу
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
```

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)
```

Контрольні запитання.

1. Що таке препроцесор асемблера і для чого він призначений.
2. Директиви **`%define`**, **`%xdefine`** і їх вкладені розширення.
3. Директиви **`%+`**, **`%?`**, **`%??`**, **`%undef`**, **`%strcat`**, **`%strlen`**, **`%substr`**.
4. Директива **`%assign`**
5. Багаторядкові макровизначення.
6. Перевантаження макросів.
7. Скупі макроси.
8. Багаторядкові макроси із змінним числом і діапазоном параметрів.
9. Параметри макросів – жадібні, змінне число і діапазон, лічильник.
10. Лічильник параметрів і прокручування параметрів багаторядкового макросу.
11. Зчеплення макропараметрів макросів.
12. Директиви асемблювання з умовами.
13. Директиви препроцесора для роботи з початковим файлом.
14. Для чого призначений стек контекстів і директиви для роботи з ним.
15. Стандартні макроси і стандартні макропакети.

Практична частина

Завдання.

1. Написати макрос `print_n` для виведення на екран символу нового рядка (“\n”).

2. Написати макрос `print_str` для виведення на екран стрічки без символу нового рядка (“\n”).

3. Написати макрос `print_str` для виведення на екран стрічки з символом нового рядка (“\n”).

4. Написати макрос `print_int` для виведення на екран цілого числа.

5. Написати макрос `print_float` для виведення на екран числа з плаваючою крапкою.

7. Написати макрос `print_reg` для виведення на екран вмісту регістра загального призначення.

8. Написати макрос `print_regs` для виведення на екран вмісту всіх регістрів загального призначення.

9. Написати макрос `print_freg` для виведення на екран вмісту регістра з плаваючою крапкою.

10. Написати макрос `print_fregs` для виведення на екран вмісту всіх регістрів з плаваючою крапкою.

11. Написати макрос `print_preg` для виведення на екран вмісту MMX регістра.

12. Написати макрос `print_pregs` для виведення на екран вмісту всіх MMX регістрів.

13. Написати макрос `print_flags` для виведення на екран вмісту системного регістра прапорів `flags`.

14. Написати макрос `print_flag` для виведення на екран вмісту заданого біту системного регістра прапорів `flags`.

15. Написати макрос `print_sp` для виведення на екран вмісту верхівки стеку.

16. Написати макрос `print_stack` для виведення на екран вмісту стеку.

17. Написати макрос `print_mem` для виведення на екран вмісту комірки пам’яті.

18. Написати макрос `print_ari` для виведення на екран результату арифметичних операцій (додавання, віднімання, множення, ділення) для двох цілих чисел.

19. Написати макрос `print_arif` для виведення на екран результату арифметичних операцій (додавання, віднімання, множення, ділення) для двох чисел з плаваючою крапкою.

20. Написати макрос `print_arip` для виведення на екран результату арифметичних операцій (додавання, віднімання, множення) для двох MMX чисел.

21. Написати макрос `print_struct` для створення, ініціалізації і друку значень структури `mytype`:

```
struc mytype
   mt_var1: resb 1
   mt_var2: resw 2
   mt_var3: resd 4
   mt_var4: resq 8
endstruc
```

Примітка. Номер варіанта завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

1. Назва групи, П.І.Б. студента, завдання до роботи.

2. Короткий опис теоретичної частини.
3. Текст програми із поясненнями.
4. Роздруки екранів із результатами виконання у консолі або налагоджувачі.

Приклади для самостійної роботи

1. Макродирективa визначення довжини стрічки:

```
; 1.asm
%strlen L "mystring"
segment .bss
buf resb L
segment .text
global main
main:
mov rax,60
mov rdi,L
syscall
```

```
$ ./asm_ld.sh 1.asm
$ ./1
$ echo $?
8
```

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'`.

```
; 2.asm
%substr s "mystring" 3
segment .bss
buf resb 1
segment .text
global main
main:
mov rax,60
mov rdi,s
syscall
```

```
$ ./asm_ld.sh 2.asm
$ ./2
$ echo $?
115
$ man ascii
Код 115 відповідає символу "s"
```

3. Директива %rep. Оголошення і ініціалізація області пам'яті із 5 комірок:

```
; 3.asm
segment .data
mas:
%assign i 0
%rep 5
```

```
  db i
  %assign i i+1
%endrep

segment .text
global main
main:
%assign i 0
%rep 5
  mov [mas+i],byte i
  %assign i i+1
%endrep

mov rax,60
mov rdi,[mas+4] ; п’ятий елемент
syscall
```

```
$ ./asm_ld.sh 3.asm
$ ./3
$ echo $?
4
```

4. Директива %rep для генерації послідовності з 12 команд inc, які збільшують на 1 значення кожного елементу масиву.

```
segment .data
array:
%assign i 0
%rep 12
  dq i
  %assign i i+1
%endrep

segment .text
global main
main:

%assign i 0
%rep 12
   inc qword [array+i]
   %assign i i+8
%endrep

mov rax,60
mov rdi, [array+10]  ; одинадцятий елемент
syscall
```

```
$ ./asm_ld.sh 4.asm
$ ./4
$ echo $?
11
```

5. Макрос для очистки області пам’яті заданої позначкою і довжиною

```
; 5.asm
%macro clear_mem 2 ; два параметри – адреса і довжина
    push rcx
```

```
    push rsi
    push qword %2
    push dword %1
    pop rsi
    pop rcx
    %%lp: mov byte [rsi], 0
        inc rsi
    loop %%lp
    pop rsi
    pop rcx
%endmacro

segment .data
mas: dq  1,2,3,4,5,6,7,8,9,10
mas_len  equ ($-mas)/8

segment .text
global main
main:
nop

clear_mem mas, mas_len

mov rax,60
mov rdi,[mas+8]  ; 1-й елемент
syscall
```

```
$ ./asm_ld.sh 5.asm
$ ./6
$ echo $?
0
```

6. Макрос для виклику процедури з трьома параметрами

```
; 6.asm
%macro prog_call 3
    push %3
    push %2
    call %1
%endmacro

segment .data
a: dq 5
b: dq 4
segment .text
global main
main:

     prog_call sub1,qword [a],qword [b]

     add rsp,16   ; звільнити стек від параметрів
                  ; в ebx результат з підпрограми
     mov rax,60   ; передати результат у Linux
     mov rsi,rbx  ; результат з підпрограми
     syscall
```

```
sub1:
     push rbp
     mov rbp,rsp          ; вхід в підпрограму

     sub rsp,16           ; місце під локальні змінні
     mov qword [rbp-8],2  ; локальна змінна 2
     mov qword [rbp-16],8  ; локальна змінна 8
     mov rax,[rbp-8]      ; 2->eax
     mul qword [rbp-16]    ; eax = eax * локальну змінну 9

     mov rbx,[rbp+24]   ; [a]
     add rbx,[rbp+16]   ; [a]+[b]
     add rbx,rax        ; [a]+[b] + eax = 25 результат в ebx

     mov rsp,rbp       ; звільнити локальні змінні, вийти з підпрограми
     pop rbp
     ret
```

```
$ ./asm_ld.sh 6.asm
$ ./6
$ echo $?
 25
```

7. Макрос для виклику процедури із довільним числом параметрів (параметри мають розмір 8 байтів і у стек поміщаються у зворотному порядку).

```
; 7.asm
%macro PCALL 1-*
%rep %0 - 1 ; цикл по всіх параметрах крім першого
%rotate -1 ; останній параметр стає %1
push qword %1
%endrep
%rotate -1 ; адреса процедури стає %1
call %1
add rsp, (%0 - 1) * 8
%endmacro
```

8. Макрос для виведення стрічки.

```
; 8.asm
;------------------
%macro print_str 1
;------------------
;; %1 - message
;pushaq
pushf
jmp %%astr
%%str db %1,10     ; 10 – new line
%%len equ $-%%str
%%astr:
mov rax,1          ; 1 - write
mov rdi,1          ; 1 - STDOUT
mov rsi,%%str
mov rdx,%%len      ; довжина msg
syscall
popf
;popaq
```

```
%endmacro

section .data
;msg db ”Привіт світ”
section .bss
section .txt
  glonal main
main:
  PRINT_STR ”Привіт світ”
mov rax,60
mov rdi,0
syscall
```

$ ./8
Привіт світ

; 9 Багаторядковий макрос з двома параметрами для друку тексту і цілого числа

```
; 9.asm
extern printf
%define double_it(r) sal r, 1 ; single line macro
%macro prntf 2 ; ,multiline macro with 2 arguments
section .data
%%arg1 db %1,0 ; перший аргумент
%%fmtint db "%s %ld",10,0 ; формат для string
section .text ; друк аргументів
 mov rdi,%%fmtint
mov rsi,%%arg1
mov rdx,[%2] ; другий аргумент
mov rax,0 ; без плаваючої крапки
call printf
%endmacro
section .data
number dq 15
section .bss
section .text
global main
main:
push rbp
mov rbp,rsp
prntf "Число ", number
mov rax, [number]
double_it(rax)
mov [number],rax
prntf "Число помножене на 2 ", number
leave
ret
```

$ ./asm_ld.sh 9.asm
$ ./9
Число 15
Число помножене на 2 30

10. Деасемблювання двійкового файлу

```
$ objdump -M intel -d 9
```

```
6_1:     file format elf64-x86-64

Disassembly of section .init:

00000000004003c8 <_init>:
  4003c8:       48 83 ec 08             sub    rsp,0x8
  4003cc:       48 8b 05 25 0c 20 00    mov    rax,QWORD PTR [rip+0x200c25]
# 600ff8 <__gmon_start__>
  4003d3:       48 85 c0                test   rax,rax
  4003d6:       74 02                   je     4003da <_init+0x12>
  4003d8:       ff d0                   call   rax
  4003da:       48 83 c4 08             add    rsp,0x8
  4003de:       c3                      ret

Disassembly of section .plt:

00000000004003e0 <.plt>:
  4003e0:       ff 35 22 0c 20 00       push   QWORD PTR [rip+0x200c22]
# 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
  4003e6:       ff 25 24 0c 20 00       jmp    QWORD PTR [rip+0x200c24]
# 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
  4003ec:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]

00000000004003f0 <printf@plt>:
  4003f0:       ff 25 22 0c 20 00       jmp    QWORD PTR [rip+0x200c22]
# 601018 <printf@GLIBC_2.2.5>
  4003f6:       68 00 00 00 00          push   0x0
  4003fb:       e9 e0 ff ff ff          jmp    4003e0 <.plt>

Disassembly of section .text:

0000000000400400 <_start>:
  400400:       31 ed                   xor    ebp,ebp
  400402:       49 89 d1                mov    r9,rdx
  400405:       5e                      pop    rsi
  400406:       48 89 e2                mov    rdx,rsp
  400409:       48 83 e4 f0             and    rsp,0xfffffffffffffff0
  40040d:       50                      push   rax
  40040e:       54                      push   rsp
  40040f:       49 c7 c0 d0 05 40 00    mov    r8,0x4005d0
  400416:       48 c7 c1 60 05 40 00    mov    rcx,0x400560
  40041d:       48 c7 c7 f0 04 40 00    mov    rdi,0x4004f0
  400424:       ff 15 c6 0b 20 00       call   QWORD PTR [rip+0x200bc6]
# 600ff0 <__libc_start_main@GLIBC_2.2.5>
  40042a:       f4                      hlt
  40042b:       0f 1f 44 00 00          nop    DWORD PTR [rax+rax*1+0x0]

0000000000400430 <_dl_relocate_static_pie>:
  400430:       f3 c3                   repz ret
  400432:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
  400439:       00 00 00
  40043c:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]

0000000000400440 <deregister_tm_clones>:
  400440:       55                      push   rbp
```

```
       400441:       b8 80 10 60 00          mov    eax,0x601080
       400446:       48 3d 80 10 60 00       cmp    rax,0x601080
       40044c:       48 89 e5                mov    rbp,rsp
       40044f:       74 17                   je     400468
<deregister_tm_clones+0x28>
       400451:       b8 00 00 00 00          mov    eax,0x0
       400456:       48 85 c0                test   rax,rax
       400459:       74 0d                   je     400468
<deregister_tm_clones+0x28>
       40045b:       5d                      pop    rbp
       40045c:       bf 80 10 60 00          mov    edi,0x601080
       400461:       ff e0                   jmp    rax
       400463:       0f 1f 44 00 00          nop    DWORD PTR [rax+rax*1+0x0]
       400468:       5d                      pop    rbp
       400469:       c3                      ret
       40046a:       66 0f 1f 44 00 00       nop    WORD PTR [rax+rax*1+0x0]

     0000000000400470 <register_tm_clones>:
       400470:       be 80 10 60 00          mov    esi,0x601080
       400475:       55                      push   rbp
       400476:       48 81 ee 80 10 60 00    sub    rsi,0x601080
       40047d:       48 89 e5                mov    rbp,rsp
       400480:       48 c1 fe 03             sar    rsi,0x3
       400484:       48 89 f0                mov    rax,rsi
       400487:       48 c1 e8 3f             shr    rax,0x3f
       40048b:       48 01 c6                add    rsi,rax
       40048e:       48 d1 fe                sar    rsi,1
       400491:       74 15                   je     4004a8
<register_tm_clones+0x38>
       400493:       b8 00 00 00 00          mov    eax,0x0
       400498:       48 85 c0                test   rax,rax
       40049b:       74 0b                   je     4004a8
<register_tm_clones+0x38>
       40049d:       5d                      pop    rbp
       40049e:       bf 80 10 60 00          mov    edi,0x601080
       4004a3:       ff e0                   jmp    rax
       4004a5:       0f 1f 00                nop    DWORD PTR [rax]
       4004a8:       5d                      pop    rbp
       4004a9:       c3                      ret
       4004aa:       66 0f 1f 44 00 00       nop    WORD PTR [rax+rax*1+0x0]

     00000000004004b0 <__do_global_dtors_aux>:
       4004b0:       80 3d c5 0b 20 00 00    cmp    BYTE PTR [rip+0x200bc5],0x0
# 60107c <completed.7698>
       4004b7:       75 17                   jne    4004d0
<__do_global_dtors_aux+0x20>
       4004b9:       55                      push   rbp
       4004ba:       48 89 e5                mov    rbp,rsp
       4004bd:       e8 7e ff ff ff          call   400440 <deregister_tm_clones>
       4004c2:       c6 05 b3 0b 20 00 01    mov    BYTE PTR [rip+0x200bb3],0x1
# 60107c <completed.7698>
       4004c9:       5d                      pop    rbp
       4004ca:       c3                      ret
       4004cb:       0f 1f 44 00 00          nop    DWORD PTR [rax+rax*1+0x0]
       4004d0:       f3 c3                   repz ret
```

```
  4004d2:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]
  4004d6:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
  4004dd:       00 00 00

00000000004004e0 <frame_dummy>:
  4004e0:       55                      push   rbp
  4004e1:       48 89 e5                mov    rbp,rsp
  4004e4:       5d                      pop    rbp
  4004e5:       eb 89                   jmp    400470 <register_tm_clones>
  4004e7:       66 0f 1f 84 00 00 00    nop    WORD PTR [rax+rax*1+0x0]
  4004ee:       00 00

00000000004004f0 <main>:
  4004f0:       55                      push   rbp
  4004f1:       48 89 e5                mov    rbp,rsp
  4004f4:       48 bf 44 10 60 00 00    movabs rdi,0x601044
  4004fb:       00 00 00
  4004fe:       48 be 38 10 60 00 00    movabs rsi,0x601038
  400505:       00 00 00
  400508:       48 8b 14 25 30 10 60    mov    rdx,QWORD PTR ds:0x601030
  40050f:       00
  400510:       b8 00 00 00 00          mov    eax,0x0
  400515:       e8 d6 fe ff ff          call   4003f0 <printf@plt>
  40051a:       48 8b 04 25 30 10 60    mov    rax,QWORD PTR ds:0x601030
  400521:       00
  400522:       48 d1 e0                shl    rax,1
  400525:       48 89 04 25 30 10 60    mov    QWORD PTR ds:0x601030,rax
  40052c:       00
  40052d:       48 bf 72 10 60 00 00    movabs rdi,0x601072
  400534:       00 00 00
  400537:       48 be 4c 10 60 00 00    movabs rsi,0x60104c
  40053e:       00 00 00
  400541:       48 8b 14 25 30 10 60    mov    rdx,QWORD PTR ds:0x601030
  400548:       00
  400549:       b8 00 00 00 00          mov    eax,0x0
  40054e:       e8 9d fe ff ff          call   4003f0 <printf@plt>
  400553:       c9                      leave
  400554:       c3                      ret
  400555:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
  40055c:       00 00 00
  40055f:       90                      nop

0000000000400560 <__libc_csu_init>:
  400560:       41 57                   push   r15
  400562:       41 56                   push   r14
  400564:       49 89 d7                mov    r15,rdx
  400567:       41 55                   push   r13
  400569:       41 54                   push   r12
  40056b:       4c 8d 25 9e 08 20 00    lea    r12,[rip+0x20089e]        #
600e10 <__frame_dummy_init_array_entry>
  400572:       55                      push   rbp
  400573:       48 8d 2d 9e 08 20 00    lea    rbp,[rip+0x20089e]        #
600e18 <__init_array_end>
  40057a:       53                      push   rbx
  40057b:       41 89 fd                mov    r13d,edi
```

```
  40057e:       49 89 f6                mov    r14,rsi
  400581:       4c 29 e5                sub    rbp,r12
  400584:       48 83 ec 08             sub    rsp,0x8
  400588:       48 c1 fd 03             sar    rbp,0x3
  40058c:       e8 37 fe ff ff          call   4003c8 <_init>
  400591:       48 85 ed                test   rbp,rbp
  400594:       74 20                   je     4005b6 <__libc_csu_init+0x56>
  400596:       31 db                   xor    ebx,ebx
  400598:       0f 1f 84 00 00 00 00    nop    DWORD PTR [rax+rax*1+0x0]
  40059f:       00
  4005a0:       4c 89 fa                mov    rdx,r15
  4005a3:       4c 89 f6                mov    rsi,r14
  4005a6:       44 89 ef                mov    edi,r13d
  4005a9:       41 ff 14 dc             call   QWORD PTR [r12+rbx*8]
  4005ad:       48 83 c3 01             add    rbx,0x1
  4005b1:       48 39 dd                cmp    rbp,rbx
  4005b4:       75 ea                   jne    4005a0 <__libc_csu_init+0x40>
  4005b6:       48 83 c4 08             add    rsp,0x8
  4005ba:       5b                      pop    rbx
  4005bb:       5d                      pop    rbp
  4005bc:       41 5c                   pop    r12
  4005be:       41 5d                   pop    r13
  4005c0:       41 5e                   pop    r14
  4005c2:       41 5f                   pop    r15
  4005c4:       c3                      ret
  4005c5:       90                      nop
  4005c6:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
  4005cd:       00 00 00

00000000004005d0 <__libc_csu_fini>:
  4005d0:       f3 c3                   repz ret

Disassembly of section .fini:

00000000004005d4 <_fini>:
  4005d4:       48 83 ec 08             sub    rsp,0x8
  4005d8:       48 83 c4 08             add    rsp,0x8
  4005dc:       c3                      ret
```

Лабораторна робота 8

Виклики підпрограм

Мета роботи: вивчення механізмів виклику підпрограм в асемблері.

Теоретична частина

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 (найбільша адреса), 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 і відновлення 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.

2. Організація стеку в процесорі x86-64

Для адресації верхівки стеку використовується регістр загального призначення `rsp` (stack pointer) – вказівник стеку.

Команда `push operand` заносить вміст операнду (регістру або комірки пам’яті) в стек, при цьому вказівник стеку `rsp` зменшує своє значення на 8. Якщо операнд не регістровий то його розмір необхідно вказати явно.

Регістр `rsp` завжди вказує на *наймолодшу адресу стека, яка в поточний момент використовується.*

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 8. Команди `push, pop` *не підтримують* операнди розміром байт

Якщо потрібно тільки звернутися до значення на верхівці стеку (без видобування його зі стеку) можна використати команду `mov` і операнд `[esp]`:

```
mov rax,rsp     - читання адреси верхівки стеку
mov rax,[rsp]   - читання значення на верхівці стеку
mov rax,[rsp+8] - 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 `rsi` містить адресу деякої стрічки символів в пам'яті, причому відомо, що вона закінчується нульовим байтом (заповненим нулями). Необхідно переписати елементи стрічки у зворотному порядку використовуючи стек.

```
; 1.asm - стек
extern printf
section .data
   strng db "ABCDE",0
   strngLen equ $ - strng-1 ; довжина стрічки без 0
   fmt1 db "Початкова стрічка: %s",10,0
   fmt2 db "Реверсована стрічка: %s",10,0
section .bss
section .text
   global main
main:
   push rbp
   mov rbp,rsp
; Друк початкової стрічки
   mov rdi, fmt1
   mov rsi, strng
   mov rax, 0
   call printf
; заштовхування символу за символом у стек
   xor rax, rax
   mov rbx, strng ; адреса strng у rbx
   mov rcx, strngLen ; довжина в лічильник rcx
   mov r12, 0 ; використання r12 як вказівника
pushLoop:
       mov al, byte [rbx+r12] ; переслати char в rax
       push rax ;заштовхнути rax у стек
       inc r12 ; збільшити char вказівник на 1
       loop pushLoop ; продовження циклу
;зчитування символу за символом із стеку
;це буде реверсувати початкову стрічку
   mov rbx, strng ; адресу strng в rbx
   mov rcx, strngLen ; довжину стрічки в лічильник rcx
   mov r12, 0 ; r12 як лічильник
popLoop:
       pop rax ; зчитування символу із стеку
       mov byte [rbx+r12], al ;пересилання символу у strng
       inc r12 ; збільшення вказівника символів на 1
       loop popLoop ; продовження циклу
       mov byte [rbx+r12], al ;пересилання символу у strng
; друк реверсованої стрічки
   mov rdi, fmt2
   mov rsi, strng
   mov rax, 0
   call printf
   mov rsp,rbp
   pop rbp
ret

$.~/asm_ld.sh 1.asm
$./1
Початкова стрічка: ABCDE
```

```
Реверсована стрічка: EDCBA
```

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. Основна програма, перед викликом підпрограми, записує у стек параметри виклику, а потім командою `call` записує адресу повернення і викликає підпрограму. Адреса повернення – це адреса наступної команди після `call`. Підпрограма, перед початком свого виконання, резервує пам'ять для зберігання своїх локальних змінних, шляхом зменшення адресу верхівки стеку на потрібну величину. Область стекової пам'яті, яка містить зв'язані з одним викликом `call` значення параметрів, адреси повернення і локальні змінні, називається *стековим фреймом*.

3.1. Виклик підпрограми і повернення з неї

Асемблер підтримує виклики наступних підпрограм: функцій і процедур. *Функція* виконує команди і повертає значення. *Процедура* виконує команди і не повертає значення. Так в асемблері можна викликати функції мови Сі, наприклад printf, scanf і т. п.

В асемблері є дві команди для організації роботи підпрограм:

`call позначка` – викликає підпрограму, код якої знаходиться за адресою 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 8 байтів з верхівки стеку і записує їх в регістр `RIP`, в результаті чого керування передається за адресою повернення, яка знаходилася в пам'яті на верхівці стеку.

Існують наступні методи передачі параметрів в підпрограму:

- через регістри;
- через загальну пам'ять;
- через стек.

Приклад виклику процедури. Нехай в процедурі потрібно заповнювати 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`.

```
; 2.asm – виклик процедури
```

```
; Процедура fill_memory (edi=address, ecx=length, al=value)
fill_memory:
    jrcxz lq  ; перевірка регістра rcx на 0
lp: mov [rdi],al
    inc rdi
    loop lp
lq: ret

segment .data
len equ 8
section .text
global main
main:
    mov rdi,my_array
    mov rcx,len
    mov al,'#'
    call fill_memory

    mov rax,1  ; print array
    mov rdi,1
    mov rsi,my_array
    mov rdx,len
    syscall

    mov rax,60  ; exit
    mov rdi,0
    syscall
section .bss
my_array resb len
```

```
>./2
########
```

Приклад виклику функції, яка обчислює площу круга. У функції використовується команда `leave`, яка виконує те саме що і команди епілогу:

```
mov rsp,rbp
pop rbp

; 4.asm - виклик функції
extern printf
section .data
  radius dq 10.0
  pi dq 3.14
  fmt db "Площа круга %.2f",10,0
section .text
global main
;-----------------------------------------------
main:
push rbp
mov rbp, rsp
  call area ; call the function
  mov rdi,fmt ; print format
  movsd xmm1, [radius] ; move float to xmm1
  mov rax,1 ; area in xmm0
  call printf
```

```
leave
ret
;-------------------------------------------
area:
push rbp
mov rbp, rsp
  movsd xmm0, [radius] ; move float to xmm0
  mulsd xmm0, [radius] ; multiply xmm0 by float
  mulsd xmm0, [pi] ; multiply xmm0 by float
leave
ret

$ ./4
Площа круга 314.00
```

3.2. Виклики функцій із функцій

У асемблері функції можуть викликати інші функції, *але не можуть бути вкладеними*. Крім того функції можуть мати власні розділи `.data`, `.bss`, `.text`. У функціях можуть використовуватися локальні змінні. У цьому випадку перед ними ставиться крапка, що зменшує ризик конфлікту імен.

При виклику функцій і процедур їм можуть передаватися/[прийматися] через регістри аргументи.

Аргументи без плаваючої крапки, такі як цілі числа та адреси передаються через наступні регістри `rdi, rsi, rdx, rcx, r8, r9` (AMD64 ABI) або `rdi, rsi, r8, r9` (Windows x64 ABI). Додаткові аргументи передаються через стек у зворотному порядку.

Аргументи з плаваючою крапкою передаються послідовно через регістри `xmm0, xmm1, . . . , xmm7`. Додаткові аргументи передаються через стек без використання команди `push`.

4. Організація стекового фрейму

У прикладі виклику підпрограми в 2.asm фактично не використовувався механізм стекових фреймів, у стеку зберігалася лише адреса повернення, а параметри підпрограмі 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 параметрів та локальних змінних, на які не вистачить регістрів. Крім того, при передачі параметрів через загальну область пам'яті або регістри не можливо використати рекурсію.

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 викликаюча 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 8-байтові параметри, а потім викликати підпрограму, то адреса повернення буде в `[rsp]`, а адреси параметрів будуть `[rsp+8]`, `[rsp+16]`, `[rsp+24]`. Якщо в підпрограмі використовуються дві локальні змінні, то їх адреси будуть `[rsp-8]`, `[rsp-16]`.

Однак, регістр вказівник верхівки стеку може використовуватися у підпрограмі (для виклику інших підпрограм), тому його значення зберігають в іншому регістрі, переважно в `rbp`

(регістрі вказівника бази). Тоді в регістрі `rbp` буде старе значення верхівки стеку, а в регістрі `rsp` нове значення верхівки стеку.

В програмі можуть викликатися декілька підпрограм, які також використовують регістр `rbp`. Для збереження значення `rbp` 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 rbp і відновлює його перед поверненням управління.*

Для зберігання `rbp` також використовується стек, причому збереження здійснюється командою `push rbp` зразу після отримання управління підпрограмою. Таким чином, старе значення `rbp` розміщується в стеку після адреси повернення з підпрограми. Саме це старе значення `rbp` використовується як реперна точка для адресації стекового фрейму:

```
[rbp+32]
[rbp+24]    параметри
[rbp+16]
[rbp+8]    адреса повернення
[rbp]      збережене старе значення rbp    <= RBP
[rbp-8]
[rbp-16]    локальні змінні
[rbp-24]                                   <= RSP
128 байтів "червоної зони"
```

128-байтова область за межами, на які вказує `rsp`, вважається зарезервованою та не повинна бути змінена обробниками сигналів або переривань.

Кожна підпрограма повинна виконати наступні команди перед початком роботи:

```
push rbp     ; зберігання старого rbp
mov rbp,rsp  ; встановлення нового rbp
sub rsp,N ; де N число, яке резервує область пам'яті  для локальних змінних
```

і перед завершенням роботи:

```
add rsp, N    ; звільнення області пам'яті локальних змінних
mov rsp, rbp  ; відновлення старого rsp
pop rbp       ; відновлення старого rbp
ret
```

Процесор підтримує спеціальні команди для обслуговування стеку. На початку підпрограми може використовуватися команда `enter N`, а в кінці програми – 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.

5. Виклик підпрограм на мові Сі

При трансляції викликів функції мови Сі параметри поміщаються у стек у порядку справа наліво і розміщуються у стековому фреймі у порядку знизу вверх. Таким чином, перший параметр завжди буде доступним за адресою `[rbp+16]`, незалежно від загальної кількості параметрів

```
aN
...
a2
a1                          <= RBP-16
адреса повернення           <= RBP-8
rbp                         <= RBP
```

Тому у підпрограму можна передавати змінне число параметрів. Так як підпрограма не знає скільки їй передали параметрів, то очищення стеку виконує викликаюча програма простим збільшенням значення `rsp` на число, яке дорівнює сукупній довжині фактичних параметрів. Наприклад, якщо підпрограма `proc1` приймає три 8-байтові параметри, то її виклик виглядатиме так:

```
push qword a3 ; занесення в стек параметрів
push qword a2
push qword a1
call proc1    ; виклик підпрограми
add rsp,24    ; вилучення параметрів зі стеку
```

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 при запуску задачі на виконання, після чого керування передається програмі. Таким чином, у момент, коли починає виконуватися програма з позначки `main`, на верхівці стеку [`rsp`] розміщується число елементів командного рядка, в [`rsp+8`] – адреса де розміщується ім’я програма, а далі адреси наступних аргументів підпрограми.

Після цього стек матиме наступний вигляд:

```
Число елементів командного рядка  <-  [rbp+(N+1)*8 + 16]
Ім'я програми                     <-  [rbp+(N+1)*8 + 8]
Аргумент N                        <-  [rbp+(N+1)*8]
...                          ...
Аргумент 2                        <-  [rbp+3*8]
Аргумент 1                        <-  [rbp+2*8]
Адреса повернення (rsp старе)     <-  [rbp+8]
rbp (старе)                       <-  [rbp]      <= rbp
Локальна змінна1                  <-  [rbp-8]
Локальна змінна2                  <-  [rbp-16]   <= rsp
```

```
; 3.asm - виведення аргументів командного рядка
extern printf
section .data
  msg db "The command and arguments: ",10,0
  fmt db "%s",10,0
section .text
  global main
main:
push rbp
mov rbp,rsp
  mov r12, rdi ; число аргументів
  mov r13, rsi ; адреса масиву аргументів
;print the title
  mov rdi, msg
  call printf
  mov r14, 0
;друк команд і аргументів
.ploop: ; цикл з друком по масиву
  mov rdi, fmt
  mov rsi, qword [r13+r14*8]
```

```
  call printf
  inc r14
  cmp r14, r12 ; чи досягнуто число оброблених аргументів?
  jl .ploop
leave
ret

$ ./3 arg1 arg2 -m 5
./3
arg1
arg2
-m
5
```

Контрольні запитання.

1. Поняття стеку і його призначення.
2. Організація стеку в процесорі x86.
3. Підпрограми, функції, процедури, аргументи, локальні змінні.
4. Виклик підпрограми і повернення з неї. Команди `call, ret`.
5. Організація стекового фрейму. Регістри `rsp, rbp`.
6. Послідовність дій при роботі зі стековим фреймом.
7. Виклик підпрограм на мові Сі. Змінне число аргументів.
8. Передача командного рядка програмі.

Практична частина

Завдання.

1. Написати програму на асемблері NASM, яка викликає функцію-1 з цілочисловими аргументами *a, b, c*. Функція-1 викликає функцію-2 і передає їй параметри *a, b, c*. Значення *a+b+c* повернути з функції-2 в функцію-1, а з неї в основну програму і значення вивести у консоль.

2. Написати програму на асемблері NASM, у яку передається змінне число аргументів командного рядка. Основна програма викликає функцію і передає їй параметри командного рядка. У функції вивести на консоль отримані аргументи.

3. Написати програму на Сі, яка викликає функцію асемблера NASM і передає їй п’ять цілих чисел через регістри. Функція на асемблері визначає максимальне з отриманих чисел і повертає значення в основну програму. Отримане значення в основній програмі вивести на екран.

4. Написати програму на Сі, яка викликає функцію написану асемблері NASM і передає їй як аргументи три стрічки символів. Функція на асемблері зчіплює отримані стрічки в одну і повертає їх основну програму. Основна програма виводить отримане значення у консоль.

5. Написати програму на Сі, яка викликає функцію написану асемблері NASM і передає їй стрічку довжиною 8 символів і цілочисельне значення 16 через стек. Функція на асемблері виводить отримані аргументи у консоль.

6. Написати програму на асемблері NASM, яка передає функції ціле число N. Рекурсивна функція обчислює добуток чисел від 1 до N і результат повертає в основну програму. Основна програма виводить результат у консоль.

7. Написати програму на NASM, яка передає підпрограмі значення *a* в радіанах. Підпрограма ітераційно через ряд Тейлора обчислює *sin a* отриманого числа і результат повертає в основну програму. Основна програма виводить результат у консоль.

8. Написати програму на NASM, яка передає підпрограмі ціле число. Підпрограма ітераційно через ряд Маклорена обчислює e^x заданого числа і результат повертає в основну програму. Основна програма виводить результат у консоль.

9. Написати програму на асемблері NASM, якій у командному рядку передається стрічка символів. Програма створює і ініціалізує стрічку символів у буфері, сортує елементи стрічки у пам'яті. Початкову і відсортовану стрічки виводить у консоль.

10. Написати програму на асемблері NASM, яка читає стрічку символів з файлу, записує у стек. Функція програми читає стрічку із стеку і виводить у консоль у зворотному порядку.

11. Написати програму на асемблері NASM, якій в командному рядку задається стрічка символів. Програма передає стрічку функції через стек. Функція записує стрічку у файл.

12. Написати програму на асемблері NASM, яка створює і ініціалізує масив цілих чисел розміром 3×4 у пам'яті, викликає функцію, яка виводить суму рядків масиву у консоль.

13. Написати програму на асемблері NASM, яка створює і ініціалізує масив цілих чисел 5×6 у пам'яті , викликає функцію, яка виводить найбільші значення рядків у консоль.

14. Написати програму на асемблері NASM, яка створює і ініціалізує масив цілих чисел 6×4 у пам'яті , викликає підпрограму, яка виводить відсортовані значення рядків у консоль.

15. Написати програму на асемблері NASM, яка створює і ініціалізує стрічку буквами і цифрами, передає стрічку у функцію через стек, яка виводить на екран окремо букви і окремо цифри.

Примітка. Номер варіанта завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

1. Назва групи, П.І.Б. студента, завдання до роботи.
2. Короткий опис теоретичної частини.
3. Текст програми із поясненнями.
4. Роздруки екранів стану пам'яті і регістрів з результатами виконання програми у консолі або налагоджувачі `KDbg/dbg`.

Приклади для самостійної роботи

1. 5.asm — виклики функції із функції

```
extern printf
section .data
  radius dq 10.0
section .bss
section .text
;-----------------------------------------------
area:
section .data
  .pi dq 3.141592654 ; локальна в area
section .text
push rbp
mov rbp, rsp
  movsd xmm0, [radius]
  mulsd xmm0, [radius]
```

```
  mulsd xmm0, [.pi]
leave
ret
;-----------------------------------------------
circum:
section .data
.pi dq 3.14 ; локальна в circum
section .text
push rbp
mov rbp, rsp
  movsd xmm0, [radius]
  addsd xmm0, [radius]
  mulsd xmm0, [.pi]
leave
ret
;-----------------------------------------------
circle:
section .data
  .fmt_area db "Площа круга %f",10,0
  .fmt_circum db "Довжина кола %f",10,0
section .text
push rbp
mov rbp, rsp
  call area
  mov rdi,.fmt_area
  mov rax,1 ; площа в xmm0
  call printf
  call circum
  mov rdi,.fmt_circum
  mov rax,1 ; довжина кола в xmm0
  call printf
leave
ret
;-----------------------------------------------
global main
main:
push rbp
mov rbp, rsp
  call circle
leave
ret
```

```
$ ./5
```
Площа круга 314.159265
Довжина кола 62.800000

2. 6.asm - програма друку цілих чисел з пам'яті і регістра

```
; Еквівалентний C код
; #include <stdio.h>
; int main()
; {
;   int a=5;
;   printf("a=%d, eax=%d\n", a, a+2);
;   return 0;
; }
```

```
; 6.asm
; Оголошення зовнішніх функцій
extern printf ; С функція, яка викликається
section .data
a: dq 5         ; int a=5;
fmt: db "a=%ld, rax=%ld",10, 0 ; printf формат, "\n",'0'
section .text
  global main  ; Стандартна точку входу в gcc
main:          ; позначка для точки входу
push rbp       ; налаштування фрейма стеку
mov rbp,rsp
  mov rdi,fmt
  mov rsi,[a]
  mov rdx,rsi
  add rdx, 2     ; rsi=rsi+2
  mov rax,0      ; значення регістра rax у стек
  call printf
mov rsp,rbp      ; звільнення стекового фрейму
pop rbp          ; відновлення старого ebp
mov rax,0        ; код повернення для return
ret              ; повернення в ОС

$ ./5
a=5, rax=7
```

3. 7.asm - приклад передачі параметрів у функцію і повернення результату

```
;nasm -f elf64 7.asm -o 7.o
;ld -m elf_x86_64 7.map 7.o -o 7
segment .data
a: dq 4
b: dq 5
segment .text
global main
main:
    push dword [a]
    push dword [b]
    mov rdi,15
    mov rsi,20
    call sub1
    add esp,16 ; звільнити стек від параметрів
               ; в ebx результат з підпрограми
    mov rax,60  ; exit
    ;mov rdi,0
    syscall
;------------
sub1:
push rbp
mov rbp,rsp  ; вхід в підпрограму
sub esp,8    ; місце під локальні змінні

     mov rbx,[rbp+24]      ; rax <= 4
     add rbx,[rbp+16]      ; 4+5=9
            ;[rbp+8]       ; адреса повернення
     mov qword [rbp-8],15  ; локальна змінна 15
     mov qword [rbp-16],20 ; локальна змінна 20
```

```
     mov rax,qword [rbp-8] ; rax<=15
     add rax,qword [rbp-16]; 15+20=35
     mov rdi,rax
     add rdi,rbx  ; 9+35=44

add rsp,16          ; звільнити локальні змінні
mov rsp,rbp        ; вихід з підпрограми
pop rbp
ret

$ ./7
echo $?
44
```

4. Обчислення максимального з трьох чисел викликом asm програми з Сі-програми

```
/* Асемблювання і компонування програм:
* nasm -f elf64 8.asm -o 8.o
* gcc 8.c 8.o -o 8
*/
// 8.c
#include <stdio.h>
int maxof3_64(int, int, int);
int main() {
    printf("%d\n", maxof3_64(1,4,6));
    printf("%d\n", maxof3_64(2,7,1));
    printf("%d\n", maxof3_64(4,3,8));
    printf("%d\n", maxof3_64(9,4,3));
    printf("%d\n", maxof3_64(2,10,5));
    printf("%d\n", maxof3_64(1,11,6));
    return 0;
}

;-------------------------------------------------------------
; 64-бітова функція повертає максимум з трьох int аргументів.
; int maxof3_64(int x, int y, int z)
; Параметри передаються через регістри rdi,rsi,rdx
;-------------------------------------------------------------
global maxof3_64
section .text
maxof3_64:
  mov rax,rdi
  mov rbx,rsi
  mov rcx,rdx
  cmp rax,rbx
  ja m1
  mov rax,rsi
  jmp m2
m1: rdi>rsi
   mov rax,rdi
m2:
  cmp rax,rdx
  ja m3
  mov rax,rdx ; rax>rdx
m3:
  mov rdi,rax
```

```
ret

$ ./8
6
7
8
9
10
11
```

5. 9.asm - Виклик рекурсивної функції

```
extern printf
section .data
fmtint: db  "%d",10,0
section .text
;
factorial:
push rbp
mov rbp,rsp
  ; видобути аргумент
  mov rax,[rbp+16]
  cmp rax,0
  jne not_zero
  mov rax,1 ; факторіал 0 дорівнює 1
  jmp return
not_zero:
  ; rax = factorial(rax - 1)
  dec rax
  push rax
  call factorial
  add rsp,8
  ; видобути в rbx аргумент і обчислити rax = rax * rbx
  mov rbx, [rbp+16]
  mul rbx
return:
mov rsp,rbp
pop rbp
ret
; основна програма
global main
main:
push rbp
mov rbp,rsp
  push 5
  call factorial
  mov rsi,rax
  mov rax,0
  mov rdi,fmtint
  ;mov rsi, 0
  call printf

; епілог
;mov rax,60
;mov rdi,0
;syscall
```

```
mov rsp,rbp
pop rbp
ret ; альтернативний епілог

$ ./9
120
```

Лабораторна робота 9

Системні виклики

Мета роботи: вивчення системних викликів і їх аргументів в ОС Linux

Теоретична частина

1. Системні виклики

Програма асемблера взаємодіє з ОС, використовуючи так звані системні виклики. Системний виклик – це звернення програми користувача до ядра ОС для виконання деяких операцій. Класичний спосіб реалізації системних викликів – це використання переривання. 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 аварійному виконанні програми (ділення на нуль, порушення захисту пам’яті, виконання не існуючої команди, спроба читання слова за непарною адресою).

Програмні переривання ініціює користувач командою `syscall`. Програмне переривання також змінює режим виконання з користувацького на привілейований.

В ОС Linux при програмному перериванні відбувається звернення до ядра, але при цьому необхідно задати у певних регістрах номер системного виклику і аргументи. Номер системного виклику задається у регістрі `rax`. Якщо системний виклик має аргументи, то вони задаються в регістрах `rdi, rsi, rdx, r10, r8, r9`. Системні виклики звичайно не використовують стек для передачі своїх аргументів. Аргументи системних викликів є 8-байтовими цілими або адресними значеннями. Результат виконання виклику повертається через регістр `rax`.

Таким чином, через 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.

Кожний системний виклик має ім’я і номер. Всі вони перераховані в файлі `/usr/include/asm/unistd_64.h (64-розрядна ОС)`.

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 переміщуваність 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.

1.1. Системні виклики Linux x64

Команда : **`syscall`**

Номер системного виклику: **`rax`**. Повернення значень: **`rax`**.

Передача параметрів у `syscall`: **`rdi, rsi, rdx, r10, r8, r9`**.

Виведення переліку системних викликів: `$ cat /usr/include/asm/unistd_64.h`

За допомогою директиви `equ` номери системних викликів можна присвоїти іменованим позначкам:

```
#Номери системних викликів
SYS_READ  equ 0
SYS_WRITE equ 1
SYS_OPEN  equ 2
SYS_CLOSE equ 3
SYS_BRK   equ 12
SYS_EXIT  equ 60

#Стандартні номери (дескриптори) потоків введення/виведення
STDIN  equ  0
STDOUT equ  1
STDERR equ  2
```

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 1 і приймає три параметри: номер (дескриптор) потоку виведення, адресу пам'яті даних, кількість даних в байтах.

Інший важливий системний виклик – `exit`, який використовується для завершення програми. Він має номер 60 і один аргумент – код завершення програми.

Тоді програма, яка друкує повідомлення на екран і завершає роботу з поверненням коду у середовище Linux буде наступною:

```
 ; a1.asm
SYS_WRITE equ 1
STDOUT    equ 1
SYS_EXIT  equ 60
section .data
msg db "Привіт світ", 10,0  ;  110 -> '\n'
msg_len   equ $-msg
section .text
global main
main:
  mov rax,SYS_WRITE ; системний виклик write
  mov rdi,STDOUT    ; виведення у консоль
  mov rsi,msg       ; адреса стрічки для виведення
  mov rdx,msg_len   ; довжина стрічки
  syscall           ; звернення до ядра (системний виклик)

  mov rax,SYS_EXIT  ; системний виклик (exit, вихід з програми)
  mov rdi,0         ; код повернення в ОС
  syscall           ; системний виклик
```

```
$ ./a1
Привіт світ
```

Для введення даних (як з клавіатури, так із файлів) використовується виклик `read`, який має номер 0. Він має три параметри: номер (дескриптора) потоку введення, адресу пам'яті, де будуть розміщені вхідні дані, розмір даних в байтах. Після виклику `read` *потрібно проаналізувати* результат його роботи. Якщо читання пройшло успішно – виклик поверне кількість прочитаних байтів. 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_READ  equ 0
SYS_WRITE equ 1
SYS_EXIT  equ 60
STDIN     equ 0
STDOUT    equ 1
section .bss
msg resb 64
msg_len   equ $-msg
global main
section .text
global main
main:
  mov rax,SYS_READ   ; номер системного виклику
  mov rdi,STDIN      ; читання з консолі
  mov rsi,msg        ; адреса буфера, коди поміщаються дані
  mov rdx, msg_len   ; довжина даних
  syscall

  mov rax, SYS_WRITE ; системний виклик write
  mov rdi,STDOUT     ; виведення у консоль
  mov rsi,msg        ; адреса буфера, коди поміщаються дані
  mov rdx, msg_len   ; довжина даних
  syscall

  mov rax,SYS_EXIT  ; системний виклик (exit, вихід з програми)
  mov rdi,0         ; код повернення в ОС
  syscall           ; системний виклик
> ./a2
Привіт світ
Привіт світ
```

При створенні, відкритті, закритті і роботі з файлами потрібні додаткові потоки введення/виведення крім стандартних потоків. Додаткові потоки введення/виведення створюються системним викликом `open` з номером 2. Виклик має три параметри: *адреса рядка тексту*, який містить ім'я файлу (ім'я має закінчуватися нульовим байтом, який 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`).

Інформацію про режими використання і права доступу до файлів можна отримати із файлу:

```
$ /usr/include/asm-generic/fcntl.h
```

Деякі прапори режиму використання файлу:

```
O_RDONLY        Тільки читання      000h/0
O_WRONLY        Тільки запис        001h/1
O_RDWR          Читання і запис     002h/2
O_CREAT         Дозволити           040h/100o
                створення файлу
O_EXCL          Вимагати            080h/200o
                створення файлу
O_TRUNC         Якщо файл існує,    200h/1000o
                знищити його
                вміст
O_APPEND        Якщо файл існує     400h/2000o
                добавити у кінець
```

Наприклад комбінації прапорів `O_WRONLY|O_CREAT|O_TRUNC` мови Сі у Linux задається значенням `241h`.

Результат виклику `open` повертається в регістрі `rax`. Якщо виклик закінчився успішно, то `rax` 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 `rax` містить негативне значення.

Коли робота з файлом закінчена його потрібно закрити системним викликом `close`, який має номер 3. Виклик має один параметр, який задає дескриптор файлу, який потрібно закрити.

Приклад:

```
SYS_OPEN  equ 2
SYS_CLOSE equ 3
O_FLAGS equ 1101o  ; прапор запису
S_MODE  equ 666o ; права доступу
SYS_WRITE equ 1
SYS_EXIT  equ 60
FD_STDOUT equ 1
segment .bss          ; сегмент неініціалізованих даних
   fd  resb  1
segment .data         ; сегмент ініціалізованих даних
   file db "111",0
   msg  db "Привіт світ",10,0
   len equ $-msg
segment .text
global main           ; експорт точки входу в програму
main:
   nop
   mov rax,SYS_OPEN
   mov rdi,file
   mov rsi,O_FLAGS
   mov rdx,S_MODE
   syscall           ; відкриття файлу із заданим прапором і правами
```

```
   mov [fd],rax      ; запис файлового дескриптора
   jl label          ; аналіз результату

   ; write to file
   mov rax,SYS_WRITE
   mov rdi,[fd]
   mov rsi,msg
   syscall            ; запис у файл повідомлення

label:
   mov rax,SYS_EXIT
   mov rdi,[fd]
   syscall       ; завершення програми з поверненням дескриптора файлу
```

Для роботи із системними викликами, які мають різне число аргументів можна використати макровизначення `my_syscall`, наприклад, записавши його у файл `my_macro`:

```
%macro my_syscall 1-*
%rep %0
%rotate -1
  push qword %1
%endrep
  pop rax
%if %0 > 1
  pop rdi
  %if %0 > 2
    pop rsi
    %if %0 > 3
      pop rdx
      %if %0 > 4
       pop r10
       %if %0 > 5
         pop r8
         %if %0 > 6
        pop r9
        %if %0 > 7
              %error " Кількість аргументів syscall > 7 "
           %endif
     %endif
      %endif
      %endif
    %endif
  %endif
%endif
  syscall
%endmacro
```

Використання макросу `my_syscall` значно скорочує текст програми виведення стрічки з пам’яті на екран:

```
%include ”my_macro”
section .data
  msg db "Hello world",10,0
  msg_len equ $-msg
```

```
section .text
global main
main:
  m_syscall 1, 0, msg, msg_len
  m_syscall 60, 0
```

Контрольні запитання.

1. Які є типи переривань. Переривання для звернення до ОС Linux.
2. Передача аргументів 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.
6. Права доступу і режими використання файлу.
7. Структура програми асемблера для запису даних у файл.
8. Структура програми асемблера для читання даних з файлу.
9. Як працює макровизначення для роботи із системними викликами із різним числом параметрів?

Практична частина

Завдання.

1. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик для виведення на екран двох символьних стрічок

```
Привіт всім студентам,
які вивчають Intel x86_64 асемблер.
```

2. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик для зчитування з консолі стрічки символів ”Привіт світ” і виводить її на екран за допомогою `printf`.
3. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик для зчитування з консолі стрічки символів ”Привіт світ” і виводить її на екран у зворотному порядку за допомогою `printf`.
4. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик для запису стрічки символів ”Привіт світ” у файл.
5. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик для запису стрічки символі ”Привіт світ” у зворотному порядку у файл.
6. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик для читання вмісту текстового файлу і виводить його на екран за допомогою `printf`.
7. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик для читання вмісту текстового файлу і виводить його довжину на екран за допомогою `printf`.
8. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик `mkdir` для створення каталогу.
9. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик `rename` для перейменування файлу.
10. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик `chdir` для зміни каталогу.

11. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик `rmdir` для вилучення каталогу.

12. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик `chmod` для зміни атрибутів файлу.

зчитує команду Linux з консольного рядка і її виконує.

13. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик `fork` для створення з батьківського дочірнього процесу і в ньому виводить на екран повідомлення ”Привіт світ” за допомогою `printf`.

14. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик `time` і водить поточний час за допомогою `printf`.

15. Написати програму на NASM з використанням підпрограми, яка використовує системний виклик `execve` для запуску на виконання команди `dir`.

Примітка. Номер варіанта завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

1. Назва групи, П.І.Б. студента, завдання до роботи.
2. Короткий опис теоретичної частини.
3. Текст програми із поясненнями.
4. Роздруки екранів стану пам’яті і регістрів з результатами виконання програми у консолі або налагоджувачі `Kdbg/dbg` (при необхідності).

Приклади для самостійної роботи

1. 1.asm – Введення даних з консолі

```
; console1.asm
section .data
msg1 db "Привіт, світ!",10,0
msg1len equ $-msg1
msg2 db "Ваша черга: ",0
msg2len equ $-msg2
msg3 db "Ваша відповідь: ",0
msg3len equ $-msg3
inputlen equ 10 ;довжина вхідного буфера
section .bss
input resb inputlen+1 ;забезпечення місця для завершального 0
section .text
global main
main:
push rbp
mov rbp,rsp
mov rsi, msg1 ; друк першої string
mov rdx, msg1len
call prints
mov rsi, msg2 ; друк другої, без NL
mov rdx, msg2len
call prints
mov rsi, input ; адреса вхідного буфера
mov rdx, inputlen ; довжина вхідного буфера
call reads ; очікування входу
mov rsi, msg3 ; друк третьої string
```

```
    mov rdx, msg3len
    call prints
    mov rsi, input ; друк вхідного буфера
    mov rdx, inputlen ; довжина вхідного буфера
    call prints
    leave
    ret
    ; ----------------------------------------
    prints:
    push rbp
    mov rbp, rsp
    ; rsi містить адресу string
    ; rdx містить довжину string
    rax, 1 ; 1 = write
    mov rdi, 1 ; 1 = stdout
    syscall
    leave
    ret
    ;------------------------------------------------
    reads:
    push rbp
    mov rbp, rsp
    ; rsi містить адресу вхідного буфера
    ; rdi містить довжину вхідного буфера
    mov rax, 0 ; 0 = read
    mov rdi, 1 ; 1 = stdin
    mov rdx, inputlen ; довжина вхідного буфера
    syscall
    leave
    ret
```

```
$ ~/asm_ld.sh 1.asm
Асемблювання 1.asm успішне - 0
Компонування 1.asm успішне - 0
$ ./1
Привіт, світ!
Ваша черга: Вітаю
Ваша відповідь: Вітаю
```

2. 2.asm – Введення даних з консолі

У першому прикладі під вхідний буфер зарезервовано 10 байт. Якщо текст не помішається у буфер то виникає помилка:

```
$ ./1
Привіт, світ!
Ваша черга: як ви себе почуваєте
Ваша відповідь: як ви abc@xxx:~/Nasm/Lab8$ себе почуваєте
себе: command not found
```

Цей спосіб переповнення буфера використовують хакери для проникнення в операційну систему. Для усунення такої можливості під буфер потрібно виділяти стільки байтів, скільки є символів у вхідному тексті.

Функцію `prints` змінено, щоб вона спочатку рахувала кількість символів для дисплею; тобто вона рахує, поки не знайде байт 0. Коли довжина визначена, рядок виводиться системним викликом.

Функція `reads` чекає на один вхідний символ і перевіряє, чи це символ нового рядка. Якщо це символ нового рядка, то читання символів з клавіатури припиняється. Регістр `r14` містить кількість введених символів. Функція перевіряє чи кількість символів більша ніж *inputlen*; якщо ні, символ додається до вхідного буфера. Якщо *inputlen* перевищено, символ ігнорується, але зчитування з клавіатури продовжується. Була задана вимога, щоб ASCII код символу був від 97 до 122. Це буде гарантувати, що приймаються лише малі літери. Зауважимо, що було збережено та відновлено регістри викликаючої програми. Регістр `r12` використовувався в обох функціях, `prints` і `read`. У цьому випадку не збережено регістри викликаючої програми, що не створило проблему, але можна уявити, що якщо одна функція викликає іншу, а та викликає ще іншу, то тоді можуть виникати проблеми.

```
;2.asm
section .data
msg1 db "Привіт світ!",10,0
msg2 db "Ваша відповідь (тільки малі букви a-z): ",0
msg3 db "You answered: ",0
inputlen equ 10 ;довжина вхідного буфера
NL db 0xa
section .bss
input resb inputlen+1 ;забезпечення місця для завершального 0
section .text
global main
main:
push rbp
mov rbp,rsp
mov rdi, msg1 ; друк першого string
call prints
mov rdi, msg2 ; друк другого string, без NL
call prints
mov rdi, input ; адреса вхідного буфера
mov rsi, inputlen ; довжина вхідного буфера
call reads ; очікування для input
mov rdi, msg3 ; друк третього string і додавання вхідного string
call prints
mov rdi, input ; друк вхідного буфера
call prints
mov rdi,NL ; друк NL
call prints
leave
ret
;----------------------------------------
reads:
section .data
section .bss
.inputchar resb 1
section .text
push rbp
mov rbp, rsp
push r12 ; зберігає викликуваний (callee saved)
push r13 ; зберігає викликуваний
push r14 ; зберігає викликуваний
mov r12, rdi ; адреса вхідного буфера
mov r13, rsi ; максимальна довжина в r13
```

```
xor r14, r14 ; лічильник символі
.readc:
mov rax, 0 ; read
mov rdi, 1 ; stdin
lea rsi, [.inputchar] ; адреса входу
mov rdx, 1 ; # символів для читання
syscall
mov al, [.inputchar] ; чи символ є NL?
cmp al, byte[NL]
je .done ; NL end
cmp al, 97 ; менший від a?
jl .readc ; ігнорувати його
cmp al, 122 ; більший від z?
jg .readc ; ігнорувати його
inc r14 ; inc лічильник
cmp r14, r13
ja .readc ; максимум буфера досягнутий, ігнорувати
mov byte [r12], al ; зберегти символ у буфер
inc r12 ; вказувати на наступний символ у буфері
jmp .readc
.done:
inc r12
mov byte [r12],0 ; додати завершальний 0 у inputbuffer
pop r14 ; викликаного збережено (callee saved)
pop r13 ; с викликаного збережено
pop r12 ; викликаного збережено
leave
ret
```

```
$ ~/asm_ld.sh 2.asm
Асемблювання 2.asm успішне - 0
Компонування 2.asm успішне - 0
$ ./2
Привіт світ!
Ваша відповідь (тільки малі букви a-z): aaabbbccdddeeefffggg
You answered: aaabbbcccd
```

3. 3.asm – створення і запис у файл

```
; Приклад програми файлового I/O. Цей приклад
; буде відкривати/створювати файл, записувати деяку інформацію у файл
; і закривати файл. Зауважимо, що ім'я файлу і записуване повідомлення
; записані у код програми (hard-coded).
section .data
; -----
; Визначення стандартних констант.
LF equ 10 ; символ нового рядка
NULL equ 0 ; 0-символ кінця символьного рядка
TRUE equ 1
FALSE equ 0
EXIT_SUCCESS equ 0 ; код успішного виходу
STDIN equ 0  ; стандартний вхід
STDOUT equ 1 ; стандартний вихід
STDERR equ 2 ; стандартний потік помилок
SYS_read equ 0  ; читання
SYS_write equ 1 ; запис
```

```
SYS_open equ 2   ; відкриття файлу
SYS_close equ 3  ; закриття файлу
SYS_fork equ 57  ; галуження fork
SYS_exit equ 60  ; завершення
SYS_creat equ 85 ; відкриття/створення файлу
SYS_time equ 201 ; отримання часу
O_CREAT equ 0x40
O_TRUNC equ 0x200
O_APPEND equ 0x400
O_RDONLY equ 000000q ; тільки читати
O_WRONLY equ 000001q ; тільки записувати
O_RDWR equ 000002q ; читати і записувати
S_IRUSR equ 00400q
S_IWUSR equ 00200q
S_IXUSR equ 00100q
; -----
; Змінні для  main.
newLine db LF, NULL
header db LF, "File Write Example."
db LF, LF, NULL
fileName db "url.txt", NULL
url db "http://www.google.com"
db LF, NULL
len dq $-url-1
writeDone db "Write Completed.", LF, NULL
fileDesc dq 0
errMsgOpen db "Error opening file.", LF, NULL
errMsgWrite db "Error writing to file.", LF, NULL
;---------------------------------------------------------
section .text
global main
main:
; -----
; висвітити рядок заголовку...
mov rdi, header
call printString
; -----
; Спроба відкрити файл.
; Використання системних викликів для відкриття файлу
; Системний виклик - Open/Create
; rax = SYS_creat (відкриття/створення файлу)
; rdi = адресу стрічки з іменем файлу
; rsi = атрибути файлу (наприклад, тільки читання і т. д.)
; Повернення:
; якщо error -> eax < 0
; якщо success -> eax = номер файлового дескриптора
; Файловий дескриптор вказує блок контролю файлів(FCB).
; FCB підтримується ОС.
; Файловий дескриптор використовується у всіх наступних
; файлових операціях (read, write, close).
openInputFile:
mov rax, SYS_creat ; відкриття/створення файлу
mov rdi, fileName  ; стрічка з іменем файлу
mov rsi, S_IRUSR | S_IWUSR ; дозвіл на read/write
syscall ; звернення до ядра
```

```
cmp rax, 0 ; перевірка на успіх
jl errorOnOpen
mov qword [fileDesc], rax ; збереження дескриптора
; -----
; Запис у файл.
; У цьому прикладі, символи для запису у файл знаходяться у
; наперед визначеній стрічці, яка містить URL.
; Системний виклик - write
; rax = SYS_write
; rdi = файловий дескриптор
; rsi = адреса символів для запису
; rdx = лічильник символів для запису
; Повернення:
; якщо error -> rax < 0
; якщо success -> rax = число дійсно прочитаних символів
mov rax, SYS_write
mov rdi, qword [fileDesc]
mov rsi, url
mov rdx, qword [len]
syscall
cmp rax, 0
jl errorOnWrite
mov rdi, writeDone
call printString
; -----
; Close the file.
; System Service - close
; rax = SYS_close
; rdi = file descriptor
mov rax, SYS_close
mov rdi, qword [fileDesc]
syscall
jmp exampleDone
; -----
; Помилка відкриття файлу.
; rax містить помилку, яка у цьому прикладі не використовується
errorOnOpen:
mov rdi, errMsgOpen
call printString
jmp exampleDone
; -----
; Помилка запису.
; rax містить помилку, яка у цьому прикладі не використовується
errorOnWrite:
mov rdi, errMsgWrite
call printString
jmp exampleDone
; -----
; Приклад виконання програми.
exampleDone:
mov rax, SYS_exit
mov rdi, EXIT_SUCCESS
syscall
; ********************************************************
; Узагальнена процедура друку повідомлення у консоль.
```

```
; Стрічка має завершуватися на NULL.
; Алгоритм:
; Кількість символів у рядку (крім NULL)
; Використовується syscall для виведення символів
; Arguments:
; 1) адреса, стрічка
; Повернення: нічого
global printString
printString:
push rbp
mov rbp, rsp
push rbx
; Підрахунок символів у стрічці.
mov rbx, rdi
mov rdx, 0
strCountLoop:
cmp byte [rbx], NULL
je strCountDone
inc rdx
inc rbx
jmp strCountLoop
strCountDone:
cmp rdx, 0
je prtDone
; Виклик  ОС для виведення стрічки.
mov rax, SYS_write ; code for write()
mov rsi, rdi ; addr of characters
mov rdi, STDOUT ; file descriptor
; кількість встановлена вище
syscall ; system call
; Рядок надруковано, повернутися до процедури виклику.
prtDone:
pop rbx
pop rbp
ret
; ******************************************************

$ ~/asm_ld.sh 3.asm
Асемблювання 3.asm успішне - 0
Компонування 3.asm успішне — 0
$ ./3
File Write Example.
Write Completed.
$ cat url.txt
http://www.google.com
```

4. 4.asm – читання і друк створеного файлу в п.3

```
; Цей приклад буде відкривати файл, читати його вміст і
; записати у консоль.
; Ця процедура також надає кілька дуже простих прикладів
; щодо обробки різних помилок системних служб.
; ----------------------------------------------------
section .data
; -----
; Визначення стандартних констант.
```

```
LF equ 10 ; символ нового рядка
NULL equ 0 ; символ кінця стрічки
TRUE equ 1
FALSE equ 0
EXIT_SUCCESS equ 0 ; код успіху
STDIN equ 0  ;  стандартний вхід
STDOUT equ 1 ;  стандартний вихід
STDERR equ 2 ;  стандартний потік помилок
SYS_read equ 0   ; читати
SYS_write equ 1  ; записувати
SYS_open equ 2   ; файл відкрити
SYS_close equ 3  ; файл закрити
SYS_fork equ 57  ; галуження fork
SYS_exit equ 60  ; завершити
SYS_creat equ 85 ; відкрити/створити файл
SYS_time equ 201 ; отримати час
O_CREAT equ 0x40
O_TRUNC equ 0x200
O_APPEND equ 0x400
O_RDONLY equ 000000q ; тільки читати
O_WRONLY equ 000001q ; тільки писати
O_RDWR equ 000002q   ; читати і писати
S_IRUSR equ 00400q
S_IWUSR equ 00200q
S_IXUSR equ 00100q
; -----
; Змінні/константи для  main.
BUFF_SIZE equ 255
newLine db LF, NULL
header db LF, "File Read Example."
db LF, LF, NULL
fileName db "url.txt", NULL
fileDesc dq 0
errMsgOpen db "Error opening the file.", LF, NULL
errMsgRead db "Error reading from the file.", LF, NULL
; ------------------------------------------------------
section .bss
readBuffer resb BUFF_SIZE
; ------------------------------------------------------
section .text
global main
main:
; -----
; Display header line...
mov rdi, header
call printString
; -----
; Спроба відкрити файл - використати системний виклик для відкриття файлу
; Системний виклик - Open
; rax = SYS_open
; rdi = адреса стрічки із іменем файлу
; rsi = атрибути (наприклад, read only, тощо)
; Повернення:
; якщо error -> eax < 0
; якщо success -> eax = номер файлового дескриптора
```

```
; Файловий дескриптор вказує блок контролю файлів(FCB).
; FCB підтримується ОС.
; Файловий дескриптор використовується у всіх наступних
; файлових операціях (read, write, close).
openInputFile:
mov rax, SYS_open ; відкрити файл
mov rdi, fileName ; стрічка з іменем файлу
mov rsi, O_RDONLY ; доступ тільки на читання
syscall ; виклик ядра
cmp rax, 0 ; перевірка успіху
jl errorOnOpen
mov qword [fileDesc], rax ; збереження дескриптора
; -----
; Читання з файлу.
; У цьому прикладі файл маж тільки 1 рядок.
; Системний виклик - Read
; rax = SYS_read
; rdi = дескриптор файлу
; rsi = адреса місця розміщення даних
; rdx = кількість символів для читання
; Повернення:
; якщо error -> rax < 0
; якщо success -> rax = кількість фактично прочитаних символів
mov rax, SYS_read
mov rdi, qword [fileDesc]
mov rsi, readBuffer
mov rdx, BUFF_SIZE
syscall
cmp rax, 0
jl errorOnRead
; -----
; Друк буфера.
; додати NULL для рядка друку
mov rsi, readBuffer
mov byte [rsi+rax], NULL
mov rdi, readBuffer
call printString
printNewLine
; -----
; Закриття файлу.
; Системний виклик - close
; rax = SYS_close
; rdi = дескриптор файлу
mov rax, SYS_close
mov rdi, qword [fileDesc]
syscall
jmp exampleDone
; -----
; Помилка відкриття файлу.
; eax містить помилку, яка не використовується у цьому прикладі
errorOnOpen:
mov rdi, errMsgOpen
call printString
jmp exampleDone
; -----
```

```
; Помилка читання.
; eax містить помилку, яка не використовується у цьому прикладі
errorOnRead:
mov rdi, errMsgRead
call printString
jmp exampleDone
; -----
; Приклад виконання програми.
exampleDone:
mov rax, SYS_exit
mov rdi, EXIT_SUCCESS
syscall
; ********************************************************
; Узагальнена процедура друку повідомлення у консоль.
; Стрічка має завершуватися на NULL.
; Алгоритм:
; Кількість символів у рядку (крім NULL)
; Використовується syscall для виведення символів
; Arguments:
; 1) адреса, стрічка
; Повернення: нічого
global printString
printString:
push rbp
mov rbp, rsp
push rbx
; -----
; Підрахунок символів у стрічці.
mov rbx, rdi
mov rdx, 0
strCountLoop:
cmp byte [rbx], NULL
je strCountDone
inc rdx
inc rbx
jmp strCountLoop
strCountDone:
cmp rdx, 0
je prtDone
; -----
; Виклик ядра для виведення стрічки.
mov rax, SYS_write ; code for write()
mov rsi, rdi ; addr of characters
mov rdi, STDOUT ; file descriptor
; кількість встановлена вище
; count set above
syscall ; system call
; -----
; Рядок надруковано, повернутися до процедури виклику.
prtDone:
pop rbx
pop rbp
ret

$ ~/asm_ld.sh 4.asm
```

```
$ ./4
File Read Example.
http://www.google.com
```

5.5.asm – робота з файлами

У цій програмі виконуються наступні дії з файлом:

1. Створення файлу, а потім запис даних у файл.
2. Перезапис частину вмісту файлу.
3. Додавання даних до файлу.
4. Запис даних в певну позицію у файлі.
5. Читання даних з файлу.
6. Читання даних з певної позиції у файлі.
7. Вилучення файлу.

```
; file.asm
section .data
; вирази, що використовуються для умовного асемблювання
CREATE equ 1
OVERWRITE equ 1
APPEND equ 1
O_WRITE equ 1
READ equ 1
O_READ equ 1
DELETE equ 1
; символи системних викликів
NR_read equ 0
NR_write equ 1
NR_open equ 2
NR_close equ 3
NR_lseek equ 8
NR_create equ 85
NR_unlink equ 87
; створення і статус прапорів
O_CREAT equ 00000100q
O_APPEND equ 00002000q
; access mode
O_RDONLY equ 000000q
O_WRONLY equ 000001q
O_RDWR equ 000002q
; створення режиму допусків
S_IRUSR equ 00400q ;user read permission
S_IWUSR equ 00200q ;user write permission
NL equ 0xa
bufferlen equ 64
fileName db "testfile.txt",0
FD dq 0 ; file descriptor
text1 db "1. Hello...to everyone!",NL,0
len1 dq $-text1-1 ;remove 0
text2 db "2. Here I am!",NL,0
len2 dq $-text2-1 ;remove 0
text3 db "3. Alife and kicking!",NL,0
len3 dq $-text3-1 ;remove 0
text4 db "Adios !!!",NL,0
len4 dq $-text4-1
```

```
error_Create db "error creating file",NL,0
error_Close db "error closing file",NL,0
error_Write db "error writing to file",NL,0
error_Open db "error opening file",NL,0
error_Append db "error appending to file",NL,0
error_Delete db "error deleting file",NL,0
error_Read db "error reading file",NL,0
error_Print db "error printing string",NL,0
error_Position db "error positioning in file",NL,0
success_Create db "File created and opened",NL,0
success_Close db "File closed",NL,NL,0
success_Write db "Written to file",NL,0
success_Open db "File opened for R/W",NL,0
success_Append db "File opened for appending",NL,0
success_Delete db "File deleted",NL,0
success_Read db "Reading file",NL,0
success_Position db "Positioned in file",NL,0
section .bss
buffer resb bufferlen
section .text
global main
main:
push rbp
mov rbp,rsp
%IF CREATE
;СТВОРЕННЯ І ВІДКРИТТЯ ФАЙЛУ, ТА ЗАКРИТТЯ --------------------
; створення і відкриття файлу
mov rdi, fileName
call createFile
mov qword [FD], rax ; save descriptor
; записати у файл #1
mov rdi, qword [FD]
mov rsi, text1
mov rdx, qword [len1]
call writeFile
; закрити файл
mov rdi, qword [FD]
call closeFile
%ENDIF
%IF OVERWRITE
;ВІДКРИТИ І ПЕРЕЗАПИСАТИ ФАЙД, ПОТІМ ЗАКРИТИ -----------------
; відкрити файл
mov rdi, fileName
call openFile
mov qword [FD], rax ; save file descriptor
; записати у файл #2 ПЕРЕЗАПИСАНО!
mov rdi, qword [FD]
mov rsi, text2
mov rdx, qword [len2]
call writeFile
; закрити файл file
mov rdi, qword [FD]
call closeFile
%ENDIF
%IF APPEND
```

```
;ВІДКРИТИ І ДОБАВИТИ У ФАЙЛ, ПОТІМ ЗАКРИТИ ----------------
; відкрити файл для добавлення
mov rdi, fileName
call appendFile
mov qword [FD], rax ; save file descriptor
; записати у файл #3 ДОБАВЛЕНО!
mov rdi, qword [FD]
mov rsi, text3
mov rdx, qword [len3]
call writeFile
; закрити файл
mov rdi, qword [FD]
call closeFile
%ENDIF
%IF O_WRITE
;ВІДКРИТИ І ПЕРЕЗАПИСАТИ ІЗ ЗМІЩЕННЯМ, ПОТІМ ЗАКРИТИ ----
; відкрити файл для запису
mov rdi, fileName
call openFile
mov qword [FD], rax ; save file descriptor
; позиціювати файл із зміщенням
mov rdi, qword[FD]
mov rsi, qword[len2] ;offset at this location
mov rdx, 0
call positionFile
; записати у файл із зміщенням
mov rdi, qword[FD]
mov rsi, text4
mov rdx, qword [len4]
call writeFile
; закрити файл
mov rdi, qword [FD]
call closeFile
%ENDIF
%IF READ
;ВІДКРИТИ І ПРОЧИТАТИ ФАЙЛ, ПОТІМ ЗАКРИТИ ----------------
; відкрити файл для читання
mov rdi, fileName
call openFile
mov qword [FD], rax ; save file descriptor
; читати з файлу
mov rdi, qword [FD]
mov rsi, buffer
mov rdx, bufferlen
call readFile
mov rdi,rax
call printString
; закрити файл
mov rdi, qword [FD]
call closeFile
%ENDIF
%IF O_READ
;ВІДКРИТИ І ЧИТАТИ ФАЙЛ ІЗ ЗМІЩЕННЯМ, ПОТІМ ЗАКРИТИ ФАЙЛ -----
; відкрити файл для читання
mov rdi, fileName
```

```
call openFile
mov qword [FD], rax ; save file descriptor
; позиціювати файл із зміщенням
mov rdi, qword[FD]
mov rsi, qword[len2] ;skip the first line
mov rdx, 0
call positionFile
; читати з файлу
mov rdi, qword [FD]
mov rsi, buffer
mov rdx, 10 ;число прочитаних символів
call readFile
mov rdi,rax
call printString
; закрити файл
mov rdi, qword [FD]
call closeFile
%ENDIF
%IF DELETE
;ВИЛУЧИТИ ФАЙЛ ---------------------------------
; вилучити файл РОЗКОМЕНТУВАТИ НАСТУПНІ РЯДКИ
mov rdi, fileName
call deleteFile
%ENDIF
leave
ret
; ФУНКЦІЇ МАНІПУЛЮВАННЯ ФАЙЛОМ-------------------
;-----------------------------------------------
global readFile
readFile:
mov rax, NR_read
syscall ; rax містить число прочитаних символів
cmp rax, 0
jl readerror
mov byte [rsi+rax],0 ; додати в кінець символ 0
mov rax, rsi
mov rdi, success_Read
push rax ; caller saved
call printString
pop rax ; caller saved
ret
readerror:
mov rdi, error_Read
call printString
ret
;-----------------------------------------------
global deleteFile
deleteFile:
mov rax, NR_unlink
syscall
cmp rax, 0
jl deleteerror
mov rdi, success_Delete
call printString
ret
```

```
deleteerror:
mov rdi, error_Delete
call printString
ret
;-----------------------------------------------
global appendFile
appendFile:
mov rax, NR_open
mov rsi, O_RDWR|O_APPEND
syscall
cmp rax, 0
jl appenderror
mov rdi, success_Append
push rax ; caller saved
call printString
pop rax ; caller saved
ret
appenderror:
mov rdi, error_Append
call printString
ret
;-----------------------------------------------
global openFile
openFile:
mov rax, NR_open
mov rsi, O_RDWR
syscall
cmp rax, 0
jl openerror
mov rdi, success_Open
push rax ; caller saved
call printString
pop rax ; caller saved
ret
openerror:
mov rdi, error_Open
call printString
ret
;-----------------------------------------------
global writeFile
writeFile:
mov rax, NR_write
syscall
cmp rax, 0
jl writeerror
mov rdi, success_Write
call printString
ret
;-----------------------------------------------
global positionFile
positionFile:
mov rax, NR_lseek
syscall
cmp rax, 0
jl positionerror
```

```
mov rdi, success_Position
call printString
ret
positionerror:
mov rdi, error_Position
call printString
ret
;-----------------------------------------
global closeFile
closeFile:
mov rax, NR_close
syscall
cmp rax, 0
jl closeerror
mov rdi, success_Close
call printString
ret
closeerror:
mov rdi, error_Close
call printString
ret
;-------------------------------------------------
global createFile
createFile:
mov rax, NR_create
mov rsi, S_IRUSR |S_IWUSR
syscall
cmp rax, 0 ; дескриптор файлу у rax
jl createerror
mov rdi, success_Create
push rax ; caller saved
call printString
pop rax ; caller saved
ret
createerror:
mov rdi, error_Create
call printString
; PRINT FEEDBACK
;-----------------------------------------------
global printString
printString:
; Count characters
mov r12, rdi
mov rdx, 0
strLoop:
cmp byte [r12], 0
je strDone
inc rdx ; довжина в rdx
inc r12
jmp strLoop
strDone:
cmp rdx, 0 ; відсутність стрічок (довжина 0)
je prtDone
mov rsi,rdi
mov rax, 1
```

```
mov rdi, 1
syscall
prtDone:
ret
```

```
$ ~/asm_ld.sh 5.asm
Асемблювання 5.asm успішне - 0
Компонування 5.asm успішне - 0
$ ./5
File created and opened
Written to file
File closed

File opened for R/W
Written to file
File closed

File opened for appending
Written to file
File closed

File opened for R/W
Positioned in file
Written to file
File closed

File opened for R/W
Reading file
2. Here I am!
Adios !!!
3. Alife and kicking!
File closed

File opened for R/W
Positioned in file
Reading file
Adios !!!
File closed

File deleted
```

Лабораторна робота 10

Модульне програмування

Мета роботи: вивчення модульного програмування.

Теоретичні відомості

1. Багатомодульні програми

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

Нижче наведено проста основна програма `main`, яка викликає функцію мови асемблера, `stats()`, щоб обчислити суму та ціле середнє для списку цілих чисел зі знаком. Основна програм `main` і функція знаходяться в різних початкових файлах.

1.2.1 Основна програма

```
; ---------------------------------------------------------
; Секція даних
section .data
; -----
; Визначення стандартних констант
LF equ 10  ; \n - перехід на новий рядок
NULL equ 0 ; кінець символьної стрічки
```

```
TRUE equ 1
FALSE equ 0
EXIT_SUCCESS equ 0 ; код успішного завершення
SYS_exit equ 60    ; завершення програми
; -----
; Declare the data
lst1 dd 1, -2, 3, -4, 5
     dd 7, 9, 11
len1 dd 8
lst2 dd 2, -3, 4, -5, 6
     dd -7, 10, 12, 14, 16
len2 dd 10
msg db "Сума, середнє",0
fmtstr db "%s",10,0
fmtint db "%d, %d",10,0
;----------------------------------------
; секція буферів (неініціалізованих даних)
section .bss
sum1 resd 1
ave1 resd 1
sum2 resd 1
ave2 resd 1
; ------------------------------------------------------------
extern printf
extern sum
section .text
global main
main:
push rbp
mov rbp,rsp

; ----
; Виклик функції з мов високого рівня
;   sum(lst, len, &sum, &ave);
mov rdi, lst1 ; набір даних 1
mov esi, dword [len1]
mov rdx, sum1
mov rcx, ave1
call sum
; друк msg
mov rax,0
mov rdi,fmtstr
mov rsi,msg
call printf
; друк суми, середнього
mov rax,0
mov rdi,fmtint
mov rsi,[sum1]
mov rdx,[ave1]
call printf
;-----------
mov rdi, lst2 ; набір даних 2
mov esi, dword [len2]
mov rdx, sum2
mov rcx, ave2
```

```
call sum
; друк msg
mov rax,0
mov rdi,fmtstr
mov rsi,msg
call printf
; друк суми, середнього
mov rax,0
mov rdi,fmtint
mov rsi,[sum2]
mov rdx,[ave2]
call printf
; ------------------
; Завершення програми
exampleDone:
mov rax, SYS_exit
mov rdi, EXIT_SUCCESS
syscall
```

1.2.2 Підпрограма

```
; Секція даних
section .data
; *********************************************************
section .text
; -------------------------------------------------
; Функція знаходження суми і середнього для
; переданого списку знакових цілих чисел
; Виклик:
; sum(lst, len, &sum, &ave);
; Передані аргументи:
; 1) rdi - адреса масиву
; 2) rsi - довжина переданого масиву
; 3) rdx - адреса змінної для суми
; 4) rcx - адреса змінної для середнього
; Повернення:
; сума цілих (через посилання - адресу змінної)
; середнє цілих (через посилання - адресу змінної)
global sum
sum:
push r12
; -----
; Обчислення і повернення суми
mov r11, 0  ; i=0
mov r12d, 0 ; sum=0
sumLoop:
mov eax, dword [rdi+r11*4] ; get lst[i]
add r12d, eax ; оновлення суми
inc r11 ; i++
cmp r11, rsi
jb sumLoop
mov dword [rdx], r12d ; повернення суми
; -----
; Обчислення і повернення середнього
mov eax, r12d
cdq
```

```
idiv esi
mov dword [rcx], eax ; повернення середнього
; -----
; Повернення до викликаючої функції
pop r12
ret
```

1.2.3. Асемблювання і компонування

```
$ nasm -g -F dwarf -f elf64 main.asm -l main.lst
$ nasm -g -F dwarf -f elf64 sum.asm  -l sum.lst
$ gcc -o main main.o sum.o -no-pie
$ ./main
Сума, середнє
30, 3
Сума, середнє
49, 4
```

2. Виклик функцій асемблера з мов високого рівня

При виклику будь-яких функцій, які знаходяться в окремому початковому файлі, компілятор потрібно повідомити, що сирцевий код функції (або функцій) є зовнішнім щодо основної програми. Для цього використовується оператор `extern` мови C або C++. Інші мови мають аналогічні оператори. Для мови високого рівня оператор `extern` включає прототип функції, який дозволяє компілятору перевірити параметри функції та асоційовані типи.

2.1. Виклик функцій асемблера з C++

```
#include <iostream>
using namespace std;
extern "C" void sum(int[], int, int *, int *);

int main()
{
int lst[] = {1, -2, 3, -4, 5, 7, 9, 11};
int len = 8;
int summ, ave;
    sum(lst, len, &summ, &ave);
cout << "Sum:" << endl;
cout << " Сума = " << summ << endl;
cout << " Середнє = " << ave << endl;
return 0;
}
```

```
$ g++ -g -Wall -c main.cpp
$ nasm -g -F dwarf -f elf64 sum.asm -l sum.lst
$ g++ -g -no-pie -o main main.o sum.o
$ ./main
Sum:
 Сума = 30
 Середнє = 3
```

2.2. Виклик функцій асемблера з Сі

```
#include<stdio.h>
extern void sum(int[], int, int *, int *);
int main()
{
int lst[] = {1, -2, 3, -4, 5, 7, 9, 11};
int len = 8;
int summ, ave;
    sum(lst, len, &summ, &ave);
printf ("Sum:\n");
printf (" Сума = %d \n", summ);
printf (" Середнє = %d \n", ave);
return 0;
}
```

```
$ gcc -g -Wall -c main.c
$ nasm -g -F dwarf -f elf64 sum.asm -l sum.lst
$ gcc -g -no-pie -o main main.o sum.o
$ ./sum
Sum:
 Сума = 30
 Середнє = 3
```

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 модулів бібліотеки `mylib.a` за допомогою системного компонувача `ld`:

```
$ ld main.o mylib.a –o main
```

Контрольні запитання.

1. В чому суть модульного програмування.
2. Що оголошують директиви `global` і `extern`.
3. Послідовність створення багатомодульної програми.
4. Як можна передати аргументи у підпрограму.
5. Які регістри використовуються для передачі параметрів у підпрограму.
6. Для чого використовуються бібліотеки об’єктних модулів.
7. Створення і використання бібліотеки об’єктних модулів.

Практична частина

Завдання.

1. Написати Nasm програму, яка використовує дві функції. Одну для зчитування з консолі послідовності цілих чисел, а іншу для додавання цих чисел і виведення суми.

2. Написати Nasm програму, яка використовує окремі функції для додавання і множення цілих чисел та виведення результату. Масиви цілих чисел оголосити в секції `.data`.

3. Написати Nasm програму, яка використовує окремі функції для додавання і ділення цілих чисел та виведення результату. Масиви цілих чисел оголосити в секції `.data`.

4. Написати Nasm програму, яка використовує окремі функції для сортування цілих чисел та виведення результату. Масиви цілих чисел оголосити в секції `.data`.

5. Написати Nasm програму, яка використовує дві функції. Одну для зчитування з консолі послідовності дійсних чисел, а іншу для додавання цих чисел і виведення результату.

6. Написати Nasm програму, яка використовує дві функції. Одну для зчитування з командного рядка послідовності дійсних чисел, а іншу для знаходження максимального числа і виведення результату.

7. Написати Nasm програму, яка використовує дві функції. Одну для зчитування з файлу послідовності дійсних чисел, а іншу для реверсування послідовності зчитаних чисел і виведення результату.

8. Напасати Nasm програму, яка використовує дві функції. Одну для зчитування з консолі двох чисел і символу арифметичної операції (+, -, *, /), яку необхідно виконати над числами, та іншу для виконання арифметичної операції та виведення результату у консоль.

9. Написати Nasm програму, яка використовує дві функції. Одну для читання текстового файлу і виведення у консоль, а іншу для запису зчитаного тексту у зворотному порядку у файл і виведення у консоль.

9. Написати Nasm програму, яка використовує дві функції. Одну для читання текстового файлу і виведення у консоль, а іншу для сортування зчитаного тексту і виведення у консоль.

10. Написати Nasm програму, яка використовує дві функції. Одну для читання текстового файлу і виведення у консоль, а іншу для заміни всіх символів “a” на “o” і запису результату у файл.

10. Написати Nasm програму, яка використовує дві функції. Одну для зчитування з консолі команди Linux, а іншу для виконання зчитаної команди.

11. Написати Сі програму, яка викликає NASM функцію для виконання команду Linux “ls –al”.

12. Написати C++ програму, яка викликає NASM функцію для виконання команду Linux “pwd”.

13. Написати Сі програму, яка зчитує з консолі команду Linux, а NASM функція її виконує.

14. Написати C++ програму, в якій оголосити вектор цілих чисел і передати у NASM функцію. NASM функція множить елементи вектора на 2 і повертає у C++ програму, яка друкує отриманий вектор.

15. Написати C++ програму, яка використовує бібліотеку об’єктних модулів NASM функцій додавання, віднімання, множення і ділення цілих чисел. Оголосити у C++ два вектори цілих чисел і виконати над ними операції додавання, віднімання, множення, ділення з використанням Nasm модулів та вивести отримані результати у консоль.

Примітка. Номер варіанта завдання вибирається за порядковим номером студента у журналі групи.

Звіт з лабораторної роботи має містити:

1. Назва групи, П.І.Б. студента, завдання до роботи.
2. Блок схема алгоритму програми виконана згідно стандартів.
3. Текст програми із поясненнями.
4. Роздруки екранів стану пам'яті і регістрів з результатами виконання програми у консолі або налагоджувачі `KDbg/dbg`.

Приклади для самостійної роботи

1. read_a.c - 64-бітова асемблерна функція read_doubles()
яка читає числа doubles з stdin *

```
; doubles.asm – читання масиву doubles з файлу
segment .data
format  db "%lf", 0    ; format for fscanf()
segment .bss
segment .text
        global  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 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)

doubles:
        push    rbp
        mov     rbp,rsp
        sub     rsp, SIZEOF_DOUBLE      ; визначення одного double у стеку
        push    rsi                     ; збереження rsi
        mov     rsi, ARRAYP             ; rsi = ARRAYP
        xor     rdx, rdx                ; rdx = індекс масиву (початково 0)
while_loop:
        cmp     rdx, ARRAY_SIZE         ;  rdx < ARRAY_SIZE?
        jnl     short quit              ; якщо ні, quit loop

; виклик fscanf() для читання double у TEMP_DOUBLE
; fscanf() може змінити rdx так що збережемо його
        push    rdx                     ; збереження edx
        push    FP                      ; push &TEMP_DOUBLE
        push    qword format            ; push &format
```

```
        push    TEMP_DOUBLE             ; push file pointer
        call    fscanf
        add     rsp, 24
        pop     rdx                     ; відновлення rdx
        cmp     rax, 1                  ; чи fscanf повернув 1?
        jne     short quit              ; якщо ні, quit loop

; копіювання TEMP_DOUBLE у ARRAYP[rdx]
        mov     rax, [ebp - 8]
        mov     [rsi + 8*rdx], rax
        inc     rdx
        jmp     while_loop
quit:
        pop     rsi                     ; відновлення rsi
        mov     rax, rdx                ; збереження return значення у rax
        mov     rsp, rbp
        pop     rbp
        ret
```

```
#include <stdio.h>
extern doubles( FILE *, double[], int );

#define MAX 100
int main() {
  int i,n;
  double a[MAX];
  n = doubles(stdin, a, MAX);
  for( i=0; i < n; i++ )
    printf("%3d %g\n", i, a[i]);
  return 0;
}
```

```
$ gcc -q -Wall -c read_a.c
$ nasm -g -F dwarf -f elf64 doubles.asm -l doubles.lst
$ gcc -o read_a read_a.o doubles_o no-pie

$ ./read_a
1.234
123.456
12345.6789
  0 1.234
  1 123.456
  2 12345.7
```

Лабораторна робота 11

Операції з плаваючою крапкою

Мета роботи: вивчення практичного застосування `mmx` команд і команд x87 FPU для виконання операцій з плаваючою крапкою.

Теоретичні відомості

1. Значення із плаваючою крапкою

Значення з плаваючою крапкою зазвичай задають як одинарну точність (32 біти) або подвійну точність (64 біти). У Сі і C++ змінні з плаваючою крапкою одинарної точності зазвичай оголошуються як типи **float**, а змінні з плаваючою крапкою подвійної точності – оголошуються як типи **double**. Команди мови асемблера використовують кваліфікатор **s** для посилання на одиничну точність і кваліфікатор **d** для позначення подвійної точності.

2. Регістри з плаваючою крапкою

МП Intel x64 мають співпроцесор x87 для операцій з плаваючою крапкою. Однак з добавлення в архітектуру МП x86 розширень SSE, які мають розширені можливості з обробки чисел з плаваючою крапкою, x87 FPU втратив свою актуальність.

Існує набір спеціальних регістрів XMM, які використовуються для підтримки операцій з плаваючою крапкою. Регістри XMM є 128-бітовими та 256-бітовими на пізніших моделях процесорів. Є 16 регістрів XMM з іменами від xmm0 до xmm15. В звичайних командах SISD використовуються лише молодші 32 або 64 біти.

3. Команди пересилання даних

Загальна форма команд пересилання даних:

Команда	Пояснення
`movss <dest>, <src>`	Копіює 32-бітовий операнд джерело у 32-бітовий операнд приймач. Обидва операнди не можуть задавати пам’ять. Операнди не можуть бути безпосередніми значеннями.

Приклад:	`movss xmm0, dword [x]`
	`movss dword [fltSVar], xmm1`
	`movss xmm3, xmm2`

`movsd <dest>, <src>`	Копіює 64-бітовий операнд джерело у 64-бітовий операнд приймач. Обидва операнди не можуть задавати пам’ять. Операнди не можуть бути безпосередніми значеннями.

Приклад:	`movsd xmm0, qword [x]`
	`movsd qword [fltDVar], xmm1`
	`movsd xmm3, xmm2`

Наприклад, є оголошення даних:

```
fSVar1 dd 3.14
fSVar2 dd 0.0
fDVar1 dq 6.28
fDVar2 dq 0.0
```

Операції присвоєння

```
fSVar2 = fSVar1 ; змінна одинарної точності
fDVar2 = fDVar1 ; змінна подвійної точності
```

виконає наступна послідовність команд

```
movss xmm0, dword [fSVar1]
movss dword [fSVar2], xmm0 ; fSVar2 = fSVar1
movsd xmm1, qword [fDVar1]
movsd qword [fDVar2], xmm1 ; fDVar2 = fDVar1
movss xmm2, xmm0 ; xmm2 = xmm0 (32-bit)
movsd xmm3, xmm1 ; xmm3 = xmm1 (64-bit)
```

4. Перетворення цілих значень у значення з плаваючою крапкою

Якщо цілі значення потрібні при обчисленнях із плаваючою крапкою, то їх потрібно перетворити в значення з плаваючою крапкою. При обчисленнях із плаваючою крапкою значення мають бути узгоджені як за розміром, так і за типом.

Для перетворення розмірів і типів використовуються наступні команди:

Команда	Пояснення

`cvtss2sd <RXdest>, <src>`	Перетворює 32-бітовий операнд джерело із плаваючою крапкою у 64-бітовий операнд приймач із плаваючою крапкою.
	Операнд приймач має бути регістром з плаваючою крапкою.
	Операнд джерело не може бути безпосереднім значенням.
	Приклад:
	`cvtsd2sd xmm0, qword [fltDVar]`
	`cvtsd2sd xmm1, rax`
	`cvtsd2sd xmm3, xmm2`

`cvtsd2ss <RXdest>, <src>`	Перетворює 64-бітовий операнд джерело із плаваючою крапкою у 32-бітовий операнд приймач із плаваючою крапкою.
	Операнд приймач має бути регістром з плаваючою крапкою.
	Операнд джерело не може бути безпосереднім значенням.
	Приклад:
	`cvtss2sd xmm0, dword [fltSVar]`
	`cvtss2sd xmm3, eax`
	`cvtss2sd xmm3, xmm2`

`cvtss2si <reg>, <src>`	Перетворює 32-бітовий операнд джерело із плаваючою крапкою у 32-бітовий цілий операнд приймач. Операнд приймач має бути регістром. Операнд джерело не може бути безпосереднім значенням. Приклад: `cvtss2si xmm1, xmm0`

	cvtss2si eax, xmm0 cvtss2si eax, dword [fltSVar]
cvtsd2si <reg>, <src> Приклад:	Перетворює 64-бітовий операнд джерело із плаваючою крапкою у 32-бітовий цілий операнд приймач. Операнд приймач має бути регістром. Операнд джерело не може бути безпосереднім значенням. cvtsd2si xmm1, xmm0 cvtsd2si eax, xmm0 cvtsd2si eax, qword [fltDVar]
cvtsi2ss <RXdest>, <src> Приклад:	Перетворює 32-бітовий цілий операнд джерело у 32-бітовий операнд приймач із плаваючою крапкою. Операнд приймач має бути регістром із плаваючою крапкою. Операнд джерело не може бути безпосереднім значенням. cvtsi2ss xmm0, eax cvtsi2ss xmm0, dword [fltDVar]
cvtsi2sd <RXdest>, <src> Приклад:	Перетворює 32-бітовий цілий операнд джерело у 64-бітовий операнд приймач із плаваючою крапкою. Операнд приймач має бути регістром із плаваючою крапкою. Операнд джерело не може бути безпосереднім значенням. cvtsi2sd xmm0, eax cvtsi2sd xmm0, dword [fltDVar]

5. Арифметичні інструкції

Арифметичні операції додавання, віднімання, множення, ділення виконуються над числами одинарної або подвійної точності із значеннями з плаваючою крапкою.

5.1. Додавання значень з плаваючою крапкою

Загальна форма команд додавання значень із плаваючою крапкою

```
addss <RXdest>, <src>
addsd <RXdest>, <src>
```

Команда	Пояснення
addss <RXdest>, <src> Приклад:	Додає два 32-бітові операнди із плаваючою крапкою і результат поміщає у операнд приймач <RXdest> = <RXdest> + <src> Операнд приймач має бути регістром із плаваючою крапкою Операнд джерело не може бути безпосереднім значенням. addss xmm0, xmm3 addss xmm5, dword [fsVar]
addsd <RXdest>, <src>	Додає два 64-бітові операнди із плаваючою крапкою

	і результат поміщає у операнд приймач <RXdest> = <RXdest> + <src> Операнд приймач має бути регістром із плаваючою крапкою Операнд джерело не може бути безпосереднім значенням. Приклад: addsd xmm0, xmm3 addsd xmm5, qword [fDVar]

Наприклад, є наступне оголошення (використовується у всіх прикладах):

```
fSNum1 dd 43.75
fSNum2 dd 15.5
fSAns dd 0.0

fDNum3 dq 200.12
fDNum4 dq 73.2134
fDAns dq 0.0
```

тоді, наступні операції

```
fSAns = fSNum1 + fSNum2
fDAns = fDNum3 + fDNum4
```

виконає послідовність команд:

```
; fSAns = fSNum1 + fSNum2
movss xmm0, dword [fSNum1]
addss xmm0, dword [fSNum2]
movss dword [fSAns], xmm0
; fDAns = fDNum3 + fDNum4
movsd xmm0, qword [fDNum3]
addsd xmm0, qword [fDNum4]
movsd qword [fDAns], xmm0
```


5.2. Віднімання значень з плаваючою крапкою

Загальна форма команд віднімання значень із плаваючою крапкою

```
subss <RXdest>, <src>
subsd <RXdest>, <src>
```

Команда	Пояснення
subss <RXdest>, <src>	Віднімає два 32-бітові операнди із плаваючою крапкою і результат поміщає у операнд приймач <RXdest> = <RXdest> - <src> Операнд приймач має бути регістром із плаваючою крапкою Операнд джерело не може бути безпосереднім значенням. Приклад: subss xmm0, xmm3 subss xmm5, dword [fsVar]
subsd <RXdest>, <src>	Віднімає два 64-бітові операнди із плаваючою крапкою і результат поміщає у операнд приймач <RXdest> = <RXdest> - <src>

Приклад:	Операнд приймач має бути регістром із плаваючою крапкою
	Операнд джерело не може безпосереднім значенням.
	`subsd xmm0, xmm3`
	`subsd xmm5, qword [fDVar]`

Наприклад, використовуючи вище наведені оголошення, наступні операції

```
fSAns = fSNum1 - fSNum2
fDAns = fDNum3 - fDNum4
```

виконає послідовність команд:

```
; fSAns = fSNum1 - fSNum2
movss xmm0, dword [fSNum1]
subss xmm0, dword [fSNum2]
movss dword [fSAns], xmm0
; fDAns = fDNum3 - fDNum4
movsd xmm0, qword [fDNum3]
subsd xmm0, qword [fDNum4]
movsd qword [fDAns], xmm0
```

5.3. Множення значень з плаваючою крапкою

Загальна форма команд множення значень із плаваючою крапкою

```
mulss <RXdest>, <src>
mulsd <RXdest>, <src>
```

Команда	Пояснення

`mulss <RXdest>, <src>`	Множить два 32-бітові операнди з плаваючою крапкою і результат поміщає у операнд приймач
	`<RXdest> = <RXdest> * <src>`
	Операнд приймач має бути регістром із плаваючою крапкою
	Операнд джерело не може бути безпосереднім значенням.
	Приклад:
	`mulss xmm0, xmm3`
	`mulss xmm5, dword [fsVar]`

`mulsd <RXdest>, <src>`	Множить два 64-бітові операнди із плаваючою крапкою і результат поміщає у операнд приймач
	`<RXdest> = <RXdest> * <src>`
	Операнд приймач має бути регістром із плаваючою крапкою
	Операнд джерело не може бути безпосереднім значенням.
	Приклад:
	`mulsd xmm0, xmm3`
	`mulsd xmm5, qword [fDVar]`

Наприклад, використовуючи вище наведені оголошення, наступні операції:

```
fSAns = fSNum1 * fSNum2
fDAns = fDNum3 * fDNum4
```

виконає послідовність команд

```
; fSAns = fSNum1 * fSNum2
movss xmm0, dword [fSNum1]
mulss xmm0, dword [fSNum2]
movss dword [fSAns], xmm0
; fDAns = fDNum3 * fDNum4
movsd xmm0, qword [fDNum3]
mulsd xmm0, qword [fDNum4]
movsd qword [fDAns], xmm0
```

5.4. Ділення значень з плаваючою крапкою

Загальна форма команд ділення значень із плаваючою крапкою

```
divss <RXdest>, <src>
divsd <RXdest>, <src>
```

Команда	Пояснення
divss <RXdest>, <src> Приклад:	Ділить два 32-бітові операнди з плаваючою крапкою і результат поміщає у операнд приймач <RXdest> = <RXdest> / <src> Операнд приймач має бути регістром із плаваючою крапкою Операнд джерело не може бути безпосереднім значенням. divss xmm0, xmm3 divss xmm5, dword [fsVar]
divsd <RXdest>, <src> Приклад:	Ділить два 64-бітові операнди із плаваючою крапкою і результат поміщає у операнд приймач <RXdest> = <RXdest> / <src> Операнд приймач має бути регістром із плаваючою крапкою Операнд джерело не може бути безпосереднім значенням. divsd xmm0, xmm3 divsd xmm5, qword [fDVar]

Наприклад, використовуючи вище наведені оголошення, наступні операції

```
fSAns = fSNum1 / fSNum2
fDAns = fDNum3 / fDNum4
```

виконає послідовність команд:

```
; fSAns = fSNum1 / fSNum2
movss xmm0, dword [fSNum1]
divss xmm0, dword [fSNum2]
movss dword [fSAns], xmm0
; fDAns = fDNum3 / fDNum4
movsd xmm0, qword [fDNum3]
divsd xmm0, qword [fDNum4]
movsd qword [fDAns], xmm0
```

5.5. Корінь квадратний із значень з плаваючою крапкою

Загальна форма команд ділення значень із плаваючою крапкою

```
sqrtss <RXdest>, <src>
sqrtsd <RXdest>, <src>
```

Команда	Пояснення
sqrtss <RXdest>, <src> Приклад:	Бере корінь квадратний із 32-бітового операнда джерела з плаваючою крапкою і результат поміщає у операнд приймач <RXdest> = SQRT(<src>) Операнд приймач має бути регістром із плаваючою крапкою Операнд джерело не може бути безпосереднім значенням. sqrtss xmm0, xmm3 sqrtss xmm7, dword [fsVar]
sqrtd <RXdest>, <src> Приклад:	Бере корінь квадратний із 64-бітового операнда із плаваючою крапкою і результат поміщає у операнд приймач <RXdest> = SQRT (<src>) Операнд приймач має бути регістром із плаваючою крапкою Операнд джерело не може бути безпосереднім значенням. sqrtsd xmm0, xmm3 sqrtsd xmm7, qword [fDVar]

Наприклад є наступне оголошення

```
fSNum1 dd 1213.0
fSAns  dd 0.0

fDNum3 dq 123456.123
fDAns  dq 0.0
```

тоді, наступні операції

```
fSAns = SQRT(fSNum1)
fDAns = SQRT(fDNum3)
```

виконає послідовність команд:

```
; fSAns = sqrt (fSNum1)
sqrtss xmm0, dword [fSNum1]
movss dword [fSAns], xmm0
; fDAns = sqrt(fDNum3)
sqrtsd xmm0, qword [fDNum3]
movsd qword [fDAns], xmm0
```

6. Команди керування для операцій із плаваючою крапкою

Команди керування для операцій із плаваючою крапкою стосуються таких конструкцій програмування, як оператори IF та цикли. Команда цілочислового порівняння, `cmp`, не працює із значеннями з плаваючою крапкою.

Команди порівняння з плаваючою крапкою порівнюють два значення з плаваючою крапкою. Результат порівняння зберігається в системному регістрі RFlags, а операнди не змінюється. За результатом порівняння виконується перехід за допомогою команд умовного переходу. Хоча всі порівняння з плаваючою крапкою є знаковими, використовується беззнаковий умовний перехід (`ja/jae/jb/jbe`).

Існує дві форми порівняння з плаваючою крапкою, впорядкована та невпорядкована. Впорядковані порівняння з плаваючою крапкою можуть викликати низку винятків. Невпорядковані порівняння з плаваючою крапкою можуть спричинити винятки лише для S-NaN (сигналізація не число), більш узагальнено називається NaN (не число).

Компілятор GNU C/C++ надає перевагу невпорядкованій команді порівняння з плаваючою крапкою. Тому будуть розглядатися тільки невпорядковані порівняння.

6.1. Порівняння з плаваючою крапкою

Загальна форма порівнянь з плаваючою крапкою

```
ucomiss <RXsrc>, <src>
ucomisd <RXsrc>, <src>
```

Команда	Пояснення
ucomiss <RXsrc>, <src> Приклад:	Порівнює два 32-бітові операнди з плаваючою крапкою і результат поміщає в регістр Rflags. Операнди не міняються. Операнд <RXsrc> має бути регістром із плаваючою крапкою Операнд <src> не може бути безпосереднім значенням. ucomiss xmm0, xmm3 ucomiss xmm5, dword [fsVar]
ucomisd <RXsrc>, <src> Приклад:	Порівнює два 64-бітові операнди з плаваючою крапкою і результат поміщає в регістр Rflags. Операнди не міняються. Операнд <RXsrc> має бути регістром із плаваючою крапкою Операнд <src> не може бути безпосереднім значенням. ucomisd xmm0, xmm3 ucomisd xmm5, qword [fsVar]

Для читання результатів порівняння у регістрі RFlags використовуються команди переходів для беззнакових операндів:

```
je  <label> ; якщо (a == b)
jne <label> ; якщо (a != b)
jb  <label> ; беззнакові (a < b )
jbe <label> ; беззнакові (a <= b)
ja  <label> ; беззнакові (a > b)
jae <label> ; беззнакові (a >= b)
```

Наприклад, є псевдокод для IF

```
if (fltNum > fltMax)
```

```
fltMax = fltNum;
```

і оголошення змінних

```
fltNum dq 7.5
fltMax dq 5.25
```

тоді умову порівняння IF реалізує код

```
movsd xmm1, qword [fltNum]
ucomisd xmm1, qword [fltMax]          ; if fltNum <= fltMax
jbe notNewFltMax                      ; skip set new max
movsd qword [fltMax], xmm1
notNewFltMax:
```

Наприклад, є псевдокод для IF-ELSE

```
if (y != 0.0) {
ans = x / y;
errFlg = FALSE;
} else {
errFlg = TRUE;
}
```

і оголошення змінних

```
TRUE equ 1
FALSE equ 0
fltZero dq 0.0
x dq 10.1
y dq 3.7
ans dq 0.0
errFlg db FALSE
```

тоді умову порівняння IF-ELSE реалізує код

```
movsd xmm1, qword [x]
ucomisd xmm1, qword [fltZero] ; if statement
je doElse
divsd xmm1, qword [y]
movsd dword [ans], eax
mov byte [errFlg], FALSE
jmp skpElse
doElse:
mov byte [errFlg], TRUE
skpElse:
```

Порівняння і операції з плаваючою крапкою, з причини неточного подання чисел і їх округлення, можуть давати неочікувані результати для недосвідченого користувача, наприклад:

$$\sum_{i=1}^{10} 0.1 \neq 10$$

7. Домовленості для передачі аргументів із плаваючою крапкою

При використанні регістрів з плаваючою крапкою жоден із регістрів не зберігається при виклику функції з плаваючою крапкою. Перші вісім аргументів з плаваючою крапкою

передаються через регістри xmm0 – xmm7. Будь-які додаткові аргументи розміщуються в стеку в зворотному порядку. Функція з плаваючою крапкою повертає значення через регістр xmm0.

Оскільки жоден із регістрів з плаваючою крапкою не зберігається, код має бути написаний ретельно.

Контрольні запитання.

1. Які регістри використовуються для операцій з плаваючою крапкою і яка їх розрядність.
2. Яка розрядність операндів в операціях з плаваючою крапкою.
3. Команди для виконання операцій додавання і віднімання з плаваючою крапкою.
4. Команди для виконання операцій множення і ділення з плаваючою крапкою.
5. Команда для взяття кореня квадратного для операндів з плаваючою крапкою.
6. Команди переходів за результатами порівняння для операцій із плаваючою крапкою.
7. Класифікація команд x87 FPU.
8. Арифметичні команди x87 FPU.

Практична частина

Завдання.

Написати програму з використання команд `xmm` і x87 FPU.

1. Знаходження суми і середнього значення списку доданих і від'ємних чисел з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
2. Знаходження абсолютного значення суми списку доданих і від'ємних чисел з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
3. Знаходження суми квадратів списку чисел з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
4. Знаходження суми кубів списку чисел з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
5. Знаходження суми добутків списку чисел з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
6. Знаходження суми синусів списку чисел з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
7. Знаходження суми косинусів списку чисел з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
8. Знаходження суми тангенсів списку чисел з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
9. Знаходження суми арктангенсів списку чисел з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
10. Знаходження суми двійкових логарифмів списку чисел з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
11. Знаходження суми ряду $a + a_1x + a_2x^2 + a_3x^3$ для значень a і x з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
12. Знаходження коренів рівняння $y = a + a_1x + a_2x^2$ для значень a і x з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.
13. Знаходження коренів рівняння $y = a + a_1x + a_2x^2 + a_3x^3$ для значень a і x з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.

14. Знаходження суми $y = a + a_1\sin(x) + a_2\sin(2x) + a_3\sin(3x)$ для значень a і x з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.

15. Знаходження суми $y = a + a_1\sin(x) + a_2\sin(x^2) + a_3\sin(x^3)$ для значень a і x з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.

16. Знаходження суми $y = a + a_1\sin(x^-1) + a_2\sin(x^2) + a_3\sin(x^{-3}) + a_4\sin(x^4)$ для значень a і x з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.

17. Знаходження суми $y = a + a_1\sin(x^-1) + a_2\cos(x^2) + a_3\sin(x^{-3}) + a_4\cos(x^4)$ для значень a і x з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.

18. Знаходження суми $y = a + a_1\sin(x^{-1}) + a_2/\cos(x^2) + a_3\sin(x^{-3}) + a_4/\cos(x^4)$ для значень a і x з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.

19. Знаходження суми $y = a + a_1\sin(x^{-1}) + a_2/\mathrm{tg}(x^2) + a_3\sin(x^{-3}) + a_4/\mathrm{tg}(x^4)$ для значень a і x з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.

20. Знаходження $y = \mathrm{arctg}[(a + a_1\sin(x^{-1}))/(a_2/\mathrm{tg}(x^2) + a_3\sin(x^{-3}))]$ для значень a і x з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.

21. Обчислити визначник матриці 2×2 для значень a_i з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.

22. Обчислити визначник матриці 3×3 для значень a_i з плаваючою крапкою 64-бітової розрядності. Результат вивести у консоль.

23. Обчислити значення $\sin(x) = x - x^3/6 + x^5/120 - x^7/5040 + x^9/362880$. Результат вивести у консоль.

24. Обчислити значення $\cos(x) = 1 - x^2/2 + x^4/24 - x^6/720 + x^8/40320$. Результат вивести у консоль.

25. Обчислити значення $e^x = 1 + x + x^2/2 + x^3/6 + x^4/24 + x^5/120$. Результат вивести у консоль.

Приклади

1. Обчислення суми і середнього списку чисел із плаваючою крапкою

```
; 1.asm - приклад обчислення суми і середнього списку чисел
; ********************************************************
extern printf
section .data
; -----
; Визначення констант
NULL equ 0 ; кінець стрічки
TRUE equ 1
FALSE equ 0
EXIT_SUCCESS equ 0 ; успішна операція
SYS_exit equ 60 ; код системного виклику для завершення програми
; -----
fltLst dq 21.34, 6.15, 9.12, 10.05, 7.75
       dq 1.44, 14.50, 3.32, 75.71, 11.87
       dq 17.23, 18.25, 13.65, 24.24, 8.88
length dd 15
lstSum dq 0.0
lstAve dq 0.0
fmtflt1 db "Сума %lf",10,0
fmtflt2 db "Середнє %lf",10,0
; ********************************************************
```

```
section .text
global main
main:
push rbp
mov rbp,rsp
; -----
; Цикл для знаходження суми.
mov rcx, [length]
mov rbx, fltLst
mov rsi, 0
movsd xmm1, qword [lstSum]
sumLp:
movsd xmm0, qword [rbx+rsi*8] ; отримання fltLst[i]
addsd xmm1, xmm0 ; корегування сум
inc rsi ; i++
loop sumLp
movsd qword [lstSum], xmm1 ; збереження суми
; -----
movsd xmm0,[lstSum]
mov rdi,fmtflt1
mov rax,1  ; кількість чисел з плаваючою крапкою
call printf
; -----
; Обчислення середнього для списку чисел
cvtsi2sd xmm1, dword [length]
movsd xmm0, [lstSum]
divsd xmm0, xmm1
movsd qword [lstAve], xmm0
; -----
mov rdi,fmtflt2
mov rax,1
call printf
; -----
; Завершення програми
last:
mov rax, SYS_exit
mov rbx, EXIT_SUCCESS ; вихід з програми
syscall

$ ~/asm_ld.sh 1.asm
$ ./1
Сума 243.500000
Середнє 16.233333
```

2. Обчислення абсолютного значення чисел із плаваючою крапкою

```
; 2.asm - обчислення абсолютного значення чисел з плаваючою крапкою
extern printf
section .data
; -----
; Визначення констант
TRUE equ 1
FALSE equ 0
EXIT_SUCCESS equ 0 ; успішна операція
SYS_exit equ 60 ; код завершення для системного виклику
; -----
```

```
; Визначення деяких змінних
dZero   dq 0.0
dNegOne dq -1.0
fltVal  dq -8.25
fmtflt  db "val=%lf abs(val)=%lf",10,0
; *****************************************************
section .text
global main
main:
push rbp
mov rbp,rsp
; -----
; функція абсолютного значення числа з плаваючою крапкою fltVal
movsd xmm0, [fltVal]
movsd xmm1, [dZero]
ucomisd xmm0,xmm1
jae isPos
movsd xmm1, qword [dNegOne]
mulsd xmm0,xmm1
movsd qword [absVal], xmm0
isPos:
; -----
movsd xmm0, [fltVal]
movsd xmm1, [absVal]
mov rdi,fmtflt
mov rax,2 ; кількість чисел з плаваючою крапкою
call printf
; -----
; Завершення програми
last:
mov rax, SYS_exit
mov rbx, EXIT_SUCCESS ; вихід з програми
syscall

$ ~/asm_ld.sh 2.asm
$ ./2
val=-8.250000 abs(val)=8.250000
```

3. Арифметичні обчислення над числами із плаваючою крапкою

```
; fcalc.asm
extern printf
section .data
number1 dq 9.0
number2 dq 73.0
fmt db "Є числа %f і %f",10,0
fmtfloat db "%s %f",10,0
f_sum db "float сума %f і %f є %f",10,0
f_dif db "float різниця %f і %f є %f",10,0
f_mul db "float добуток %f і %f є %f",10,0
f_div db "float ділення %f на %f є %f",10,0
f_sqrt db "float корінь квадратний з %f є %f",10,0
section .bss
section .text
global main
main:
```

```
push rbp
mov rbp,rsp
; print the numbers
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rdi,fmt
mov rax,2 ; two floats
call printf
; sum
movsd xmm2, [number1] ; double precision float into xmm
addsd xmm2, [number2] ; add doube precision to xmm
; print the result
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rdi,f_sum
mov rax,3 ; three floats
call printf
; difference
movsd xmm2, [number1] ; double precision float into xmm
subsd xmm2, [number2] ; subtract from xmm
; print the result
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rdi,f_dif
mov rax,3 ; three floats
call printf
; multiplication
movsd xmm2, [number1] ; double precision float into xmm
mulsd xmm2, [number2] ; multiply with xmm
; print the result
mov rdi,f_mul
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rax,3 ; three floats
call printf
; division
movsd xmm2, [number1] ; double precision float into xmm
divsd xmm2, [number2] ; divide xmm0
; print the result
mov rdi,f_div
movsd xmm0, [number1]
movsd xmm1, [number2]
mov rax,1 ; one float
call printf
; squareroot
sqrtsd xmm1, [number1] ; squareroot double precision in xmm
; print the result
mov rdi,f_sqrt
movsd xmm0, [number1]
mov rax,2 ; two floats
call printf
; exit
mov rsp, rbp
pop rbp ; undo the push at the beginning
ret
```

```
$ ~/asm_ld.sh 3.asm
$ ./3
Є числа 9.000000 і 73.000000
float сума 9.000000 і 73.000000 є 82.000000
float різниця 9.000000 і 73.000000 є -64.000000
float добуток 9.000000 і 73.000000 є 657.000000
float ділення 9.000000 на 73.000000 є 0.123288
float корінь квадратний з 9.000000 є 3.000000
```

4. Арифметичні обчислення над числами із плаваючою крапкою з використанням команд x87 FPU і xmm.

```
extern printf
section .data
a dq 1.2
b dq 3.4
c dq 5.6
d dq 7.8
res dq 0.0
double_format db "a=%lf b=%lf", 10, 0
resfmt db "Результат res=%lf",10,0
section .text
global main
main:
push rbp
mov rbp,rsp

mov rdi, double_format
movsd xmm0, [a]
movsd xmm1, [b]
mov rax,2
call printf

mov rdi, double_format
movsd xmm0,[c]
movsd xmm1,[d]
mov rax,2
call printf

finit
fld  qword [a]      ; a   -> st(0) TOP=7
fmul qword [b]      ; a*b -> st(1) YOP=7
fld  qword [c]      ; c   -> st(0) TOP=6
fmul qword [d]      ; c*d -> st(0) TOP=6
fadd                ; a*b + c*d -> st(0) TOP=7
fstp qword [res]    ; st(0) -> res st(0) TOP=0

mov rdi, resfmt
movsd xmm0, [res]
mov rax,1
call printf

mov rax, 60
xor rbx, 0
```

```
syscall

$ ~/asm_ld.sh 4.asm
$ ./5
a=1.200000 b=3.400000
a=5.600000 b=7.800000
Результат res=47.760000
```

5. Арифметичні обчислення над числами із плаваючою крапкою з використанням команд x87 FPU.

```
;          1         2         3         4         5         6         7
;01234567890123456789012345678901234567890123456789012345678901234567890
;=============================================================================
;+-----------------------------------------------------------------------+
;|                                                                       |
;|    fpu.asm - приклади використання FPU регістрів                      |
;|       для обчислень з плаваючою крапкою                               |
;|                                                                       |
;|    Є 11 прикладів у початковому коді:                                 |
;|    1: Пересилання значення single точності у FPU стек                 |
;|    2: Пересилання значення double точності у FPU стек                 |
;|    3: Пересилання значення extended точності у FPU стек               |
;|    4: Операція додавання (add) (single + single)                      |
;|    5: Операція додавання (add) (single + single + single)             |
;|    6: Операція додавання (add) (single + double + extended)           |
;|    7: Операція віднімання (sub)(single - single)                      |
;|    8: Операція множення (mul)  (single * single)                      |
;|    9: Операція ділення (div)   (single / single)                      |
;|    10: Перетворення single у double                                   |
;|    11: Порівняння чисел з плаваючою крапкою                            |
;=============================================================================
;---- секція неініціалізованих даних----------------------------------
section .bss noexec write align=8

single_sum:       resd 1
single_sub:       resd 1
single_mul:       resd 1
single_div:       resd 1
single2double:    resq 1
double_sum:       resq 1

;---- секція читання/записування даних--------------------------------
section .data noexec write align=8

;---- секція даних тільки для читання --------------------------------
section .rodata noexec nowrite align=8

single_value1:    dd 12.34                    ; 32-біти
single_value2:    dd 102.35
single_value3:    dd -52.02

double_value1:    dq 12.34                    ; 64-біти
double_value2:    dq 102.35
double_value3:    dq -52.02
```

```
extended_value1:    dt 12.34                    ; 100-біт
extended_value2:    dt 102.35
extended_value3:    dt -52.02

string_1_begin:
    db "example_11: "
    db "single_value1 > single_value2"
    db 0aH                                      ;newline символ
    db 00H                                      ;null символ
string_1_end:

string_2_begin:
    db "example_11: "
    db "single_value1 < single_value2"
    db 0aH
    db 00H
string_2_end:

string_3_begin:
    db "example_11: "
    db "single_value1 == single_value2"
    db 0aH                                      ;newline character
    db 00H                                      ;null str terminator
string_3_end:

string_4_begin:
    db "example_11 ERROR! "
    db "st0 і джерело невизначені!"
    db 0aH
    db 00H
string_4_end:

;---- секцію команд -------------------------------------------------
section .text exec nowrite align=32

global _start:function
_start:

.example_1:
;+--------------------------------------------------------------------+
;| Пересилання single значення у FPU стек                             |
;|                                                                    |
;| В цьому прикладі виконуються 3 команди:                            |
;|   Крок 1: завантажити single_value1 у fpu стек                     |
;|   Крок 2: завантажити single_value2 у fpu стек                     |
;|   Крок 3: завантажити single_value3 у fpu стек                     |
;|                                                                    |
;| Нижче показано стан стеку під час виконання 3-х кроків:            |
;|  На кроці 1: st0 = single_value1                                   |
;|              st1 = 0                                               |
;|              st2 = 0                                               |
;|              st4 = 0                                               |
;|              st5 = 0                                               |
;|              st6 = 0                                               |
```

```
;|              st7 = 0                                                  |
;|                                                                       |
;|  На кроці 2: st0 = single_value2                                      |
;|              st1 = single_value1                                      |
;|              st2 = 0                                                  |
;|              st4 = 0                                                  |
;|              st5 = 0                                                  |
;|              st6 = 0                                                  |
;|              st7 = 0                                                  |
;|                                                                       |
;|  На кроці 3: st0 = single_value3                                      |
;|              st1 = single_value2                                      |
;|              st2 = single_value1                                      |
;|              st4 = 0                                                  |
;|              st5 = 0                                                  |
;|              st6 = 0                                                  |
;|              st7 = 0                                                  |
;|                                                                       |
;| Зауваження: Перед використання fpu стеку, скинути його регістри      |
;|             командою finit у значення за замовчуванням.               |
;+-----------------------------------------------------------------------+

    finit                         ;скинути fpu регістри
    fld    dword [single_value1] ;завантажити  12.34 у fpu стек
    fld    dword [single_value2] ;завантажити 102.35 у fpu стек
    fld    dword [single_value3] ;завантажити -52.02 у fpu стек

.example_2:
;+-----------------------------------------------------------------------+
;| Пересилання double значень у FPU стек.                                |
;|                                                                       |
;| В цьому прикладі виконуються 3 команди:                               |
;|   Крок 1: завантажити double_value1 у fpu стек                        |
;|   Крок 2: завантажити double_value2 у fpu стек                        |
;|   Крок 3: завантажити double_value3 у fpu стек                        |
;|                                                                       |
;| Нижче показано стан стеку під час виконання 3-х кроків:               |
;|  На кроці 1: st0 = double_value1                                      |
;|              st1 = 0                                                  |
;|              st2 = 0                                                  |
;|              st4 = 0                                                  |
;|              st5 = 0                                                  |
;|              st6 = 0                                                  |
;|              st7 = 0                                                  |
;|                                                                       |
;|  На кроці 2: st0 = double_value2                                      |
;|              st1 = double_value1                                      |
;|              st2 = 0                                                  |
;|              st4 = 0                                                  |
;|              st5 = 0                                                  |
;|              st6 = 0                                                  |
;|              st7 = 0                                                  |
;|                                                                       |
;|  На кроці 3: st0 = double_value3                                      |
;|              st1 = double_value2                                      |
```

```
;|              st2 = double_value1                                        |
;|              st4 = 0                                                    |
;|              st5 = 0                                                    |
;|              st6 = 0                                                    |
;|              st7 = 0                                                    |
;+-------------------------------------------------------------------------+

    finit                         ;скинути fpu регістри
    fld    qword [double_value1] ;завантажити  12.34 у fpu стек
    fld    qword [double_value2] ;завантажити 102.35 у fpu стек
    fld    qword [double_value3] ;завантажити -52.02 у fpu стек

.example_3:
;+-------------------------------------------------------------------------+
;| Пересилання extended значення FPU стек.                                 |
;|                                                                         |
;| В цьому прикладі виконуються 3 команди:                                 |
;|   Крок 1: завантажити extended_value1 у fpu стек                        |
;|   Крок 2: завантажити extended_value2 у fpu стек                        |
;|   Крок 3: завантажити extended_value3 у fpu стек                        |
;|                                                                         |
;| Below are the observations on FPU stacks during these 3 steps:          |
;|   AT STEP 1: st0 = extended_value1                                      |
;|              st1 = 0                                                    |
;|              st2 = 0                                                    |
;|              st4 = 0                                                    |
;|              st5 = 0                                                    |
;|              st6 = 0                                                    |
;|              st7 = 0                                                    |
;|                                                                         |
;|   AT STEP 2: st0 = extended_value2                                      |
;|              st1 = extended_value1                                      |
;|              st2 = 0                                                    |
;|              st4 = 0                                                    |
;|              st5 = 0                                                    |
;|              st6 = 0                                                    |
;|              st7 = 0                                                    |
;|                                                                         |
;|   AT STEP 3: st0 = extended_value3                                      |
;|              st1 = extended_value2                                      |
;|              st2 = extended_value1                                      |
;|              st4 = 0                                                    |
;|              st5 = 0                                                    |
;|              st6 = 0                                                    |
;|              st7 = 0                                                    |
;+-------------------------------------------------------------------------+

    finit                           ;скинути fpu регістри
    fld    tword [extended_value1] ;завантажити  12.34 у fpu стек
    fld    tword [extended_value2] ;завантажити 102.35 у fpu стек
    fld    tword [extended_value3] ;завантажити -52.02 у fpu стек

.example_4:
;+-------------------------------------------------------------------------+
;| Обчислення суми: single_sum = single1 + single2                         |
```

```
;|                                                                    |
;|                                                                    |
;|   Стан стеку перед і після виконання команди FAdd:                |
;|   Перед fadd:  st0 = 102.34999847412109375                         |
;|                st1 = 12.340000152587890625                         |
;|                                                                    |
;|   Після fadd:  st0 = 114.689998626708984375                        |
;|                st1 = 0                                             |
;+--------------------------------------------------------------------+

    finit                       ;скинути fpu регістри
    fld    dword [single_value1] ;завантажити single_value1 у fpu стек
    fld    dword [single_value2] ;завантажити single_value2 у fpu стек
    fadd                        ;st0 := st1 + st0
    fstp   dword [single_sum]    ;записати результат суми у пам'ять

.example_5:
;+--------------------------------------------------------------------+
;| Add Operation single_sum = single1 + single2 + single3             |
;+--------------------------------------------------------------------+

    finit                       ;скинути fpu регістри
    fld    dword [single_value1] ;завантажити single_value1 у fpu стек
    fld    dword [single_value2] ;завантажити single_value2 у fpu стек
    fadd                        ;st0 := st1 + st0
    fld    dword [single_value3] ;завантажити single_value3 у fpu стек
    fadd                        ;st0 := st1 + st0
    fstp   dword [single_sum]    ;записати результат суми у пам'ять

.example_6:
;+--------------------------------------------------------------------+
;| Обчислення суми double_sum = single1 + double2 + extended3         |
;+--------------------------------------------------------------------+

    finit                         ;скинути fpu регістри
    fld    dword [single_value1]   ;завантажити single_value1 у fpu стек
    fld    qword [double_value2]   ;завантажити double_value2 у fpu стек
    fadd                          ;st0 := st1 + st0
    fld    tword [extended_value3] ;завантажити extended_value3 у fpu стек
    fadd                          ;st0 := st1 + st0
    fstp   qword [double_sum]      ;записати результат суми у пам'ять

.example_7:
;+--------------------------------------------------------------------+
;| Обчислення різниці single_sum = single1 - single2                  |
;+--------------------------------------------------------------------+

    finit                       ;скинути fpu регістри
    fld    dword [single_value1] ;завантажити single_value1 у fpu стек(st1)
    fld    dword [single_value2] ;завантажити single_value2 у fpu стек(st0)
    fsub                        ;st0 := st1 - st0
    fstp   dword [single_sub]    ;записати результат різниці у пам'ять

.example_8:
;+--------------------------------------------------------------------+
```

```
;| Обчислення добутку  single_mul = single1 * single2                    |
;+-----------------------------------------------------------------------+

    finit                          ;скинути fpu регістри
    fld    dword [single_value1]   ;завантажити single_value1 у fpu стек(st1)
    fld    dword [single_value2]   ;завантажити single_value2 у fpu стек(st0)
    fmul                           ;st0 := st1 * st0
    fstp   dword [single_mul]      ;записати результат добутку у пам'ять

.example_9:
;+-----------------------------------------------------------------------+
;| Обчислення результату ділення single_div = single1 / single2          |
;+-----------------------------------------------------------------------+

    finit                          ;скинути fpu регістри
    fld    dword [single_value1]   ;завантажити single_value1 у fpu стек (st1)
    fld    dword [single_value2]   ;завантажити single_value2 у fpu стек (st0)
    fdiv                           ;st0 := st1 / st0
    fstp   dword [single_div]      ;записати результат ділення у пам'ять

.example_10:
;+-----------------------------------------------------------------------+
;| Перетворення single у double                                          |
;|                                                                       |
;| Перетворення single у double виконується за два кроки:                |
;|   Крок 1: Завантажити single_value1 у FPU стек                        |
;|   Крок 2: Записати  double значення з single_value1 у пам'ять         |
;+-----------------------------------------------------------------------+

    finit                          ;скинути fpu регістри
    fld    dword [single_value1]   ;завантажити single_value1 у fpu стек (st0)
    fstp   qword [single2double]   ;записати double значення у пам'ять

.example_11:
;+-----------------------------------------------------------------------+
;| Порівняння float чисел (знаходження більшого)                         |
;|                                                                       |
;| Цей приклад показує як порівняти два floating-point числа.            |
;|                                                                       |
;| Задано single_value1 і single_value2,                                 |
;| Якщо single_value1 > single_value2 тоді                               |
;|     друк "single_value1 > single_value2"                              |
;| інакше, якщо single_value1 < single_value2 тоді                       |
;|     друк "single_value1 < single_value2"                              |
;| інакше                                                                |
;|     друк "single_value1 == single_value2"                             |
;|                                                                       |
;| Нижче показано                                                        |
;| 16-бітовий регістр FPU Status Word (зберігає загальний стан FPU):     |
;|                                                                       |
;|            +-----+                                                    |
;|     Bit 0  | IE  | --> Invalid operation exception                     |
;|            +-----+                                                    |
;|     Bit 1  | DE  | --> Denormalized exception                          |
;|            +-----+                                                    |
```

```
;|     Bit 2  | ZE  | --> Zero divide exception                              |
;|            +-----+                                                        |
;|     Bit 3  | OE  | --> Overflow exception                                 |
;|            +-----+                                                        |
;|     Bit 4  | UE  | --> Underflow exception                                |
;|            +-----+                                                        |
;|     Bit 5  | PE  | --> Precision exception                                |
;|            +-----+                                                        |
;|     Bit 6  | SF  | --> Stack Fault exception                              |
;|            +-----+                                                        |
;|     Bit 7  | IR  | --> Interupt Request                                   |
;|            +-----+                                                        |
;|     Bit 8  | C0  | --> CF (carry flag) [CONDITION CODE]                   |
;|            +-----+                                                        |
;|     Bit 9  | C1  | --> =1 (stack overflow), =0 (stack underflow)          |
;|            +-----+                                                        |
;|     Bit 10 | C2  | --> PF (parity flag) [CONDITION CODE]                  |
;|            +-----+                                                        |
;|     Bit 11 |     |                                                        |
;|     Bit 12 | TOP | --> Top of stack pointer                               |
;|     Bit 13 |     |                                                        |
;|            +-----+                                                        |
;|     Bit 14 | C3  | --> ZF (zero flag) [CONDITION CODE]                    |
;|            +-----+                                                        |
;|     Bit 15 |  B  | --> B=1 (busy), B=0 (idle)                             |
;|            +-----+                                                        |
;|                                                                           |
;| У цій таблиці показано біти коду умов FPU:                                |
;| +-------------+-------------------+-------------------------------+ |
;| |             |Condition Code bits|                               | |
;| | Instruction +----+----+----+----+          Condition            | |
;| |             | C3 | C2 | C1 | C0 |                               | |
;| +-------------+----+----+----+----+-------------------------------+ |
;| | fcom,       | 0  | 0  |    | 0  | st0 > source                  | |
;| | fcomp,      +----+----+----+----+-------------------------------+ |
;| | fcompp,     | 0  | 0  |    | 1  | st0 < source                  | |
;| | ficom,      +----+----+----+----+-------------------------------+ |
;| | ficomp      | 1  | 0  |    | 0  | st0 = source                  | |
;| |             +----+----+----+----+-------------------------------+ |
;| |             | 1  | 1  |    | 1  | st0 or source are undefined   | |
;| +-------------+----+----+----+----+-------------------------------+ |
;+---------------------------------------------------------------------+
```

```
    finit                           ;скинути fpu регістри у замовчування
    fld    dword [single_value2]    ;завантажити single_value2 у fpu стек(st1)
    fld    dword [single_value1]    ;завантажити single_value1 у fpu стек(st0)
    fcom   st0, st1                 ;порівняти st0 з st1
    fstsw  ax                       ;ax := fpu регістр стану

    and    eax, 0100011100000000B ;залишити тільки біти коду умов
    cmp    eax, 0000000000000000B ;чи st0 > source ?
    je     .example_11_greater
    cmp    eax, 0000000100000000B ;чи st0 < source ?
    je     .example_11_less
    cmp    eax, 0100000000000000B ;чи st0 = source ?
```

```
    je      .example_11_equal
    jmp     .example_11_error       ;інакше, st0 або source невизначені

.example_11_greater:
    mov     rsi, string_1_begin                   ;ecx := addr string_1
    mov     rdx, (string_1_end - string_1_begin) ;edx := length string_1
    jmp     .example_11_write

.example_11_less:
    mov     rsi, string_2_begin                   ;ecx := addr string_2
    mov     rdx, (string_2_end - string_2_begin) ;edx := length string_2
    jmp     .example_11_write

.example_11_equal:
    mov     rsi, string_3_begin                   ;ecx := addr string_3
    mov     rdx, (string_3_end - string_3_begin) ;edx := length string_3
    jmp     .example_11_write

.example_11_error:
    mov     rsi, string_4_begin                   ;ecx := addr string_4
    mov     rdx, (string_4_end - string_4_begin) ;edx := length string_4

.example_11_write:
        mov rax,1          ; системний виклик write
        mov rdi,1          ; виведення у консоль
        syscall            ; звернення до ядра


;+------------------------------------------------------------------------+
;| Вихід з програми!                                                      |
;+------------------------------------------------------------------------+
_exit:
        mov rax, 60 ; системний виклик (exit, вихід з програми)
        mov rbx,  0 ; код повернення в ОС
        syscall     ; звернення до ядра

$ ~/asm_ld.sh fpu.asm
$ ./fpu.asm
example_11: single_value1 < single_value2
```

6. Завантаження програми в gdb:

```
$ gdb fpu
```

7. Друк програми в gdb:

```
(gdb)list 1,600
1   1         2         3         4         5         6
2  ;01234567890123456789012345678901234567890123456789012345678901234567890123456789
3  ;============================================================
4  ;+----------------------------------------------------------+
5  ;|                                                          |
6  ;|    Приклади використання FPU регістрів для обчислень     |
7  ;|      з плаваючою крапкою|                                |
8  ;+----------------------------------------------------------+
...
437     ;+------------------------------------------------------+
```

```
438     ;| Вихід з програми!                                          |
439     ;+-----------------------------------------------------------+
440     ;.exit:
441     _exit:
442             mov rax, 60 ; системний виклик (exit, вихід з програми)
443             mov rbx,  0 ; код повернення в ОС
444             syscall     ; звернення до ядра
```

8. Задання точок зупинки:

```
(gdb) b 157
Breakpoint 1 at 0x401018: file fpu.asm, line 193.
(gdb) b 197
Breakpoint 2 at 0x401030: file fpu.asm, line 233.
(gdb) b 237
Breakpoint 3 at 0x401048: file fpu.asm, line 251.
(gdb) b 256
Breakpoint 4 at 0x401062: file fpu.asm, line 262.
(gdb) b 269
Breakpoint 5 at 0x401085: file fpu.asm, line 275.
(gdb) b 282
Breakpoint 6 at 0x4010a8: file fpu.asm, line 288.
(gdb) b 293
Breakpoint 7 at 0x4010c2: file fpu.asm, line 299.
(gdb) b 304
Breakpoint 8 at 0x4010dc: file fpu.asm, line 310.
(gdb) b 315
Breakpoint 9 at 0x4010f6: file fpu.asm, line 325.
(gdb) b 328
Breakpoint 10 at 0x401107: file fpu.asm, line 395.
(gdb) b 436
Breakpoint 11 at 0x401185: file fpu.asm, line 441.
```

9. Інформація про точки зупинок

```
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000401018 fpu.asm:193
2       breakpoint     keep y   0x0000000000401030 fpu.asm:233
3       breakpoint     keep y   0x0000000000401048 fpu.asm:251
4       breakpoint     keep y   0x0000000000401062 fpu.asm:262
5       breakpoint     keep y   0x0000000000401085 fpu.asm:275
6       breakpoint     keep y   0x00000000004010a8 fpu.asm:288
7       breakpoint     keep y   0x00000000004010c2 fpu.asm:299
8       breakpoint     keep y   0x00000000004010dc fpu.asm:310
9       breakpoint     keep y   0x00000000004010f6 fpu.asm:325
10      breakpoint     keep y   0x0000000000401107 fpu.asm:395
11      breakpoint     keep y   0x0000000000401185 fpu.asm:441
```

10. Покрокове виконання програми

```
(gdb)run
Breakpoint 1, _start.example_2 () at fpu.asm:193
193         finit                       ;скинути fpu регістри
(gdb) next
194         fld    qword [double_value1] ;завантажити  12.34 у fpu стек
(gdb) next
195         fld    qword [double_value2] ;завантажити 102.35 у fpu стек
```

```
(gdb) next
196         fld    qword [double_value3] ;завантажити -52.02 у fpu стек
(gdb) next
```

9. Виведення значень комірок з пам'яті

```
(gdb) x/f &single_value1
0x402000 <single_value1>:       12.3400002
(gdb) x/f &double_value2
0x402014 <double_value2>:       2.72008302e+23
```

11. Покрокове виконання програми

```
(gdb) n
234         fld    tword [extended_value1] ;завантажити  12.34 у fpu стек
(gdb) n
235         fld    tword [extended_value2] ;завантажити 102.35 у fpu стек
...

(gdb) n
255         fstp   dword [single_sum]    ;записати результат суми у пам'ять
(gdb) x/wf &single_sum
0x403108 <single_sum>:  114.690002
...
(gdb) x/f &single_div
0x403114 <single_div>:  0.120566688
...
```

12. Вилучення всіх точок зупинок

```
(gdb) clear
```

13. Виконання програми

```
(gbd) run
Starting program: /home/user/asm/fpu
example_11: single_value1 < single_value2
[Inferior 1 (process 403) exited with code 01]
(gdb) disassemble _start
```

14. Деасемблювання програми починаючи з точку входу _start і до точки виходу _exit:

```
(gdb) disassemble _start, _exit
Dump of assembler code from 0x401000 to 0x401185:
   0x0000000000401000 <_start+0>:       finit
   0x0000000000401003 <_start+3>:       flds   0x402000
   0x000000000040100a <_start+10>:      flds   0x402004
   0x0000000000401011 <_start+17>:      flds   0x402008
   0x0000000000401018 <_start.example_2+0>:     finit
   0x000000000040101b <_start.example_2+3>:     fldl   0x40200c
   0x0000000000401022 <_start.example_2+10>:    fldl   0x402014
   0x0000000000401029 <_start.example_2+17>:    fldl   0x40201c
   0x0000000000401030 <_start.example_3+0>:     finit
   0x0000000000401033 <_start.example_3+3>:     fldt   0x402024
   0x000000000040103a <_start.example_3+10>:    fldt   0x40202e
   0x0000000000401041 <_start.example_3+17>:    fldt   0x402038
   0x0000000000401048 <_start.example_4+0>:     finit
   0x000000000040104b <_start.example_4+3>:     flds   0x402000
   0x0000000000401052 <_start.example_4+10>:    flds   0x402004
   0x0000000000401059 <_start.example_4+17>:    faddp  %st,%st(1)
```

```
   0x000000000040105b <_start.example_4+19>:    fstps  0x403108
   0x0000000000401062 <_start.example_5+0>:     finit
   0x0000000000401065 <_start.example_5+3>:     flds   0x402000
   0x000000000040106c <_start.example_5+10>:    flds   0x402004
   0x0000000000401073 <_start.example_5+17>:    faddp  %st,%st(1)
   0x0000000000401075 <_start.example_5+19>:    flds   0x402008
   0x000000000040107c <_start.example_5+26>:    faddp  %st,%st(1)
   0x000000000040107e <_start.example_5+28>:    fstps  0x403108
   0x0000000000401085 <_start.example_6+0>:     finit
   0x0000000000401088 <_start.example_6+3>:     flds   0x402000
   0x000000000040108f <_start.example_6+10>:    fldl   0x402014
   0x0000000000401096 <_start.example_6+17>:    faddp  %st,%st(1)
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000000000401098 <_start.example_6+19>:    fldt   0x402038
   0x000000000040109f <_start.example_6+26>:    faddp  %st,%st(1)
   0x00000000004010a1 <_start.example_6+28>:    fstpl  0x403120
   0x00000000004010a8 <_start.example_7+0>:     finit
   0x00000000004010ab <_start.example_7+3>:     flds   0x402000
   0x00000000004010b2 <_start.example_7+10>:    flds   0x402004
   0x00000000004010b9 <_start.example_7+17>:    fsubrp %st,%st(1)
   0x00000000004010bb <_start.example_7+19>:    fstps  0x40310c
   0x00000000004010c2 <_start.example_8+0>:     finit
   0x00000000004010c5 <_start.example_8+3>:     flds   0x402000
   0x00000000004010cc <_start.example_8+10>:    flds   0x402004
   0x00000000004010d3 <_start.example_8+17>:    fmulp  %st,%st(1)
   0x00000000004010d5 <_start.example_8+19>:    fstps  0x403110
   0x00000000004010dc <_start.example_9+0>:     finit
   0x00000000004010df <_start.example_9+3>:     flds   0x402000
   0x00000000004010e6 <_start.example_9+10>:    flds   0x402004
   0x00000000004010ed <_start.example_9+17>:    fdivrp %st,%st(1)
   0x00000000004010ef <_start.example_9+19>:    fstps  0x403114
   0x00000000004010f6 <_start.example_10+0>:    finit
   0x00000000004010f9 <_start.example_10+3>:    flds   0x402000
   0x0000000000401100 <_start.example_10+10>:   fstpl  0x403118
   0x0000000000401107 <_start.example_11+0>:    finit
   0x000000000040110a <_start.example_11+3>:    flds   0x402004
   0x0000000000401111 <_start.example_11+10>:   flds   0x402000
   0x0000000000401118 <_start.example_11+17>:   fcom   %st(1)
   0x000000000040111a <_start.example_11+19>:   fstsw  %ax
   0x000000000040111d <_start.example_11+22>:   and    $0x4700,%eax
   0x0000000000401122 <_start.example_11+27>:   cmp    $0x0,%eax
      0x0000000000401125  <_start.example_11+30>:       je           0x401137
<_start.example_11_greater>
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000000000401127 <_start.example_11+32>:   cmp    $0x100,%eax
      0x000000000040112c  <_start.example_11+37>:       je           0x401148
<_start.example_11_less>
   0x000000000040112e <_start.example_11+39>:   cmp    $0x4000,%eax
      0x0000000000401133  <_start.example_11+44>:       je           0x401159
<_start.example_11_equal>
       0x0000000000401135  <_start.example_11+46>:       jmp          0x40116a
<_start.example_11_error>
   0x0000000000401137 <_start.example_11_greater+0>:    movabs $0x402042,%rsi
   0x0000000000401141 <_start.example_11_greater+10>:   mov    $0x2b,%edx
```

```
         0x0000000000401146 <_start.example_11_greater+15>:     jmp       0x401179
<_start.example_11_write>
        0x0000000000401148 <_start.example_11_less+0>:       movabs $0x40206d,%rsi
        0x0000000000401152 <_start.example_11_less+10>:      mov    $0x2b,%edx
          0x0000000000401157 <_start.example_11_less+15>:          jmp       0x401179
<_start.example_11_write>
        0x0000000000401159 <_start.example_11_equal+0>:      movabs $0x402098,%rsi
        0x0000000000401163 <_start.example_11_equal+10>:     mov    $0x2c,%edx
          0x0000000000401168 <_start.example_11_equal+15>:         jmp       0x401179
<_start.example_11_write>
        0x000000000040116a <_start.example_11_error+0>:      movabs $0x4020c4,%rsi
        0x0000000000401174 <_start.example_11_error+10>:     mov    $0x41,%edx
        0x0000000000401179 <_start.example_11_write+0>:      mov    $0x1,%eax
        0x000000000040117e <_start.example_11_write+5>:      mov    $0x1,%edi
        0x0000000000401183 <_start.example_11_write+10>:     syscall
    End of assembler dump.
```

15. Друк значень комірок пам'яті за адресами:

```
(gdb) x/f 0x402000
0x402000 <single_value1>:       12.3400002
(gdb) x/f &single_value1
0x402000 <single_value1>:       12.3400002
(gdb) x/fg &double_value2
0x402014 <double_value2>:       102.34999999999999
```

СПИСОК ЛІТЕРАТУРИ

Основний

1. Зілінський Ю. В., Перекрест А. Л., Юдіна А. Л. Системне програмування. Програмування на асеблері: навч. Посібник. Кременчук: Кременчуцький національний університет імені Михайла Остроградського, 2023. 258 с.
2. Системне програмування. Програмування на асемблері: комп'ютерний практикум [Електронний ресурс]: навч. посіб. для студ. освітньої програми «Комп'ютерні системи та мережі» спеціальності 123 «Комп'ютерна інженерія» / КПІ ім. Ігоря Сікорського; уклад. Порєв В.М. – Електронні текстові дані (1 файл: 3,2 МБайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 146 с. – Назва з екрана.
3. Методичні вказівки для виконання лабораторних робіт з дисципліни «Системне програмування» для студентів денної та заочної форми навчання розроблені у відповідності з навчальним планом спеціальності 123 «Комп’ютерна інженерія» / Уклад. Паламар А.М., Паламар М.І. – Тернопіль: ТНТУ, 2020. – 70 с.
4. Методичні вказівки до виконання лабораторних робіт з дисципліни "Системне програмування" для студентів спеціальності 125 "Кібербезпека" всіх форм навчання – Частина 1 / Укл.: В.В. Шкарупило. – Київ: НУБіП, 2022. – 42 с.
5. Основи комп’ютерної техніки та програмування мікропроцесорів: навч. посіб. / Д.О. Гололобов. – К. : Редакційно-видавничий центр Державного університету телекомунікацій, 2019. – 58с. : іл
6. Рисований О. М. Системне програмування: підручник для студентів напрямку «Комп’ютерна інженерія» вищих навчальних закладів у 2-х томах. Том 1. Видання четверте: виправлено та доповнено – Харків: «Слово», 2015. – 576 с.
7. Рисований О. М. Системне програмування: підручник для студентів напрямку «Комп’ютерна інженерія» вищих навчальних закладів у 2-х томах. Том 2. Видання четверте: виправлено та доповнено – Харків: «Слово», 2015. – 378 с.

Додатковий

8. Randall Hyde. The Art Of 64-bit Assembly. No Starch Press, 2021. – 1032 p.
9. Sherwyn Allibang. Assembly Language: Simple, Short, and Straightforward Way of Learning Assembly Programming. Independently published, 2020. – 160 p.
10. Irvine Kip. Assembly Language for x86 Processors 8th ed. Pearson, 2019. – 880 p.
11. Jo Van Hoey. Beginning x64 Assembly Programming, From Novice to AVX Professional. Apress, 2019. – 432 p.
12. Daniel Kusswurm. Modern X86 Assembly Language Programming: Covers x86 64-bit, AVX, AVX2 and AVX-512 1st ed. Apress, 2018. – 625 p.
13. Ray Seyfarth. Introduction to 64 Bit Windows Assembly Language Programming 8th ed. CreateSpace Independent Publishing Platform, 2017. – 288 p.
14. Alexey Lyashko. Follow Mastering Assembly Programming: From instruction set to kernel module with Intel processor 1st ed. Packt Publishing, 2017. – 290 p.
15. Jeff Duntemann. Follow Assembly Language Step-by-Step 3rd ed. Wiley, 2011. – 656 p.

Інформаційні ресурси

16. GDB: The GNU Project Debugger [електронний ресурс]: – Режим доступу: https://www.gnu.org/software/gdb/ – назва з екрану
17. DDD (data display debugger) [електронний ресурс]: – Режим доступу: https://www.gnu.org/software/ddd/ – назва з екрану
18. KGDB A Graphical Debugger Interface [електронний ресурс]: – Режим доступу: https://www.kdbg.org/ – назва з екрану
19. SASM – data display debugger [електронний ресурс]: – Режим доступу: https://www.gnu.org/software/ddd/manual/ – назва з екрану.
20. NASM [Електронний ресурс]. – Режим доступу: https://www.nasm.us. – Заголовок з екрану