Лабораторна робота №1. Трансляція мов високого рівня у мови низького рівня. Частина 1

Мета лабораторної роботи наступна:

- ознайомитись із роботою сучасних трансляторів на прикладі трансляції невеликої програми, що містить різні конструкції керування потоком виконання і написаної мовою програмування високого рівня, у код цієї програми мовою низького рівня;
- навчитись виокремлювати синтаксичні/семантичні конструкції програми, записаної мовою високого рівня, у відповідній їй програмі, записаній мовою низького рівня на прикладі мови високого рівня C.

1. Вимоги до апаратного і програмного забезпечення

Для виконання лабораторної роботи діють наступні вимоги щодо програмного та апаратного забезпечення:

1) доступ до мережі Інтернет, наявність сучасного браузера (Mozilla Firefox, Google Chrome або ін.);
2) компілятор мови C — gcc версії 14.2 або вище.

2. Загальне завдання та термін виконання

Завдання лабораторної роботи складається з трьох пунктів:

1. Реалізувати програму сортування масиву згідно із варіантом мовою C (інформація про варіанти наведена в п. 4). *Результатом* виконання цього пункту є лістинг програми мовою C.
2. Виконати трансляцію програми, написаної мовою C, в асемблерний код за допомогою **gcc** й встановити семантичну відповідність між командами мови C та командами одержаного асемблерного коду, додавши відповідні коментарі з поясненням. *Результатом* виконання даного пункту буде лістинг асемблерного коду програми із коментарями в коді, в яких наведено відповідний код програми, записаною мовою C (приклад. див. в п. 3.2).
3. Розібратись і вміти пояснити, що виконують ті чи інші команди мови асемблера, що будуть присутні в коді мовою асемблера, отриманого в другому пункті загального завдання; вміти пояснити зв’язок між кодом мовою C та кодом мовою асемблера.

В результаті виконання лабораторної роботи має бути підготовлено **звіт**, що складатиметься з наступних частин:

1) титульний аркуш;
2) загальне завдання лабораторної роботи;
3) варіант і завдання за варіантом до першого пункту загального завдання (інформація про варіанти наведена в п. 4);
4) лістинг програми мовою С;
5) лістинг асемблерного коду з позначеннями.

Граничний **термін виконання** лабораторної роботи визначається викладачем в інформаційному листі, що надсилається на пошту групи та/або в групу в телеграмі разом із даними методичними вказівками. В разі недотримання граничного терміну виконання лабораторної роботи, максимальна оцінка за роботу буде знижена на 1 бал за кожен тиждень протермінування. Кількість балів за дотримання дедлайну даної лабораторної роботи визначається силабусом (РСО). Ця ж кількість відповідає максимальній кількості балів, на яку може бути знижена оцінка за лабораторну роботу в разі недотримання дедлайну.

3. Методичні вказівки

В даному розділі наведено методичні вказівки щодо виконання пунктів завдання лабораторної роботи на спрощеному прикладі.

Нехай маємо наступне завдання: реалізувати програму (функцію) визначення суми парних чисел в масиві заданої довжини.

3.1 Реалізація завдання за варіантом мовою С

Реалізацією визначеного завдання мовою С буде наступний код:

```
#include<stdio.h>

int calculate_even_sum(int arr[], int size) {
  int sum = 0;
  // main loop
  for (int idx = 0; idx < size; idx++) {
    if ((arr[idx] & 1) == 0) {
      sum += arr[idx];
    }
  }

  return sum;
}

int main() {
  int arr[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  int arr_bytes_size = sizeof arr;
  int arr_size = arr_bytes_size > 0 ? arr_bytes_size / sizeof arr[0] : 0 ;
  int sum = calculate_even_sum(arr, arr_size);
  printf("Sum is: %d\n", sum);
  return 0;
}
```

Вимогою до реалізації завдання за варіантом є відсутність глобальних змінних, які виконують роль локальних параметрів. Також необхідно виконати завдання в одній функції. З метою декомпозиції коду, можна винести повторювані частини коду в окремі функції, проте лише у випадку, якщо кількість повторюваних рядків перевищує 3 команди (вирази/statement). Крім того код реалізації завдання не має містити коду, що використовується для налагодження програми (виклики printf і т. п.) та макросів.

Варто зауважити, що у звіт слід додати весь код, що реалізовує завдання за виключенням коду, що виконує виклик і передачу тестових даних. В прикладі вище у звіт необхідно додати лише реалізацію функції *calculate_even_sum*.

При реалізації програми мовою С слід дотримуватись конвенцій програмування мовою С. Наприклад, C Coding Style and Conventions.

3.2 Трансляція коду мовою С в код мовою асемблера

Трансляція файлу з програмою, записаною мовою С, у програму мовою асемблера відбувається за допомогою виклику наступною команди з командного рядка:

```
gcc -S -masm=intel <file_name.c> [-o <output_file_name.s>]
```

Опис параметрів:

- *<file_name.c>* - назва вхідного файлу;
- *-S* вказує GCC, що необхідно лише скомпілювати вхідний файл <file_name.c>, й повернути асемблерний код;
- *-masm=intel* вказує компілятору, що необхідно використовувати форму запису команд для процесорів Intel;
- *-o <output_file_name.s>* - опційно можна вказати файл, в який необхідно зберегти асемблерний код.

Якщо в коді мовою С будуть помилки — gcc видасть повідомлення про них. Після того, як всі помилки будуть виправлені, на виході буде отримано файл з розширенням .s (або іншим, якщо це буде явно вказано в параметрі *-o*). Для функції *calculate_even_sum* було згенеровано наступний код мовою асемблера:

```
      ...
calculate_even_sum:
.LFB0:
      .cfi_startproc
      push  rbp
      .cfi_def_cfa_offset 16
      .cfi_offset 6, -16
      mov   rbp, rsp
      .cfi_def_cfa_register 6
      mov   QWORD PTR [rbp-24], rdi
      mov   DWORD PTR [rbp-28], esi
      mov   DWORD PTR [rbp-4], 0
      mov   DWORD PTR [rbp-8], 0
      jmp   .L2
.L4:
      mov   eax, DWORD PTR [rbp-8]
      cdqe
      lea   rdx, [0+rax*4]
      mov   rax, QWORD PTR [rbp-24]
      add   rax, rdx
      mov   eax, DWORD PTR [rax]
      and   eax, 1
      test  eax, eax
      jne   .L3
      mov   eax, DWORD PTR [rbp-8]
      cdqe
      lea   rdx, [0+rax*4]
      mov   rax, QWORD PTR [rbp-24]
      add   rax, rdx
      mov   eax, DWORD PTR [rax]
      add   DWORD PTR [rbp-4], eax
.L3:
      add   DWORD PTR [rbp-8], 1
.L2:
      mov   eax, DWORD PTR [rbp-8]
      cmp   eax, DWORD PTR [rbp-28]
      jl    .L4
      mov   eax, DWORD PTR [rbp-4]
      pop   rbp
      .cfi_def_cfa 7, 8
      ret
      .cfi_endproc
```

Для того, щоб швидше розібратись з тим, що відбувається в асемблерному коді й визначити семантичні відповідники між мовою C та мовою асемблера, можна скористатись сервісом Compiler Explorer. Приклад його використання для функції *calculate_even_sum* наведено на рис. 1. Слід зазначити, що за замовчуванням мовою файлу з сирцями може бути обрано не C, а C++. Для обох мов є змога використовувати один і той самий пакет (наприклад, *x86_64 gcc 14.2*), проте програма, яка викликатиметься для трансляції відрізнятиметься: для C це gcc, а для C++ це g++. Асемблерний код, отриманий в результаті запуску gcc та g++, дещо відрізнятиметься (у чому можна переконатись виконавши трансляцію одного і того ж коду для обох мов, і порівнявши асемблерний код). Тому варто пересвідчитись, що мовою файлу з сирцями є саме C.

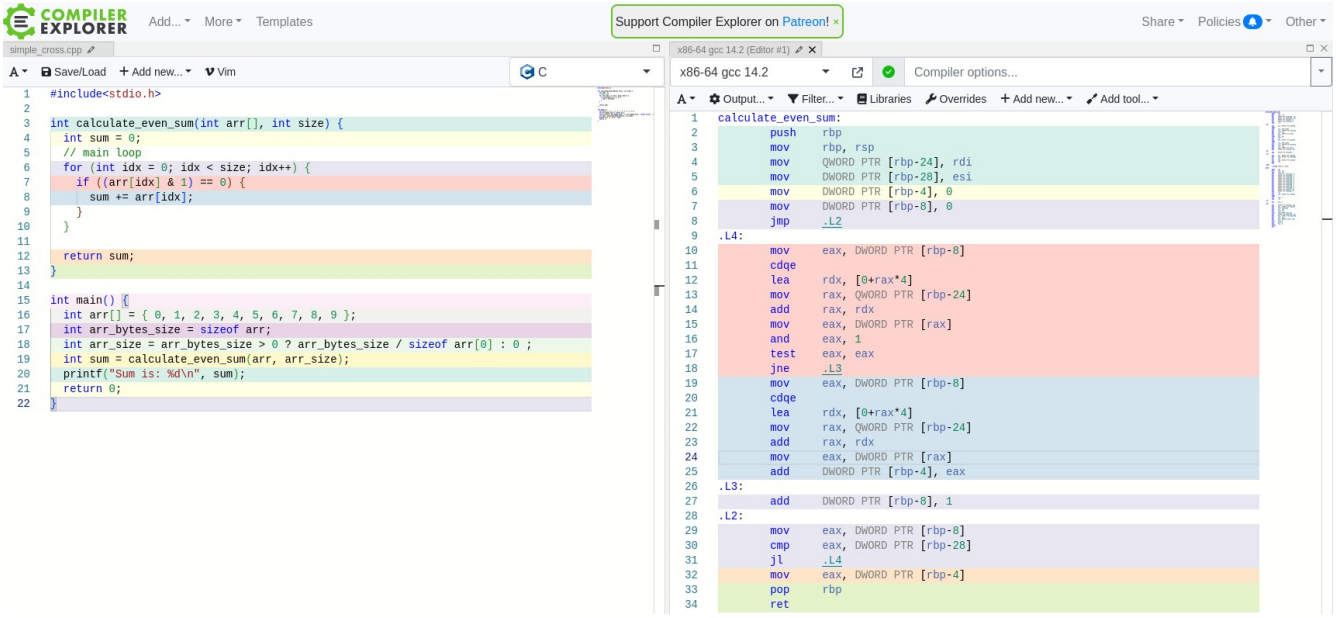

Рис. 1. Використання Compiler Explorer для визначення семантичних відповідників

Після аналізу семантичних відповідників, необхідно додати коментарі до асемблерного коду, які пояснюють що в ньому відбувається. В протоколі такі позначення необхідно виділити кольором. Зокрема, необхідно додати такі позначення:

- початки (позначаються перед першою інструкцією мовою асемблера) і кінці (позначаються після останньої інструкції мови асемблера) структурних блоків:
  - тіло функції;
  - цикл;
  - тіло циклу;
  - умова циклу;
  - умова умовного оператора if/switch;
  - true/false гілки умовного оператора if;
  - case/defualt гілки оператора розгалуження switch;
- відповідні фрагменти коду мовою С біля останньої інструкції мовою асемблера, що їх реалізує.

Для прикладу вище отримаємо наступний результат.

```
	.text
	.globl	calculate_even_sum
	.type calculate_even_sum, @function
calculate_even_sum:
.LFB0:
	.cfi_startproc
	push  rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
```

```
        mov   rbp, rsp
        .cfi_def_cfa_register 6
        // associate actuals with formals
        mov   QWORD PTR [rbp-24], rdi   // associate actual value with formal arr
        mov   DWORD PTR [rbp-28], esi   // associate actual value with formal size
        // function body start
        mov   DWORD PTR [rbp-4], 0      // int sum = 0;
        // main loop start: for (int idx = 0; idx < size; idx++)
        mov   DWORD PTR [rbp-8], 0      // initilize “counter” idx: int idx = 0
        jmp   .L2                       // jump to main loop condition check
.L4:
        // main loop body start
        // if check start: if ((arr[idx] & 1) == 0)
        mov   eax, DWORD PTR [rbp-8]    // save value of idx to EAX
        cdqe                            // extend value in EAX to RAX preserving sign
        lea   rdx, [0+rax*4]            // calculate and save shift in arr to RDX
        mov   rax, QWORD PTR [rbp-24]   // save arr start address to RAX
        add   rax, rdx                  // create address of certain arr[idx]
        mov   eax, DWORD PTR [rax]      // save value from arr[idx] to EAX
        and   eax, 1                    // perform AND operation: arr[idx] & 1
        test  eax, eax                  // perform AND operation on value in EAX with
                                        // itself and set various flags used by
                                        // following jne
        jne   .L3                       // if (arr[idx] & 1) == 0 check fails - jump
                                        // to the end of loop body
        // end if check
        // if true branch start
        mov   eax, DWORD PTR [rbp-8]    // save value of idx to EAX
        cdqe                            // extend value in EAX to RAX preserving sign
        lea   rdx, [0+rax*4]            // calculate and save shift in arr to RDX
        mov   rax, QWORD PTR [rbp-24]   // save arr start address to RAX
        add   rax, rdx                  // create address of certain arr[idx]
        mov   eax, DWORD PTR [rax]      // save value from arr[idx] to EAX
        add   DWORD PTR [rbp-4], eax    // sum += arr[idx];
        // if true branch end
.L3:
        // main loop body end
        add   DWORD PTR [rbp-8], 1      // increment “counter” idx: idx++
.L2:
        // main loop exit condition start
        mov   eax, DWORD PTR [rbp-8]    // save value of idx to register EAX
        cmp   eax, DWORD PTR [rbp-28]   // compare idx with size
        jl    .L4                       // jump to loop body start if idx < size
        // main loop exit condition end
        // main loop end
        mov   eax, DWORD PTR [rbp-4]    // save value in sum to EAX
        pop   rbp
        .cfi_def_cfa 7, 8
        ret                             // return sum;
        // function end
        .cfi_endproc
```

Асемблерний код з коментарями і має потрапити до звіту. Коментарі, позначені синім та зеленим кольорами у прикладі, мають обов’язково бути присутніми у звіті (у звіті їх так само варто виділити кольором). Частини асемблерного коду, відмічені коментарями у прикладі (маються на увазі всі коментарі “// ...”, а не лише виділені кольором), необхідно вміти пояснити. При цьому коментарі з поясненням можна або додати до звіту, або не додавати (але

позначені кольором коментарі мають бути обов'язково). Синім позначені коментарі, що виділяють окремі логічні частини програми (початок певного циклу, перевірка умови циклу, початок тіла циклу, перевірка умови певного if, then/else гілки if та ін.), а зеленим — позначаються коментарі з відповідниками коду мовою C (наприклад, інструкції add DWORD PTR -8[rbp], eax відповідає операція додавання + в програмі, записані мовою C). Деякі відповідники не є точними, оскільки одна інструкція мови асемблера не завжди відповідає одній інструкції мови C. Наприклад, це стосується перевірки умови в if в прикладі вище. Перевірки умови мовою асемблера реалізована двома інструкціями: test і jne (друга також виконує перехід).

Код мовою асемблера в результаті запуску gcc локально, в Compiler Explorer або ж відмінною версією компілятора може дещо відрізнятись. Наприклад:

1. Відносне зміщення у стеку може бути записане або в квадратних дужках, або перед ними: -8[rbp] vs [rbp-8].
2. Різні зміщення можуть бути використані для локальних змінних. Наприклад, sum та idx в результаті запуску можуть мати зміщення rbp-8 і rbp-4 відповідно, або ж навпаки — rbp-4 і rbp-8 відповідно. На результат виконання програми це не впливає.

Опис команд мови асемблера можна знайти в методичних вказівках до лабораторних робіт з курсу "Системне програмування" або в документації фірми Intel — Intel® 64 and IA-32 Architectures Software Developer's Manual.

4. Варіанти завдань

Варіанти завдань для студентів групи визначаються викладачем згідно зі списком групи. Список варіантів надає викладач разом із методичними вказівками. В якості завдань необхідно використати завдання з лабораторної роботи №2.2 з курсу "Структури даних та алгоритми", що викладався на першому курсі, згідно із варіантом, визначеним викладачем. Методичні вказівки до лабораторної роботи з курсу "Структури даних та алгоритми" викладач надає разом з цими методичними вказівками.

5. Контрольні запитання

1) Що таке архітектура комп'ютера?
2) Які існують архітектурні рівні?
3) Як цілі числа представлені в пам'яті комп'ютера? Як представлені числа з рухомою комою?
4) Що таке операція поширення знаку?
5) Поясніть, що виконують команди мови асемблера процесорів фірми Intel: MOV, LEA, JNE, JE, CMP, TEST, CDQE, MOVSX та ін.