



2

Lab

Lập trình ngôn ngữ Assembly cơ bản

Basic Assembly Programming

Thực hành Lập trình Hệ thống

Lưu hành nội bộ

A. TỔNG QUAN

A.1 Mục tiêu

- Tìm hiểu và làm quen với Hợp ngữ (Assembly Language - ASM)
- Tìm hiểu quá trình dịch từ một mã nguồn assembly thành tập tin thực thi
- Thực hành viết một số chương trình mẫu dưới dạng hợp ngữ (theo AT&T)

A.2 Môi trường

- Máy cài Hệ điều hành Linux (máy ảo).
- Các công cụ biên dịch, hợp dịch, liên kết gồm **gcc**, **as**, **ld**.

A.3 Liên quan

- Sinh viên cần vận dụng kiến thức trong Chương 3 (Lý thuyết).
- Các kiến thức này đã được giới thiệu trong nội dung lý thuyết đã học do đó sẽ không được trình bày lại trong nội dung thực hành này.
- Tham khảo tài liệu (Mục F).

B. KIẾN THỨC NỀN TẢNG

B.1 Hợp ngữ là gì?

Hợp ngữ - Assembly Language (hay viết tắt là ASM) là ngôn ngữ bậc thấp, và nằm ở vị trí trung gian bên cạnh ngôn ngữ máy và ngôn ngữ bậc cao.

ASM sử dụng các từ gợi nhớ (mnemonics) để viết các chỉ thị (instructions) lập trình cho máy tính thay vì bằng những dãy số 0 và 1 (ngôn ngữ máy).

```
push %ebp
mov %esp, %ebp
sub $0x8, %esp
movl $0x1, -4(%ebp)
sub $0xc, %esp
push $0x0
call 8048348
```

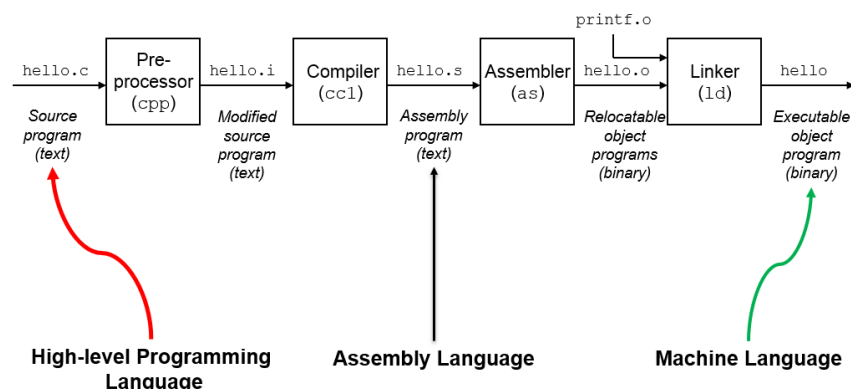
Hình 1. Ví dụ về một đoạn hợp ngữ.

Về **ưu điểm**, Hợp ngữ thân thiện hơn so với ngôn ngữ máy – vốn rất khó hiểu, dễ gây lỗi và tốn nhiều thời giờ. Hợp ngữ có thể tương tác rất sâu dưới hệ thống, chúng có thể giao tiếp trực tiếp với các phần cứng và bắt chúng hoạt động theo ý người lập trình. Bên cạnh đó, hợp ngữ cũng rất hữu ích trong kỹ thuật dịch ngược (reverse engineering) – được sử dụng rộng rãi trong lĩnh vực an toàn thông tin.

Về **nhược điểm**, so với các ngôn ngữ bậc cao quen thuộc (C, C++, Java, Python ...), hợp ngữ ít thân thiện hơn, và phụ thuộc vào kiến trúc CPU (ARM, x86-32, x86-64), hệ điều hành (Linux, Windows, Mac) và các tập chỉ thị mà nhà sản xuất phần cứng đưa ra. Nói cách khác, ASM là một ngôn ngữ không giống như các ngôn ngữ lập trình khác, không có một định dạng chuẩn nào cho các trình hợp dịch (Assembler) sử dụng để dịch các chương trình ASM.

Một chương trình viết bằng ngôn ngữ bậc cao hay hợp ngữ không thể được thực thi trực tiếp bởi máy tính. Sau khi được viết xong, chương trình này phải trải qua quá trình dịch thành ngôn ngữ máy. Quá trình này được mô tả trong **Hình 2** bao gồm các giai đoạn:

- Giai đoạn tiền xử lý (Pre-processor)
- Giai đoạn dịch từ ngôn ngữ bậc cao sang hợp ngữ (Compiler)
- Giai đoạn dịch hợp ngữ sang ngôn ngữ máy (Assembler)
- Giai đoạn liên kết (Linker)



Hình 2. Quá trình dịch từ 1 chương trình dạng C sang tập tin thực thi.

Trong bài thực hành này, sinh viên được yêu cầu viết hợp ngữ theo định dạng AT&T (sử dụng Hệ điều hành Linux, các công cụ biên dịch, hợp dịch như gcc, as, ld)

B.2 Cấu trúc cơ bản và các thành phần của một chương trình hợp ngữ

B.2.1 Cấu trúc của chương trình hợp ngữ

Trong Linux, các chương trình viết bằng mã assembly sẽ có đuôi **.s**. Thông thường chương trình hợp ngữ có ba phần (section) được khai báo:

B.2.1.1 Section **.data**

Là section khai báo những **vùng nhớ** hoặc **hằng số**, nơi chứa các dữ liệu dùng cho chương trình. Thường sử dụng để khai báo biến đã được khởi tạo giá trị hoặc các hằng số sử dụng cho chương trình. Định dạng khai báo:

```
.section .data
<name1>:  <type> <data1>      # vùng nhớ chứa dữ liệu
<name2> = <data2>              # hằng số
...
```

Trong đó:

- + **name** là tên gọi nhớ, có thể được dùng ở các lệnh để truy xuất vùng nhớ hoặc hằng số sau này.
- + **type** là kiểu dữ liệu, ví dụ: int, string, byte...
- + **data** là giá trị cụ thể cần khởi tạo cho vùng nhớ trong vùng nhớ.

Ví dụ: đoạn mã khai báo vùng nhớ **a** lưu số nguyên **10**, vùng nhớ **my_str** lưu chuỗi **"Hello"** và 1 hằng số **b** có giá trị 9.

```
.section .data
a:      .int 10
my_str: .string "Hello"
b = 9
```

B.2.1.2 Section **.bss**

Phần **.bss** khai báo các dữ liệu chưa được gán giá trị (hoặc null), thường được dùng khi cần chuẩn bị sẵn các **vùng nhớ** trống để lưu dữ liệu sau này. Đây là thành phần không bắt buộc, nếu không sử dụng có thể bỏ qua. Định dạng khai báo:

```
.section .bss
    .lcomm <name1>, <length>
    .lcomm <name2>, <length>
...
```

Trong đó:

- + **name** là tên gọi nhớ, có thể được dùng ở các lệnh để truy xuất vùng nhớ này.
- + **length** là kích thước vùng nhớ cần được cấp, tính theo byte.

Ví dụ: đoạn mã bên dưới khai báo vùng nhớ tên **mem** gồm 2 bytes và vùng nhớ **result** có kích thước 4 bytes.

```
.section .bss
    .lcomm mem, 2
    .lcomm result, 4
```

B.2.1.3 Section .text

Là section **bắt buộc** phải có trong tất cả các chương trình hợp ngữ. Đây là nơi các câu lệnh được khai báo. Định dạng khai báo:

```
.section .text
    .globl _start
```

```
_start:
    // các lệnh assembly
```

Sau nhãn `_start`, các câu lệnh assembly sẽ được viết tùy thuộc vào chức năng cần lập trình. Trong đó có thể sử dụng các vùng nhớ đã được khai báo ở 2 section `.data` hoặc `.bss` để lấy hoặc lưu trữ dữ liệu vào các vùng nhớ nhất định, hoặc tương tác với các thanh ghi.

Với 1 vùng nhớ tên **a** được khai báo trong section `.data` (hoặc `.bss`):

- Ký hiệu **\$a** tương ứng với việc lấy địa chỉ của vùng nhớ **a**, được xem là 1 hằng số.
- Ký hiệu **(a)** hoặc **a** tương ứng với việc truy xuất giá trị đang lưu tại vùng nhớ **a**.

B.2.2 Định dạng lệnh assembly

Trong phạm vi bài thực hành này, các lệnh assembly được viết theo định dạng AT&T và tập lệnh của kiến trúc 32bit.

B.3 Linux System Call (Hàm gọi hệ thống Linux)

Có nhiều lời gọi hệ thống (system call) được cung cấp bởi kernel Linux và biết cách tìm cũng như sử dụng chúng rất có lợi trong việc lập trình hợp ngữ (assembly language). Những system call này có sẵn cho các lập trình viên sử dụng. Thông thường, với mỗi lần phát hành một kernel mới, các system call mới được thêm vào danh sách.

Các system call thường được định nghĩa trong file sau:

`/usr/include/asm/unistd.h` hoặc `/usr/include/asm-generic/unistd.h`

Không giống như các hàm kiểu C, trong đó các giá trị đầu vào được đặt trên stack, các system call yêu cầu đầu vào sao cho các giá trị được đặt trong thanh ghi. Có một thứ tự cụ thể trong đó mỗi giá trị đầu vào được đặt trong các thanh ghi tương ứng. Đặt giá trị đầu vào trong một thanh ghi không đúng chuẩn có thể tạo ra kết quả sai.

Thứ tự trong đó các system call mong đợi các giá trị đầu vào như sau:

- | | |
|---------------------------|--------------------------|
| - %eax | - %edx (tham số thứ ba) |
| - %ebx (tham số đầu tiên) | - %esi (tham số thứ tư) |
| - %ecx (tham số thứ hai) | - %edi (tham số thứ năm) |

Sử dụng quy ước này, các giá trị đầu vào sẽ được gán cho các thanh ghi sau:

- %eax: Giá trị của hàm gọi hệ thống (tham khảo bảng bên dưới)
- %ebx: Bộ mô tả tệp (số nguyên)
- %ecx: Con trỏ (địa chỉ bộ nhớ) của chuỗi cần hiển thị
- %edx: Kích thước của chuỗi cần hiển thị

%eax	Name	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-

Giá trị mô tả tệp cho vị trí đầu ra (hoặc vào) để chỉ định vị trí nhận hoặc xuất dữ liệu được đặt trong %ebx. Các hệ thống Linux chứa ba mô tả tệp đặc biệt:

- 0 (STDIN): Đầu vào tiêu chuẩn cho thiết bị đầu cuối (thường là bàn phím)
- 1 (STDOUT): Đầu ra tiêu chuẩn cho thiết bị đầu cuối (thường là màn hình đầu cuối)
- 2 (STDERR): Đầu ra lỗi tiêu chuẩn cho thiết bị đầu cuối (thường là màn hình đầu cuối)

Ví dụ để gọi hàm in ra màn hình chuỗi "Hello, World".

- **%eax = 4** tương ứng với chỉ thị in chuỗi.
- **%ebx = 1** tương ứng với vị trí in ra là stdout (terminal).
- **%ecx = địa chỉ ô nhớ** đang chứa chuỗi cần in.
- **%edx = độ dài** (tính theo byte) của chuỗi cần in.

Sau khi khai báo các giá trị tham số cần thiết của system call trong các thanh ghi, lệnh **int \$0x80** được dùng để thực thi system call đó.

```
.section .data
output:
    .string "Hello, World"

.section .text
.globl _start
_start:
    movl $13, %edx        # message length
    movl $output, %ecx     # address of message to write.
    movl $1, %ebx         # file descriptor (stdout)
    movl $4, %eax          # system call number (sys_write)
    int $0x80              # call kernel

    movl $1, %eax          # system call number (sys_exit)
    int $0x80              # call kernel
```

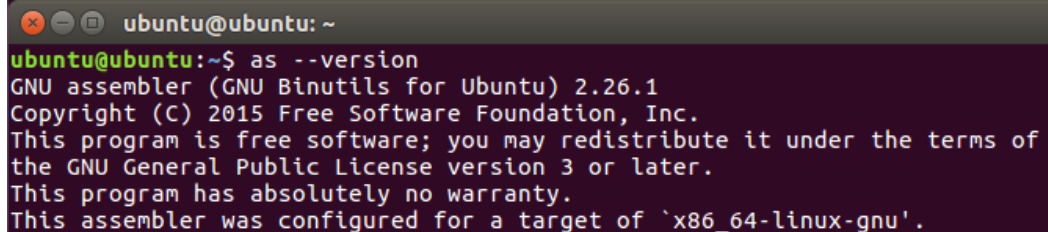
C. THỰC HÀNH

C.1 Yêu cầu 1 – Thiết lập môi trường

Sinh viên cần chuẩn bị môi trường máy ảo Linux có đầy đủ các công cụ cần thiết bao gồm **as** và **ld** để sử dụng trong quá trình biên dịch file mã hợp ngữ sang file thực thi.

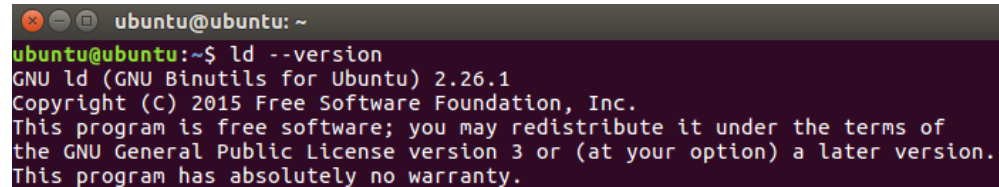
Các công cụ trên có thể được kiểm tra đã được cài đặt trên hệ thống chưa bằng việc gõ các lệnh sau trong terminal:

```
as --version
```



```
ubuntu@ubuntu: ~  
ubuntu@ubuntu:~$ as --version  
GNU assembler (GNU Binutils for Ubuntu) 2.26.1  
Copyright (C) 2015 Free Software Foundation, Inc.  
This program is free software; you may redistribute it under the terms of  
the GNU General Public License version 3 or later.  
This program has absolutely no warranty.  
This assembler was configured for a target of `x86_64-linux-gnu'.
```

```
ld --version
```



```
ubuntu@ubuntu: ~  
ubuntu@ubuntu:~$ ld --version  
GNU ld (GNU Binutils for Ubuntu) 2.26.1  
Copyright (C) 2015 Free Software Foundation, Inc.  
This program is free software; you may redistribute it under the terms of  
the GNU General Public License version 3 or (at your option) a later version.  
This program has absolutely no warranty.
```

Nếu có kết quả nào trả về không tìm thấy công cụ, sinh viên gõ lệnh để cài đặt:

```
sudo apt-get install binutils
```

C.2 Yêu cầu 2 – Bài tập lập trình hợp ngữ (assembly)

Hướng dẫn chung: Sinh viên thực hiện viết các chương trình dưới dạng hợp ngữ (assembly) trong các file **.s**, sau đó thực hiện các bước hợp dịch và liên kết như sau:

- Hợp dịch với công cụ **as** để thu được một file nhị phân **.o** từ file **.s**:

```
as -o <file .o> <file .s đầu vào>
```

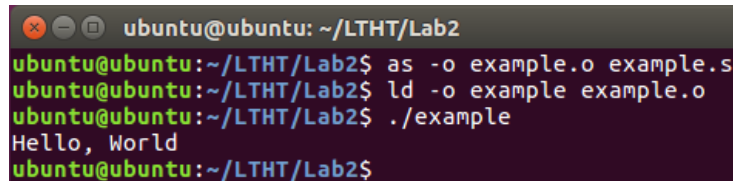
- Liên kết với công cụ **ld** để tạo file thực thi của chương trình

```
ld -o <file thực thi> <file .o đầu vào>
```

- Thực thi file

```
./<file thực thi đầu ra>
```

Ví dụ:



```
ubuntu@ubuntu: ~/LTHT/Lab2  
ubuntu@ubuntu:~/LTHT/Lab2$ as -o example.o example.s  
ubuntu@ubuntu:~/LTHT/Lab2$ ld -o example example.o  
ubuntu@ubuntu:~/LTHT/Lab2$ ./example  
Hello, World  
ubuntu@ubuntu:~/LTHT/Lab2$
```

C.2.1 Chương trình in ra độ dài của một chuỗi cho trước (tối đa 9 ký tự)

Input: Chuỗi **msg** được khai báo sẵn trong section **.data**, tối đa 9 ký tự.

Output: Xuất ra màn hình độ dài của chuỗi (số ký tự - không tính ký tự null).

Ví dụ: Với chuỗi **msg = "Love UIT"**, kết quả đầu ra như hình dưới:

```
ubuntu@ubuntu: ~/LTHT/Lab2/2024/C21
ubuntu@ubuntu:~/LTHT/Lab2/2024/C21$ as -o c21.o c21.s
ubuntu@ubuntu:~/LTHT/Lab2/2024/C21$ ld -o c21 c21.o
ubuntu@ubuntu:~/LTHT/Lab2/2024/C21$ ./c21
8
ubuntu@ubuntu:~/LTHT/Lab2/2024/C21$
```

Gợi ý 1.1: Phần **D.3** hướng dẫn cách lấy được độ dài của một chuỗi đã khai báo trong section **.data**. Lưu ý độ dài này sẽ có tính luôn ký tự null ở cuối chuỗi.

Gợi ý 1.2: Để in một dữ liệu nào đó ra màn hình, cần đảm bảo 2 yếu tố:

- Dữ liệu cần in cần **nằm trong 1 vùng nhớ** (trong section **.bss** hoặc **.data**), không hỗ trợ in trực tiếp giá trị từ thanh ghi hay hằng số.
- Khi in, hệ thống tự động xem dữ liệu được cung cấp là **các mã ASCII** của 1 hoặc nhiều ký tự nào đó và in ra các ký tự tương ứng. Do vậy, nếu dữ liệu không phải là 1 mã ASCII in được thì khi in sẽ không nhìn thấy được. Ví dụ: Khi dữ liệu là 65 thì hệ thống sẽ in ra ký tự 'A', còn dữ liệu là 0 thì không in ra được gì.

Gợi ý 1.3: Ta cần thiết lập giá trị các thanh ghi cho việc xuất dữ liệu như sau:

- %eax: SYS_WRITE (4).
- %ecx: **địa chỉ ô nhớ** có dữ liệu cần xuất
- %ebx: với STDOUT (1).
- %edx: độ dài sẽ xuất ra (bytes).

C.2.2 Chương trình in MSSV (8 ký tự) từ SBD (6 ký tự)

Input: SBD gồm 6 ký tự (bản rút gọn của MSSV, giới hạn SBD từ 15xxxx trở đi)

Output: Xuất ra màn hình **MSSV** (8 ký tự, thêm vào 2 ký tự "52").

Yêu cầu: Sử dụng tối đa **02 lần** in ra màn hình để in các kết quả.

Nâng cao: In thêm thông tin niên khóa và là sinh viên năm mấy (năm 1, 2...), giới hạn **03 lần** in ra màn hình.

Ví dụ (phần cơ bản):

```
ubuntu@ubuntu: ~/LTHT/Lab2/2024/C22
ubuntu@ubuntu:~/LTHT/Lab2/2024/C22$ as -o c22.o c22.s
ubuntu@ubuntu:~/LTHT/Lab2/2024/C22$ ld -o c22 c22.o
ubuntu@ubuntu:~/LTHT/Lab2/2024/C22$ ./c22
Nhap SBD (6 ky tu): 221362
22521362
ubuntu@ubuntu:~/LTHT/Lab2/2024/C22$ ./c22
Nhap SBD (6 ky tu): 200143
20520143
ubuntu@ubuntu:~/LTHT/Lab2/2024/C22$
```


Kết quả phân nâng cao:

```
ubuntu@ubuntu: ~/LTHT/Lab2/2024/C22
ubuntu@ubuntu:~/LTHT/Lab2/2024/C22$ as -o c22.o c22.s
ubuntu@ubuntu:~/LTHT/Lab2/2024/C22$ ld -o c22 c22.o
ubuntu@ubuntu:~/LTHT/Lab2/2024/C22$ ./c22
Nhap SBD (6 ky tu): 221362
22521362, Nien khoa: 2022, Sinh vien nam: 2
ubuntu@ubuntu:~/LTHT/Lab2/2024/C22$ ./c22
Nhap SBD (6 ky tu): 200143
20520143, Nien khoa: 2020, Sinh vien nam: 4
ubuntu@ubuntu:~/LTHT/Lab2/2024/C22$
```

Gợi ý 2.1: Để nhận input từ bàn phím, ta thực hiện các bước:

- Chuẩn bị sẵn 1 vùng nhớ trong section .bss có độ dài phù hợp để lưu input.

Ví dụ bên dưới tạo 1 vùng nhớ có kích thước **2 bytes** (lưu 2 ký tự) có nhãn là **number1**.

```
.section .bss
.lcomm number1, 2
```

- Ta cần thiết lập giá trị các thanh ghi cho 1 system call nhập dữ liệu như sau:

- o %eax: SYS_READ (3).
- o %ecx: **địa chỉ ô nhớ** để lưu giá trị nhập vào (đã tạo trong section .bss)
- o %ebx: với STDIN (0).
- o %edx: độ dài của chuỗi sẽ nhận

Lưu ý: Độ dài nhập vào cần tính cả ký tự xuống dòng. Do đó, với input là 1 ký tự hay số 1 chữ số cần khai báo độ dài của chuỗi nhập vào trong %edx là 2.

Gợi ý 2.2: 1 ký tự chỉ ứng với 1 byte, chỉ nên lấy đủ 1 byte tương ứng để xử lý ký tự đã nhập. Dựa vào địa chỉ của chuỗi để truy xuất các ký tự dựa trên chỉ số của chúng. Ví dụ bên dưới đoạn mã lấy ký tự thứ 2 (chỉ số index = 1) của chuỗi **input**.

```
movl $input, %eax
mov 1(%eax), %bl
```

Gợi ý 2.3 (cho phần nâng cao): Dữ liệu nhập vào là các ký tự số, cần chuyển từ ký tự số sang giá trị số nguyên để tính toán chính xác (ví dụ: chuyển từ '1' sang 1, xem phần **D.2**).

C.2.3 Tính và in ra màn hình số dư khi tổng hai số a và b chia cho 4

Input: Hai số nguyên a và b có 2 chữ số ($10 \leq a, b \leq 99$).

Output: Kết quả của phép tính $(a+b) \% 4$ (chia lấy dư)

Ví dụ:

```
ubuntu@ubuntu: ~/LTHT/Lab2/2024/C23
ubuntu@ubuntu:~/LTHT/Lab2/2024/C23$ as -o c23.o c23.s
ubuntu@ubuntu:~/LTHT/Lab2/2024/C23$ ld -o c23 c23.o
ubuntu@ubuntu:~/LTHT/Lab2/2024/C23$ ./c23
Enter a number (2-digit): 10
Enter a number (2-digit): 99
Reminder: 1
ubuntu@ubuntu:~/LTHT/Lab2/2024/C23$ ./c23
Enter a number (2-digit): 80
Enter a number (2-digit): 15
Reminder: 3
ubuntu@ubuntu:~/LTHT/Lab2/2024/C23$ ./c23
Enter a number (2-digit): 20
Enter a number (2-digit): 20
Reminder: 0
ubuntu@ubuntu:~/LTHT/Lab2/2024/C23$
```

Gợi ý 3.1: Nhập dữ liệu với system call như ở **C.2.2**.

Gợi ý 3.2: Xem phần **D.2** để chuyển từ chuỗi ký tự số sang số nguyên tương ứng.

Gợi ý 3.3: Tham khảo về lệnh div: [div instruction](#) để thực hiện phép chia.

C.2.4 Mã hóa Caesar cho chuỗi gồm 5 ký tự

Input: Chuỗi gồm 5 ký tự chữ in hoa, số bước dịch **n** (0 – 9), trong đó với 1 ký tự **x** thì $'A' \leq x + n \leq 'Z'$.

Output: Xuất ra chuỗi ký tự đã được mã hóa Caesar với bước dịch **n**.

Nâng cao: Xử lý được trường hợp xoay vòng khi kết quả sau khi dịch vượt quá ký tự 'Z'.

Ví dụ 'Z' + 1 = 'A', 'Z' + 2 = 'B'.

Ví dụ (phần cơ bản):

```
ubuntu@ubuntu: ~/LTHT/Lab2/2024/C24
ubuntu@ubuntu:~/LTHT/Lab2/2024/C24$ as -o c24.o c24.s
ubuntu@ubuntu:~/LTHT/Lab2/2024/C24$ ld -o c24 c24.o
ubuntu@ubuntu:~/LTHT/Lab2/2024/C24$ ./c24
Nhap chuoi (5 ky tu): ABCDE
Nhap n (0-9): 1
BCDEF
ubuntu@ubuntu:~/LTHT/Lab2/2024/C24$ ./c24
Nhap chuoi (5 ky tu): ABCDE
Nhap n (0-9): 5
FGHIJ
ubuntu@ubuntu:~/LTHT/Lab2/2024/C24$
```

Kết quả phần nâng cao:

```
ubuntu@ubuntu: ~/LTHT/Lab2/2024/C24
ubuntu@ubuntu:~/LTHT/Lab2/2024/C24$ as -o c24.o c24.s
ubuntu@ubuntu:~/LTHT/Lab2/2024/C24$ ld -o c24 c24.o
ubuntu@ubuntu:~/LTHT/Lab2/2024/C24$ ./c24
Nhap chuoi (5 ky tu): ABXYZ
Nhap n (0-9): 1
BCYZA
ubuntu@ubuntu:~/LTHT/Lab2/2024/C24$ ./c24
Nhap chuoi (5 ky tu): NAMEZ
Nhap n (0-9): 2
PCOGB
ubuntu@ubuntu:~/LTHT/Lab2/2024/C24$
```

D. THAM KHẢO

D.1 Một số lưu ý khi lập trình assembly định dạng AT&T

- **Cần phân biệt được các toán hạng:**

Khác biệt trong ký hiệu của các dạng toán hạng khác nhau:

- Hằng số: **\$1**
- Thanh ghi: **%eax**
- Địa chỉ ô nhớ: **\$output** với output là nhãn của ô nhớ trong .data hoặc .bss
- Giá trị trong ô nhớ: **(output)** hoặc **output**

- **Trong các câu lệnh assembly, cần đảm bảo:**

- Có **tối đa 1 toán hạng là hằng số**, nếu có hằng số cần đứng ở vị trí src.
Ví dụ: `addl $1, %eax`
- Có **tối đa 1 toán hạng liên quan đến truy xuất ô nhớ**.

- **Cách viết biểu thức địa chỉ để truy xuất ô nhớ**

Giả sử với ô nhớ có nhãn tên **output**

- Truy xuất giá trị ô nhớ từ địa chỉ bắt đầu: viết dưới dạng **(output)** hoặc **output**.
- Truy xuất giá trị ô nhớ từ địa chỉ cách n bytes: cần đưa địa chỉ vào thanh ghi và dùng biểu thức tính toán địa chỉ

```
movl $output, %eax
movl 1(%eax), %ebx    // truy xuất từ địa chỉ cách 1 byte
```

D.2 Bảng mã ASCII

- **Chuyển đổi số có 1 chữ số**

Sử dụng bảng mã ASCII để chuyển đổi từ số (decimal) sang chữ (dạng ascii) và ngược lại. Tham khảo tại <https://www.ascii-code.com/>. Ví dụ: Có thể chuyển đổi từ số 5 sang ký tự '5' bằng cách cộng **\$48** (ký tự '0'). Ngược lại, để chuyển từ ký tự sang số thì trừ ký tự '0'.

- **Chuyển đổi số có nhiều chữ số**

Tách riêng từng số (hay ký tự số) sau đó áp dụng cách chuyển đổi trên số có 1 chữ số để chuyển qua lại giữa số và ký tự số tương ứng của nó.

Ví dụ: Chuyển chuỗi ký tự số '123' sang số nguyên: $1 \times 100 + 2 \times 10 + 3 = 123$.

D.3 Lấy độ dài chuỗi trong section .data

Như đã trình bày ở trên, trong section .data có thể dùng để khai báo một số biến đã gán trước giá trị hoặc hằng số được dùng trong chương trình, ví dụ chuỗi output sẽ in ra màn hình. Tuy nhiên, thay vì gán cứng độ dài các chuỗi này khi dùng trong các system call, đoạn khai báo bên dưới cho phép lấy giá trị độ dài của chuỗi **rs**, lưu ý dù nằm trong .data nhưng **rs_len** là hằng số.

```
.section .data:
rs: .string "Max is "
rs_len = . -rs
```

E. YÊU CẦU & ĐÁNH GIÁ

Sinh viên thực hành theo **nhóm tối đa 3 sinh viên**, có thể nộp bài theo 2 hình thức:

- Nộp trực tiếp trên lớp: báo cáo và demo kết quả với GVTH.
- Nộp file code tại website môn học theo thời gian quy định, *có chú thích chức năng của các đoạn code.*

Tạo **thư mục** tên **Lab2-NhomX-MSSV1-MSSV2-MSSV3**, trong đó nộp đầy đủ **các file .s** với định dạng:
Lab2-NhomX-<yêu cầu>.s
Ví dụ: *Lab2-Nhom2-C21.s*

F. THAM KHẢO

[1] Linux assemblers: A comparison of GAS and NASM [Online] Available at:

<https://www.ibm.com/developerworks/library/l-gas-nasm/index.html>

[2] Randal E. Bryant, David R. O'Hallaron (2011). *Computer System: A Programmer's Perspective (CSAPP)*

[3] Richard Blum (2005). *Professional Assembly Language*

HẾT