**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP.HỒ CHÍ MÌNH**

**KHOA ĐÀO TẠO CHẤT LƯỢNG CAO**



****

**BÁO CÁO MÔN HỌC**

**(MÔN HỌC: Kiến trúc máy tính và hợp ngữ)**

**GVHD: Th.S Nguyễn Đăng Quang**

**SVTH: Hồ Hoài Phong**

**MSSV: 19110262**

**TP.HỒ CHÍ MINH – Tháng 06/20**

**Mục Lục**

Table of Contents

[Giới thiệu về đề tài 2](#_Toc73553279)

[CHƯƠNG 1. Chương trình C 3](#_Toc73553280)

[1.1 Biến trong chương trình C 3](#_Toc73553281)

[1.2 Danh sách các biến dùng trong chương trình C 3](#_Toc73553282)

[1.3 Chương trình chính 3](#_Toc73553283)

[1.4 Các chương trình con 4](#_Toc73553284)

[1.4.1 Hàm lấy độ dài chuỗi 4](#_Toc73553285)

[1.4.2 Hàm đọc các kí tự có trong file text 6](#_Toc73553286)

[CHƯƠNG 2. Chương trình hợp ngữ 8](#_Toc73553287)

[2.1 Input 8](#_Toc73553288)

[2.2 Hàm writeData 10](#_Toc73553289)

[2.2.1 Hàm readData 11](#_Toc73553290)

[2.3 Hàm print\_Address\_Hexdump 18](#_Toc73553291)

[2.3.1 Hàm write\_hex\_dword 21](#_Toc73553292)

[2.3.2 Chép 16 kí tự vào printTable 36](#_Toc73553293)

[2.3.3 In ra màn hình cột kí tự 47](#_Toc73553294)

[KẾT LUẬN 49](#_Toc73553295)

[Tài Liệu Tham Khảo 50](#_Toc73553296)

**Mục lục hình ảnh**

[Hình 1.4.1‑1: Kết quả của chương trình 2](#_Toc73553749)

[Hình 1.4.2‑1 Dữ liệu đầu vào của hàm 8](#_Toc73553750)

[Hình 1.4.2‑2 Đặt breakpoint ở dòng 42 9](#_Toc73553751)

[Hình 1.4.2‑3 Breakpoint ở dòng 42 9](#_Toc73553752)

[Hình 1.4.2‑4 Chương trình con trong display\_memory 10](#_Toc73553753)

[Hình 1.4.2‑1 Hàm writeData 10](#_Toc73553754)

[Hình 1.4.2‑2 Khai báo mảng buff 10](#_Toc73553755)

[Hình 2.2.1‑1 Hàm readData 11](#_Toc73553756)

[Hình 2.2.1‑2 Debug hàm readData 11](#_Toc73553757)

[Hình 2.2.1‑3 Copy esp vào ebp 12](#_Toc73553758)

[Hình 2.2.1‑4 Stack frame của 2 tham số 13](#_Toc73553759)

[Hình 2.2.1‑5 Gọi Chương trình con writeData 13](#_Toc73553760)

[Hình 2.2.1‑6 Hình ảnh gọi chương trình con readData 14](#_Toc73553761)

[Hình 2.2.1‑7 Giá trị thanh ghi ebp 14](#_Toc73553762)

[Hình 2.2.1‑8 Gía trị trong stack 14](#_Toc73553763)

[Hình 2.2.1‑9 Gía trị của thanh ghi edx và ecx 15](#_Toc73553764)

[Hình 2.2.1‑10 Khai báo cho biến length 15](#_Toc73553765)

[Hình 2.2.1‑11 Hình ảnh cho biến length 15](#_Toc73553766)

[Hình 2.2.1‑12 Giải thích hàm writeData 16](#_Toc73553767)

[Hình 2.2.1‑13 Qúa trình lặp để đưa vào từng kí tự 16](#_Toc73553768)

[Hình 2.2.1‑14 Khi thanh ghi esi bằng 16 17](#_Toc73553769)

[Hình 2.2.1‑15 Chương trình quay lại hàm display\_memory 17](#_Toc73553770)

[Hình 2.2.1‑16 Kết quả của mảng buff sau khi chép 16 kí tự 18](#_Toc73553771)

[Hình 2.2.1‑17 Kết quả của biến length 18](#_Toc73553772)

[Hình 2.2.1‑1 Kết quả được in trên 1 dòng 18](#_Toc73553773)

[Hình 2.2.1‑2 Chạy gdb hàm print\_Adress\_hexdump 19](#_Toc73553774)

[Hình 2.2.1‑3 Hàm print\_Address\_hexdump 20](#_Toc73553775)

[Hình 2.2.1‑4 Hàm print\_Address\_Hexdump 20](#_Toc73553776)

[Hình 2.2.1‑5 Hàm print\_Address\_Hexdump 21](#_Toc73553777)

[Hình 2.3.1‑1 Khởi tạo biến address 21](#_Toc73553778)

[Hình 2.3.1‑2 Tăng địa chỉ address sau mỗi lần in 21](#_Toc73553779)

[Hình 2.3.1‑3 Gán giá trị address cho thanh ghi edx 22](#_Toc73553780)

[Hình 2.3.1‑4 Xem giá trị thanh ghi edx 22](#_Toc73553781)

[Hình 2.3.1‑5 Hình ảnh hàm write\_hex\_dword ở display.asm 23](#_Toc73553782)

[Hình 2.3.1‑6 Khai báo chuỗi kí tự arrAddress 23](#_Toc73553783)

[Hình 2.3.1‑7 Chuẩn bị cho hàm write\_hex\_dword 24](#_Toc73553784)

[Hình 2.3.1‑8 Xem kết quả edx tại hàm write\_hex\_dwordỞ hình 2.6.1-7 em sẽ xóa 3 thanh ghi eax,esi,edi và đưa giá trị của thanh ghi edx cho ebx 24](#_Toc73553785)

[Hình 2.3.1‑9 Nhãn loop\_1 24](#_Toc73553786)

[Hình 2.3.1‑10 Nhãn flag\_bit\_1 25](#_Toc73553787)

[Hình 2.3.1‑11 Nhãn \_loop2 25](#_Toc73553788)

[Hình 2.3.1‑12 Debug write\_hex\_dword 26](#_Toc73553789)

[Hình 2.3.1‑13 Debug write\_hex\_dword 27](#_Toc73553790)

[Hình 2.3.1‑14 Debug write\_hex\_dword 27](#_Toc73553791)

[Hình 2.3.1‑15 Debug write\_hex\_dword 27](#_Toc73553792)

[Hình 2.3.1‑16 Debug write\_hex\_dword 28](#_Toc73553793)

[Hình 2.3.1‑17 Hàm write\_hex\_reverse\_flag\_0 28](#_Toc73553794)

[Hình 2.3.1‑18 Hàm write\_hex\_reverse\_flag\_0 29](#_Toc73553795)

[Hình 2.3.1‑19 Hàm write\_hex\_reverse\_flag\_0 29](#_Toc73553796)

[Hình 2.3.1‑20 Hàm write\_hex\_reverse\_flag\_0 30](#_Toc73553797)

[Hình 2.3.1‑21 Hàm write\_hex\_digit\_dWord 30](#_Toc73553798)

[Hình 2.3.1‑22 Hàm write\_hex\_digit\_dWord 30](#_Toc73553799)

[Hình 2.3.1‑23 Hàm write\_hex\_dword 31](#_Toc73553800)

[Hình 2.3.1‑24 Hàm write\_hex\_dword 31](#_Toc73553801)

[Hình 2.3.1‑25 Hàm write\_hex\_dword 32](#_Toc73553802)

[Hình 2.3.1‑26 Hàm write\_hex\_dword 32](#_Toc73553803)

[Hình 2.3.1‑27 Hàm write\_hex\_dword 33](#_Toc73553804)

[Hình 2.3.1‑28 Hàm write\_hex\_dword 33](#_Toc73553805)

[Hình 2.3.1‑29 Hàm write\_hex\_dword 34](#_Toc73553806)

[Hình 2.3.1‑30 Hàm write\_hex\_dword 34](#_Toc73553807)

[Hình 2.3.1‑31 Hàm write\_hex\_dword 34](#_Toc73553808)

[Hình 2.3.1‑32 Hàm write\_hex\_dword 35](#_Toc73553809)

[Hình 2.3.1‑33 Kết quả sau khi ra khỏi hàm write\_hex\_dword 35](#_Toc73553810)

[Hình 2.3.2‑1 In 16 bit kí tự về dạng hex 36](#_Toc73553811)

[Hình 2.3.2‑2 In 16 bit kí tự về dạng hex 36](#_Toc73553812)

[Hình 2.3.2‑3 In 16 bit kí tự 37](#_Toc73553813)

[Hình 2.3.2‑4 In 16 bit kí tự 37](#_Toc73553814)

[Hình 2.3.2‑5 In 16 bit kí tự 38](#_Toc73553815)

[Hình 2.3.2‑6 In 16 bit ki tự 38](#_Toc73553816)

[Hình 2.3.2‑7 In 16 bit kí tự 39](#_Toc73553817)

[Hình 2.3.2‑8 In 16 bit kí tự 39](#_Toc73553818)

[Hình 2.3.2‑9 In 16 bit kí tự 39](#_Toc73553819)

[Hình 2.3.2‑10 In 16 bit kí tự 40](#_Toc73553820)

[Hình 2.3.2‑11 In 16 bit kí tự 41](#_Toc73553821)

[Hình 2.3.2‑12 In ra màn hình 16 kí tự bằng hệ hex 41](#_Toc73553822)

[Hình 2.3.2‑13 In ra màn hình 16 kí tự bằng hệ hex 42](#_Toc73553823)

[Hình 2.3.2‑14 In ra màn hình 16 kí tự bằng hệ hex 43](#_Toc73553824)

[Hình 2.3.2‑15 In ra màn hình 16 kí tự bằng hệ hex 43](#_Toc73553825)

[Hình 2.3.2‑16 In ra màn hình 16 kí tự bằng hệ hex 43](#_Toc73553826)

[Hình 2.3.2‑17 Kết quả khi truyền vào đủ 16 kí tự 44](#_Toc73553827)

[Hình 2.3.2‑18 Kết quả nếu không truyền đủ 16 kí tự 44](#_Toc73553828)

[Hình 2.3.2‑19 In ra màn hình 16 kí tự hệ hex 44](#_Toc73553829)

[Hình 2.3.2‑20 In ra màn hình 16 kí tự bàng hex 45](#_Toc73553830)

[Hình 2.3.2‑21 In 2 lần khoảng cách 45](#_Toc73553831)

[Hình 2.3.2‑22 In 2 lần khoảng cách 46](#_Toc73553832)

[Hình 2.3.2‑23 In 2 lần khoảng cách 46](#_Toc73553833)

[Hình 2.3.2‑24 Trường hợp nhỏ hơn 16 kí tự 47](#_Toc73553834)

[Hình 2.3.2‑25 Trường hợp nhỏ hơn 16 kí tự 47](#_Toc73553835)

[Hình 2.3.3‑1 In ra màn hình cột kí tự 47](#_Toc73553836)

[Hình 2.3.3‑2 In ra màn hình cột kí tự 48](#_Toc73553837)

[Hình 2.3.3‑3 In ra màn hình cột kí tự 48](#_Toc73553838)

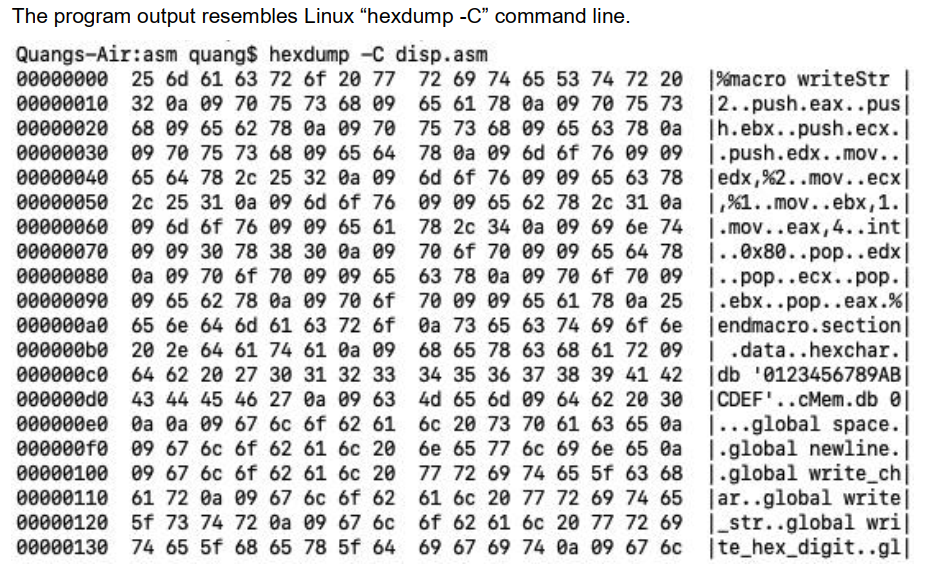
[Hình 2.3.3‑4 In ra màn hình cột kí tự 48](#_Toc73553839)

[Hình 2.3.3‑5 Xử lý kiểu này sẽ giúp cột kí tự không bị xô lệch 48](#_Toc73553840)

**NỘI DUNG BÁO CÁO**

# Giới thiệu về đề tài

* **Tên đề tài**: Viết chương trình C đọc file và in ra màn hình nội dung của file ở cả hai dạng hex va asci
* **Nội dung của đề tài**: Sử dụng chương trình C và hợp ngữ để xứ lý và in ra màn hình, chương trình C bắt buộc phải gọi chương trình con từ hợp ngữ với tên chương trình con display\_memory() được tạo trong mem.asm (function library) và mem.asm lại gọi tới function từ display.asm (function library).
* **Kết quả của đề tài**: Chương trình in ra kết quả như lệnh “hexdump –C” trong Linux



Hình 1.4.1‑1: Kết quả của chương trình

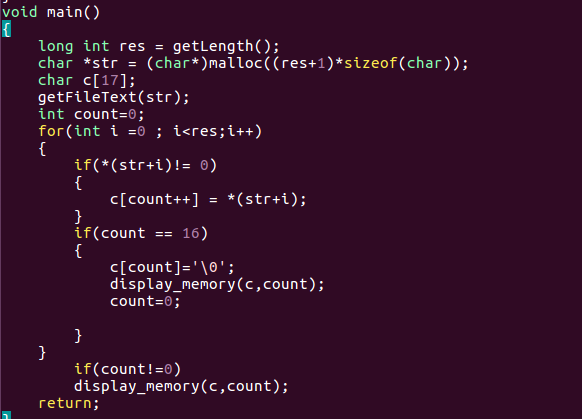
# Chương trình C

## Biến trong chương trình C

## Danh sách các biến dùng trong chương trình C

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| STT | Kiểu dữ liệu | Kiểu biến | Tên biến | Mô tả |
| 1 | FILE | Biến con trỏ | fp | Nhiệm vụ của biến này dùng để đọc file text và lấy dữ liệu từ file text thông qua các hàm được xây dựng trong lớp FILE của C |
| 2 | char | Biến con trỏ | str | Nhiệm vụ của biến này lưu số lượng kí tự có trong file text. |
| 3 | char[] | Mảng char | c | Mảng c này sẽ đọc 16 kí tự từ str |
| 4 | int | Số nguyên | count | Nhiệm vụ của biến count này là biến đếm để lưu vào ô nhớ tiếp theo của mảng kí tự c |

## Chương trình chính



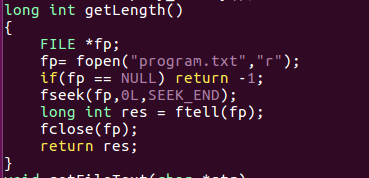
Hình 2‑1 Hình ảnh hàm main trong C

* Biến res;
  + Lấy ra độ dài chuỗi có trong file “program.txt”
* Biến con trỏ \*str
  + Cấp phát độ dài mảng con trỏ bằng malloc đúng bằng độ dài của file “program.txt” + 1 để lưu kí tự kết chuỗi
  + Mảng kí tự còn đươc truyền để làm tham số ở chương trình con trong hợp ngữ như là một mảng kí tự
  + Dùng hàm getFileText(str) để lấy số lượng kí tự có trong file “program.txt”
* Lưu kí tự có trong con trỏ str vào mảng kí tự c[17];
  + Khai báo mảng kí tự có 17 phần tử do phải cần phải có kí tự kết thúc chuỗi
* Biến count
  + Dùng để làm biến đếm để giúp mảng kí tự char lưu vào từng ô nhớ.
  + Biến đếm còn được truyền làm tham số ở chương trình con trong hợp ngữ như là độ dài hiện tại của chuỗi kí tự nhận được
* Ý tưởng vòng lặp for:
  + Em sẽ dùng vòng lặp for đưa từng kí tự để đưa từng kí tự vào mảng char c[]
  + Điều kiện để đưa vào mảng char là không có 2 kí tự như ‘\n’ và kí tự kết thúc chuỗi
  + Cứ mỗi lần nhận một kí tự hợp lệ thì biến count sẽ tăng lên để lưu vào ô nhớ tiếp theo của mảng char
  + Nếu biến count = 16 thì em sẽ gọi tới chương trình con từ hợp ngữ là display\_memory() để xử lý và in ra 16 bit trên 1 dòng.
  + Nếu không đủ khi trong qua trình chạy vòng lặp mà biến count chua bằng 16 thì khi ra khỏi vòng lặp em sẽ gọi lại hàm display\_memory để xử lý các kí tự còn lại nếu chưa đủ 16 kí tự

## Các chương trình con

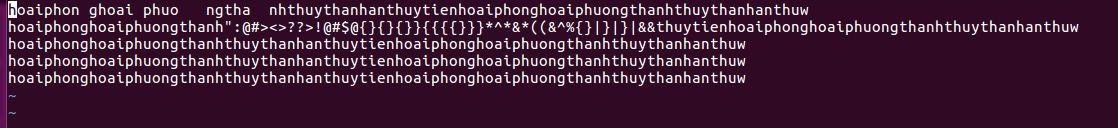
### Hàm lấy độ dài chuỗi

* **Mục đích của hàm:** Lý do em viết hàm này là vì đầu vào của một file txt là số lượng kí tự chuỗi không xác định nên em cần phải lấy ra độ dài của chuỗi. Mục đích của việc này là dùng để cấp phát malloc đúng bằng số lượng kí tự có trong file txt trừ kí tứ “\n”
* **Giá trị trả về:** Giá trị trả về của hàm này là một số nguyên kiểu long int có 4 byte.



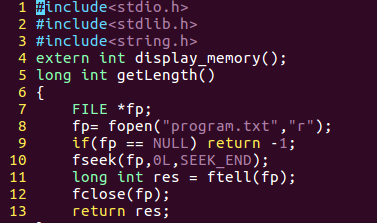
Hình 2‑2 Hình ảnh hàm lấy độ dài chuỗi

* **Dữ liệu đầu vào:** Hàm không có tham số do dữ liệu sẽ đọc từ file program.txt



Hình 2‑3 Hình ảnh của file program.txt

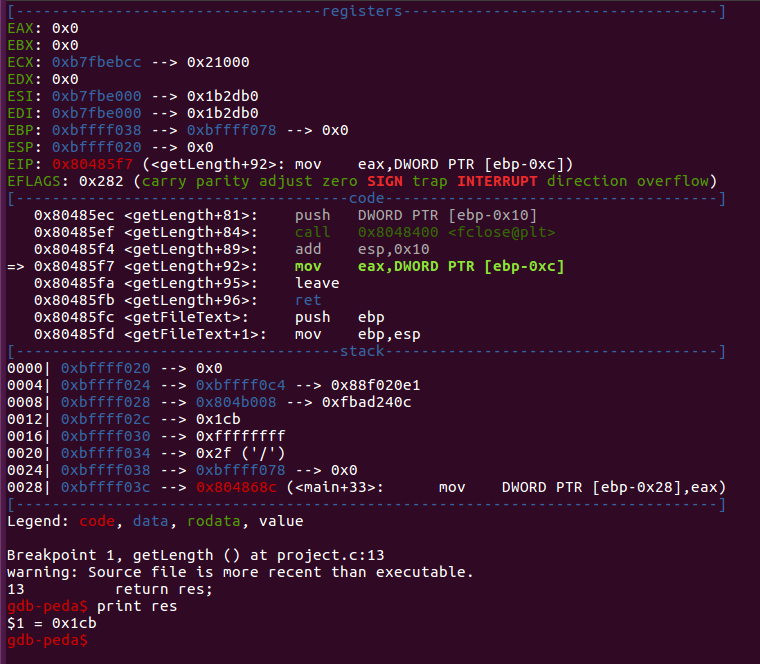
* **GDB để xem kết quả:** Chạy chương trình debug bằng gdb-peda để xem kết quả trong gdb. Ở đây em sẽ đặt debug tại dòng 13 để xem kết quả trả về của res



Hình 2‑4 Hình ảnh debug gdb-peda



Hình 2-5 Hình ảnh debug ở dòng 13

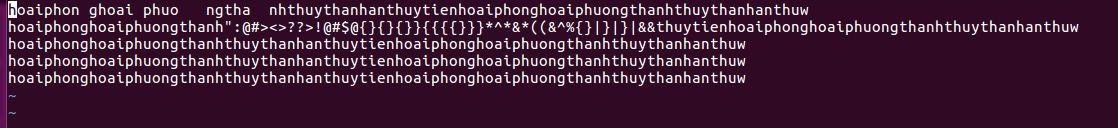


Hình 2‑6 Hình ảnh kết quả debug ở dòng 13

Kết quả debug ở dòng 13 sẽ ra chuỗi độ dài file text là 0x1cb do đã trừ đi 5 kí tự xuống dòng.

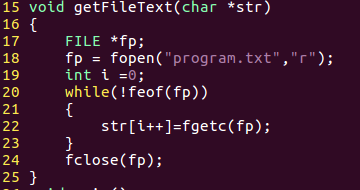
### Hàm đọc các kí tự có trong file text

* **Mục đích của hàm:** Hàm này có chức năng đọc từng kí tự từ file program.txt luu vào mảng con trỏ \*str được cấp phát malloc với độ dài từ hàm ở mục 2.2.1
* **Giá trị trả về:** Hàm này không có giá trị trả về
* **Dữ liệu đầu vào:** Tham số truyền vào là biến kiểu con trỏ \*str

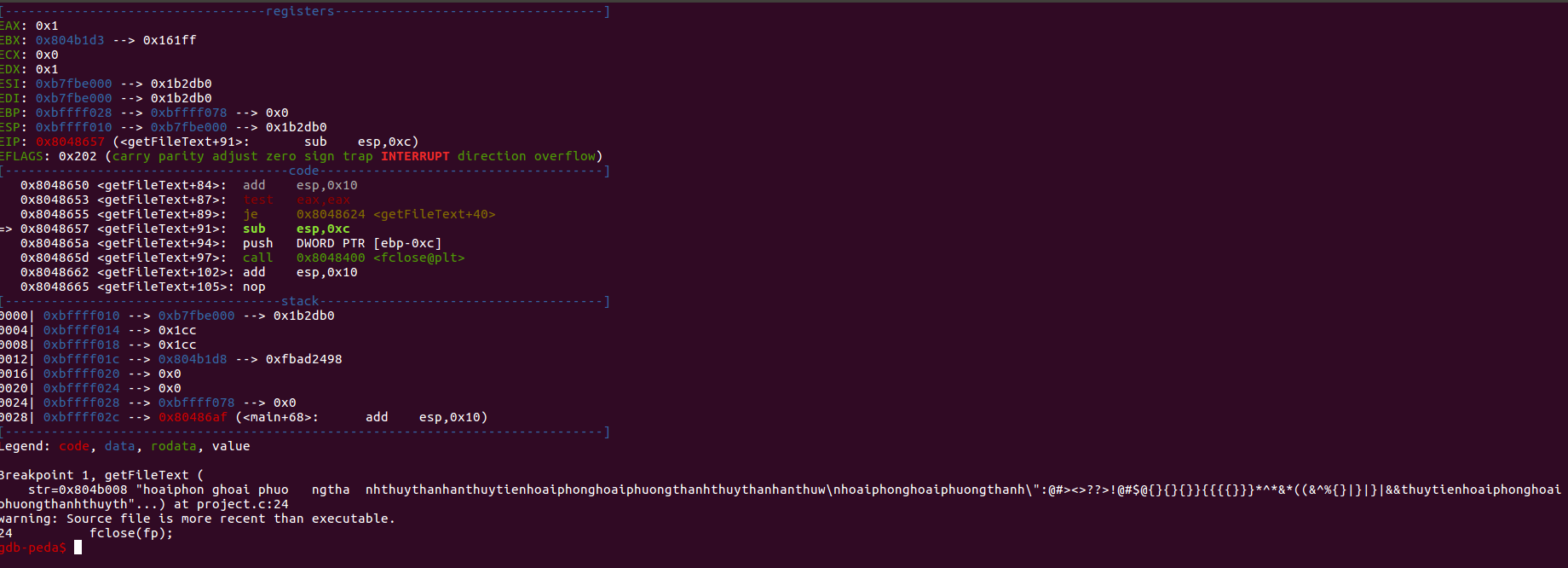


Hình 2‑7 Hình ảnh dữ liệu đầu và

* GDB để xem kết quả: Chạy gdb-peda để xem kết quả ở đây em sẽ đặt debug ở dòng 24 để xem kết của con trỏ str



Hình 2‑8 Hình ảnh của hàm getFileText

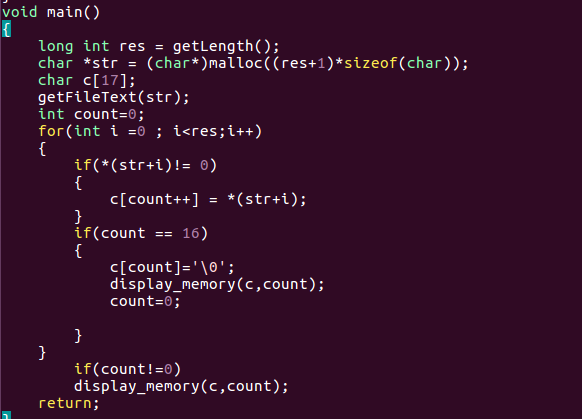


Hình 2‑9 Hình ảnh kết quả khi chạy debug của hàm lấy chuỗi

Kết quả sau khi lấy từng kí tự em sẽ lưu vào quản lý danh sách mảng bằng con trỏ với độ dài đúng bằng số lượng kí tự có trong mảng.

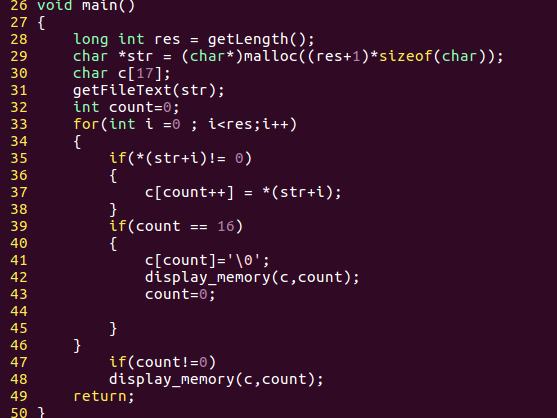
# Chương trình hợp ngữ

## Input



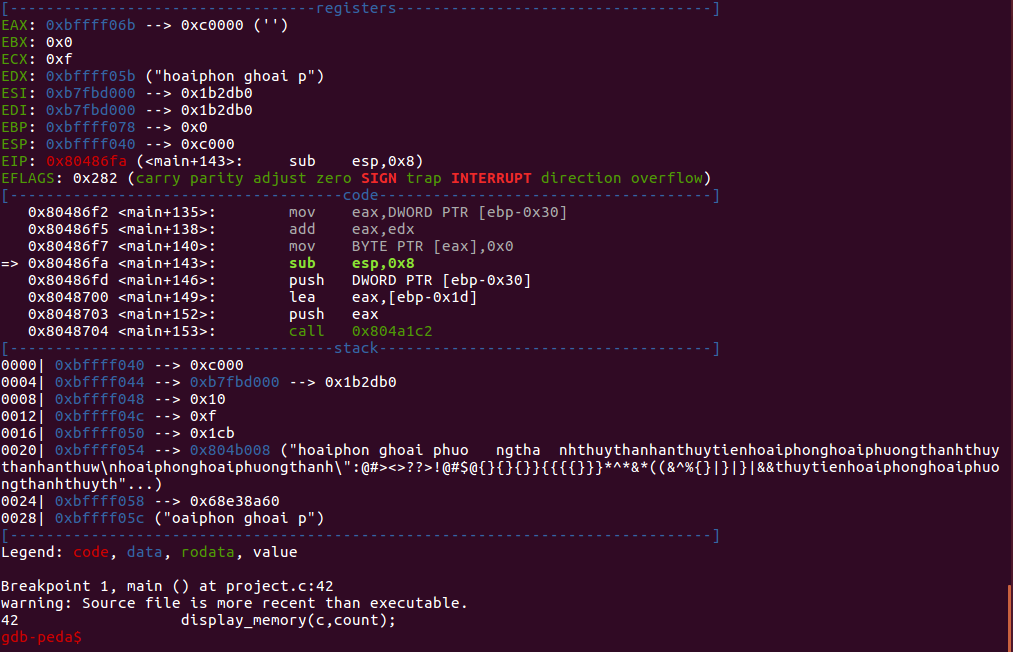
Hình 1.4.2‑1 Dữ liệu đầu vào của hàm

* **Ý tưởng về dữ liệu đầu vào:** Em sẽ chạy vòng lặp và đọc hết tất cả các kí tự đã lưu vào con trỏ str. Nếu biến count tăng đến 16 em sẽ gọi display\_memory() để in ra dòng đầu tiên và cứ tiếp tục vòng lặp như thế. Trường hợp không đủ 16 kí tự thì em sẽ gọi lại hàm display\_memory ngoài vòng lặp
* **Giải thích dữ liệu đầu vào:** Dữ liệu đầu vào display\_memory gồm 2 tham số là:
  + Biến count: là biến đếm để lưu từng kí tự vào mảng c đồng thời cũng là tham số để truyền vào hàm display\_memory nếu count 16 sẽ in ra trên 1 dòng. Còn nếu như count <> 16 thì sẽ xử lý riêng ở trường hợp ngoài vòng lặp
  + Mảng c: Dùng để lưu 16 kí tự sẽ được in ra trên 1 dòng nếu rơi vào trường hợp không đủ 16 kí tự thì sẽ xử lý trường hợp riêng ngoài vòng lặp
* **Chạy gdb giải thích cách lấy tham số:**



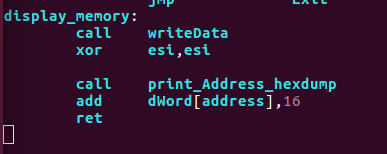
Hình 1.4.2‑2 Đặt breakpoint ở dòng 42

* Ở đây em sẽ chạy gdb và đặt breakpoint ở dòng 42 để và debug cách lấy ra các tham số đã truyền từ C vào hợp ngữ



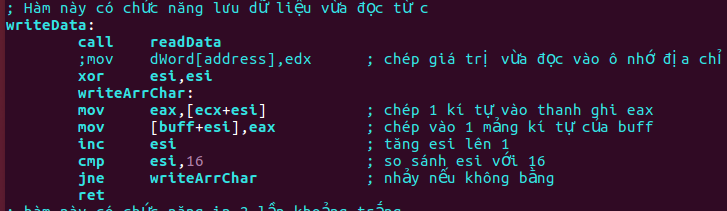
Hình 1.4.2‑3 Breakpoint ở dòng 42

* Lúc này em sẽ dùng câu lệnh step để vào chương trình con của display\_memory và chạy gdb giải thích cách em sẽ lưu 2 tham số vừa đươc truyền từ C như thế nào.



Hình 1.4.2‑4 Chương trình con trong display\_memory

## Hàm writeData



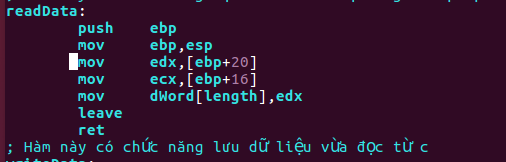
Hình 1.4.2‑1 Hàm writeData

* **Chức năng hàm writeData:** Mục đích của hàm write data là em sẽ lưu 16 kí tự vừa đọc từ C vào mảng buff được khai báo với 16 bit.



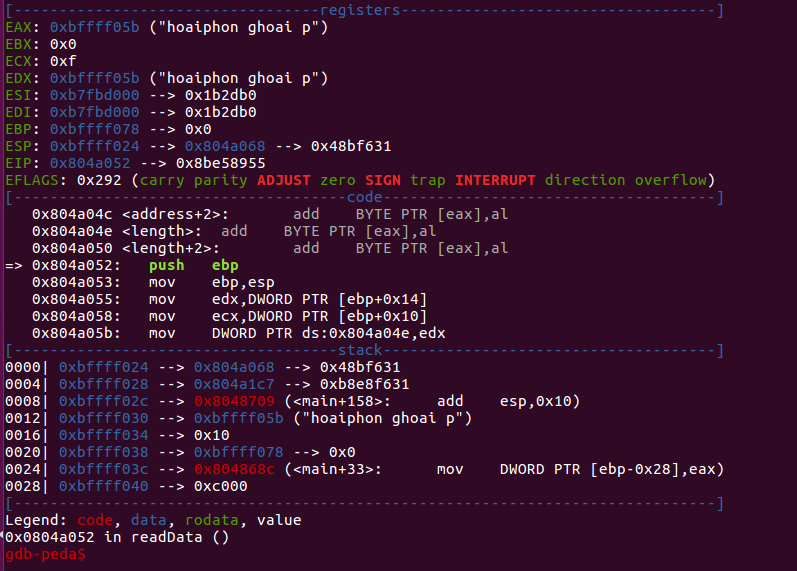
Hình 1.4.2‑2 Khai báo mảng buff

### Hàm readData



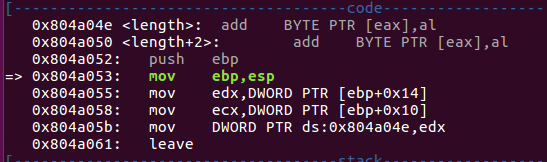
Hình 2.2.1‑1 Hàm readData

* **Chức năng của hàm readData**: Hàm readData sẽ lấy dữ liệu từ C truyền vào và chuyển vào cho thanh ghi edx,ecx
* **Chạy debug và giải thích**:



Hình 2.2.1‑2 Debug hàm readData

* Đầu tiên em sẽ cất giá trị thanh ghi ebp
  + Vì ban đầu ebp đang trỏ tới stack frame trước ở hàng này nên em phải có động tác cất thanh ghi ebp bằng câu lệnh push

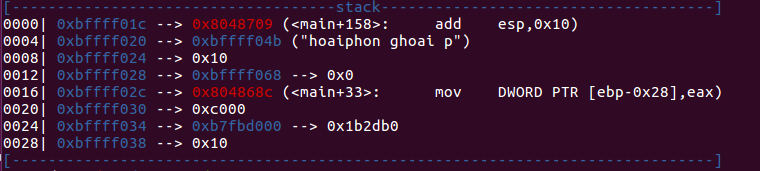


Hình 2.2.1‑3 Copy esp vào ebp

* Copy esp vào thanh ghi ebp để cùng trỏ tới 1 vùng nhớ

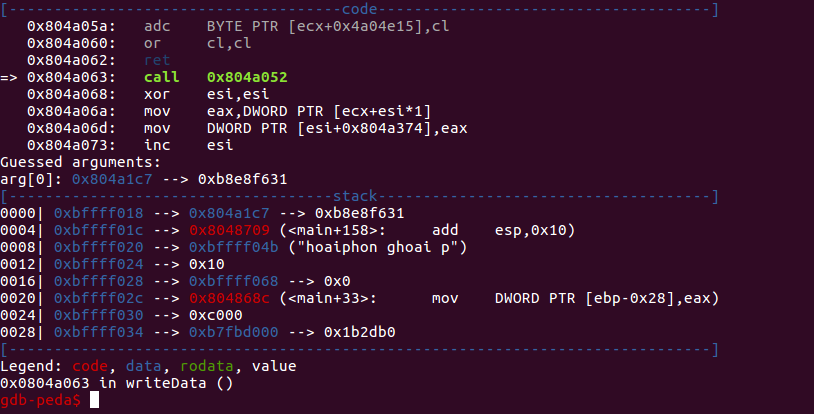
Giải thích tại sao ebp + 0x14:

* + Trong stack frame: thì hàm return address chiếm 4 byte



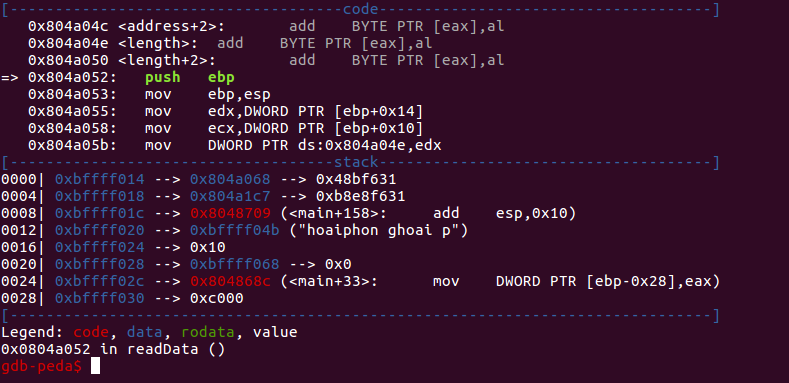
Hình 2.2.1‑4 Stack frame của 2 tham số

* + 2 tham số được buff vào là chuỗi kí tự và độ dài chuỗi: tổng 2 tham số này được buff vào trong stack là 8 byte

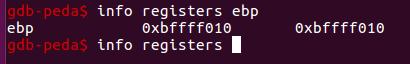


Hình 2.2.1‑5 Gọi Chương trình con writeData

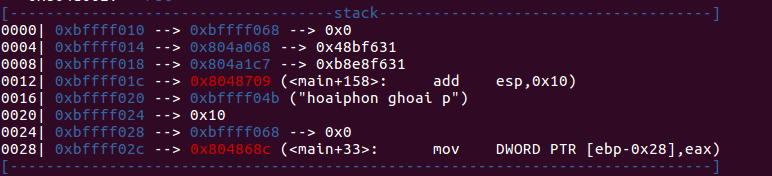
* + Khị gọi tới chương trình con writeData thì trong stack frame phải lưu thêm 4 byte để luu đia chỉ trả v

Hình 2.2.1‑6 Hình ảnh gọi chương trình con readData

* + Trong writeData lại gọi tới readData lúc này phải cần thêm 4byte để chưa đại chỉ trả về



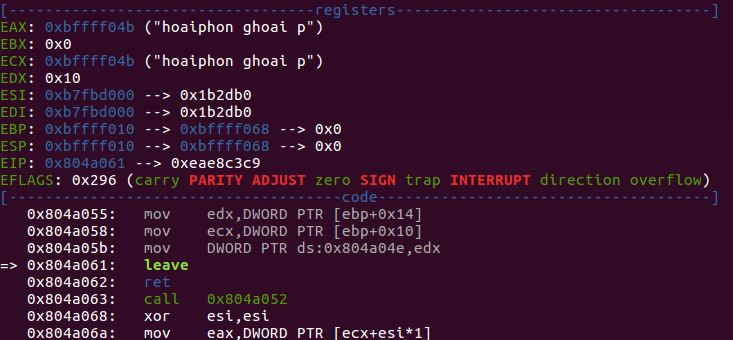
Hình 2.2.1‑7 Giá trị thanh ghi ebp



Hình 2.2.1‑8 Gía trị trong stack

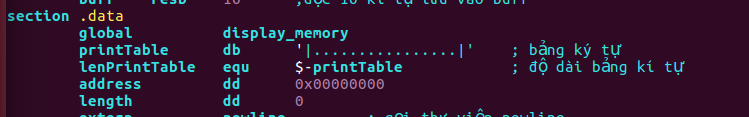
* Vì là có thêm 2 đia chỉ trả về thì lúc này địa chỉ của chuỗi và độ dài chuỗi cần lấy phải cộng cho +16 và +20
  + Vậy để lấy được giá trị độ dài chuỗi thì ở thanh ghi ebp + 20 = 0xbffff010 + 0x14 = 0xbffff24 là độ dài chuỗi
  + Vậy để lấy được độ địa chỉ chuỗi thì ở thanh ghi ebp + 16 = 0xbffff010 + 0x10 = 0xbffff20 là địa chỉ chuỗi.

Ở đây em sẽ dùng thanh ghi edx để giữ địa chỉ chuỗi và thanh ghi ecx để giữ độ dài chuỗi.

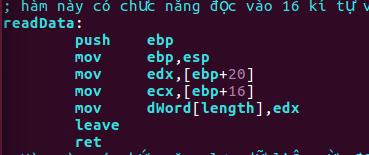


Hình 2.2.1‑9 Gía trị của thanh ghi edx và ecx

* Sau đó em sẽ dùng biến length để giữ giá trị độ dài chuỗi từ thanh ghi edx và mảng buff giữ các kí tự trong thanh ghi ecx

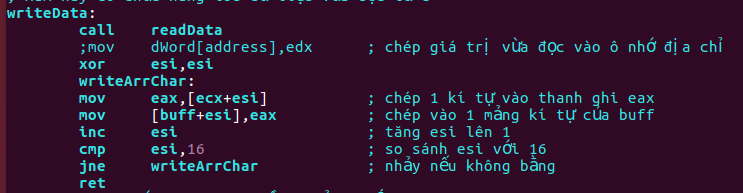


Hình 2.2.1‑10 Khai báo cho biến length



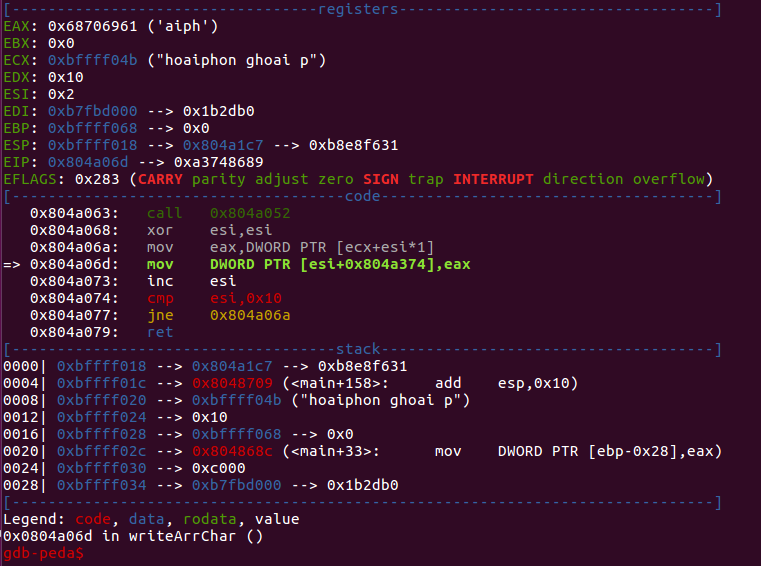
Hình 2.2.1‑11 Hình ảnh cho biến length

* Ở hình 2.4.1-15 khi chạy tới cậu lệnh leave thì vi xử lý sẽ tư động thực hiện câu lệnh mov esp,ebp và pop ebp để esp trỏ về địa chỉ trả về của hàm đã gọi trước đó.

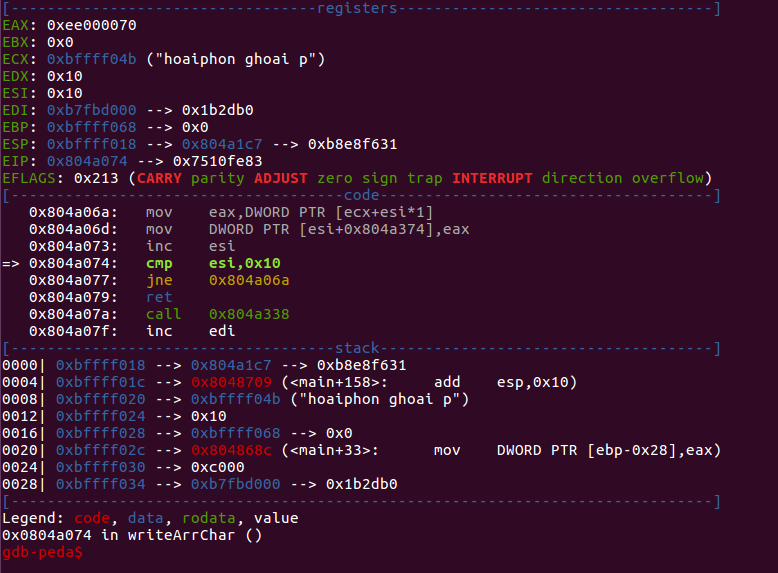


Hình 2.2.1‑12 Giải thích hàm writeData

* Sau khi thực hiện xong hàm read data thì lúc này em sẽ chép các kí tự và thanh ghi ecx đã lưu địa chị chuỗi vào buff
  + Ở đây em dùng biến đếm là esi để đưa từng kí tự vào mảng buff đã khai báo
* Chạy debug và xem kết quả của mảng buff và length

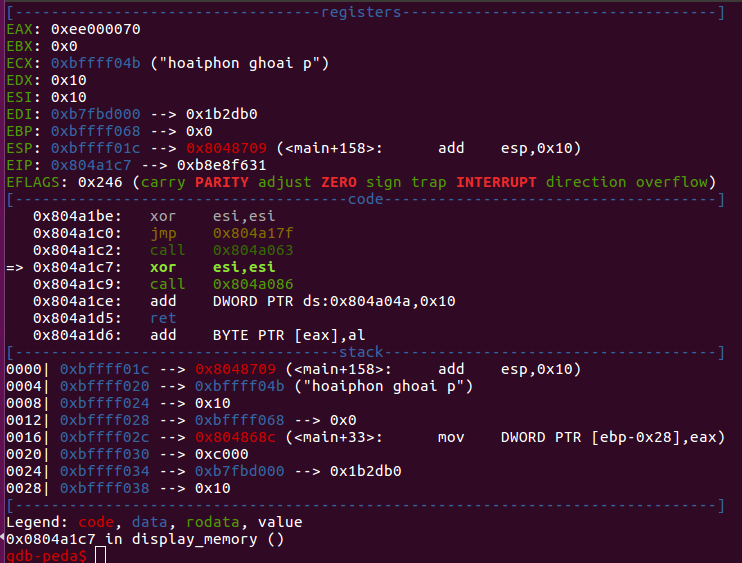


Hình 2.2.1‑13 Qúa trình lặp để đưa vào từng kí tự

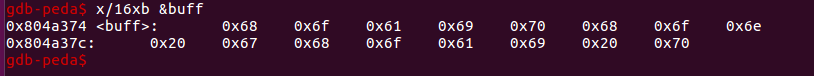


Hình 2.2.1‑14 Khi thanh ghi esi bằng 16

* Khi thanh ghi esi bằng 16 lúc này em đã lưu 16 kí tự vào buff.



Hình 2.2.1‑15 Chương trình quay lại hàm display\_memory



Hình 2.2.1‑16 Kết quả của mảng buff sau khi chép 16 kí tự



Hình 2.2.1‑17 Kết quả của biến length

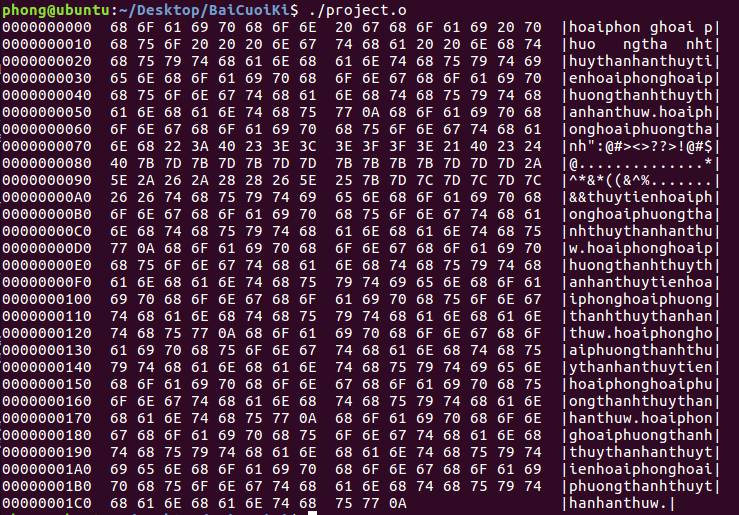
* Đây là kết quả sau khi em đọc 16 kí tự và độ dài chuỗi từ C vào hợp ngữ lưu vào thanh ghi edx với địa chỉ chuỗi và độ dài chuỗi với thanh ghi ecx sau đó lưu vào mảng buff và biến length.

## Hàm print\_Address\_Hexdump

* **Chức năng của hàm:** Hàm này sẽ in ra địa hexdump và kí tự hex cùng với mã ascii của 16 kí tự. Em sẽ chạy chương trình và xem kết quả của hàm này



Hình 2.2.1‑1 Kết quả được in trên 1 dòng

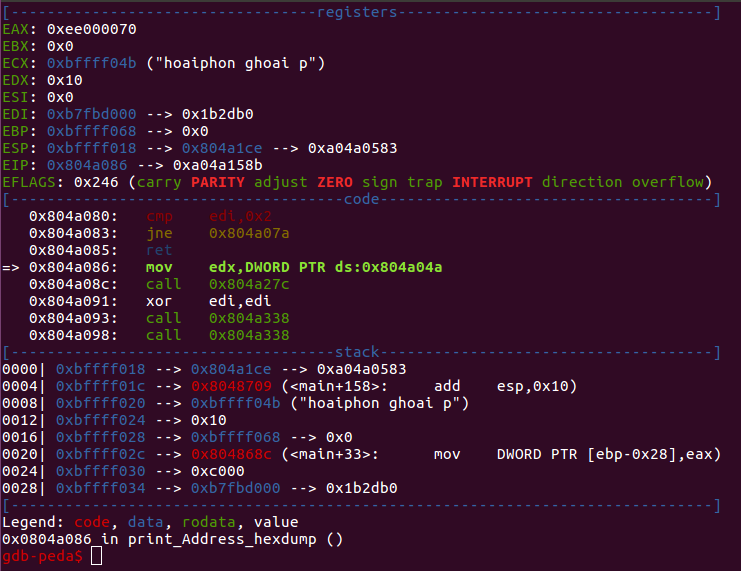


Hình 2.6‑2 Kết quả được in trên n dòng

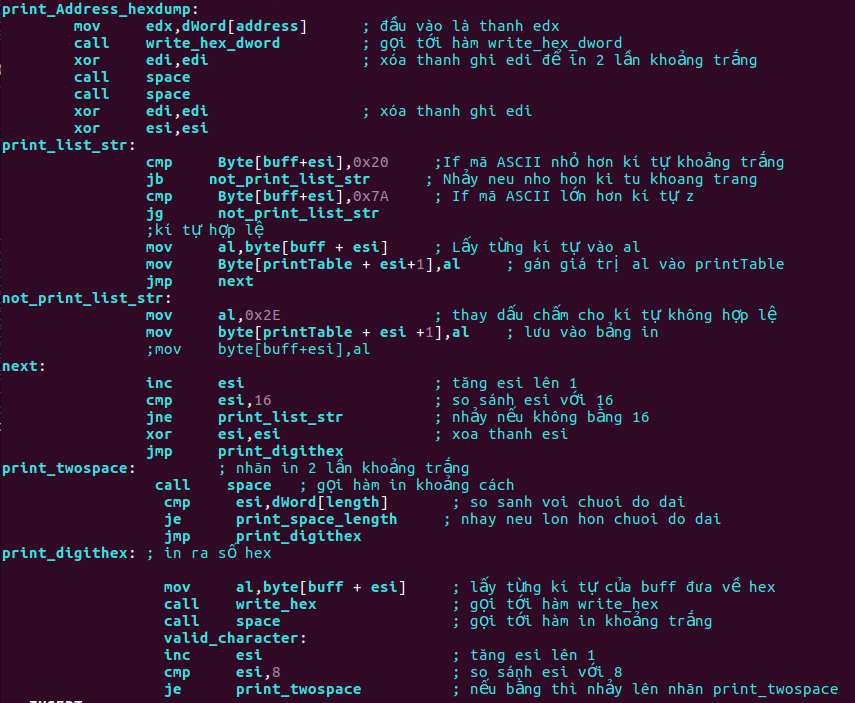
* Kết quả này sẽ phụ thuộc vào vòng for được lặp từ chương trình C.

**Giải thích sơ lược cách in của hàm này:**

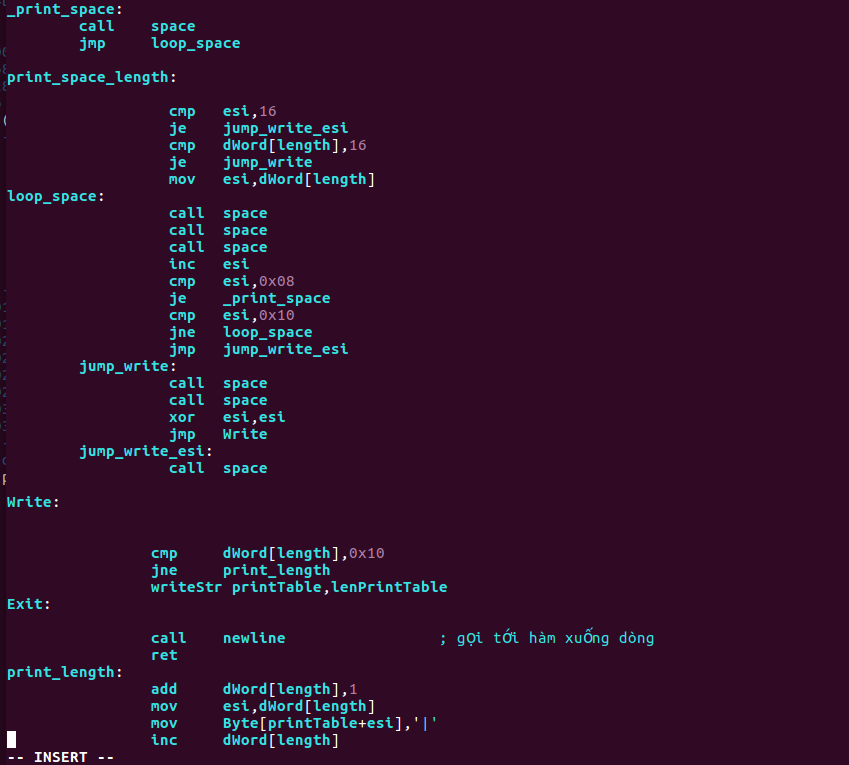
* + Đầu tiên em sẽ in giá trị địa chỉ hexdump có 32 bit sau đó in tiếp 2 khoảng trắng
  + Tiếp tục em sẽ in từng kí tự trong mảng buff tương đương với giá trị hex của kí tự đó. Cứ 8 và 16 kí tự em sẽ in 2 khoảng trắng
  + Sau đó tiếp tục in ra 16 kí tự và kết thúc bằng kí tự xuống dòng
  + Quá trình này sẽ tiếp tục diễn ra đến khi hết vòng for và nếu còn có kí tự nào không đủ 16 kí tự thì sẽ in ra hết phần còn lại.
* **Chạy gdb để giải thích cách hoat động của hàm này:**



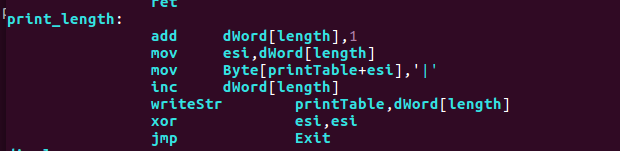
Hình 2.2.1‑2 Chạy gdb hàm print\_Adress\_hexdump



Hình 2.2.1‑3 Hàm print\_Address\_hexdump



Hình 2.2.1‑4 Hàm print\_Address\_Hexdump

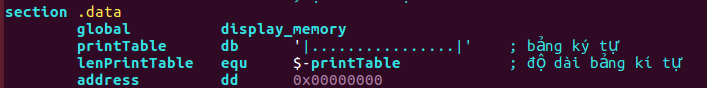


Hình 2.2.1‑5 Hàm print\_Address\_Hexdump

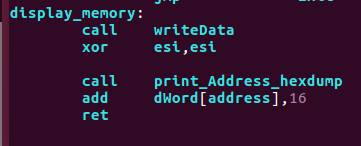
* Đây là tất cả các chương trình em viết vào trong hàm print\_Address\_Hexdump em sẽ debug hàm này và in ra kết quả trên 1 dòng bằng gdb-peda

### Hàm write\_hex\_dword

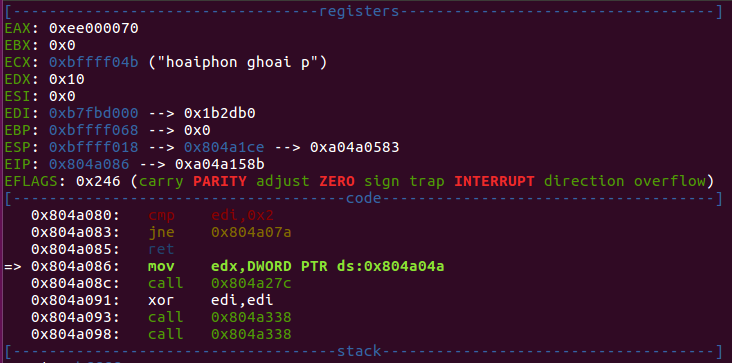
* **Chức năng của hàm này:** Bởi vì khi in trên 1 dòng việc đầu tiên em làm sẽ in ra địa chỉ hexdump trước. Nên ghi vào hàm print\_Address\_Hexdump em sẽ gọi tới hàm này để in ra địa chỉ 32bit của hexdump
* **Dữ liệu đầu vào của hàm:** Dữ liệu đầu vào của hàm em sẽ đưa giả trị address được khai báo là 0x00000000 đây là địa chỉ đầu của hexdump vào cho thanh ghi edx. Vì sau khi in ra trên 1 dòng biến address + 16 lên địa chỉ tiếp theo nếu có.



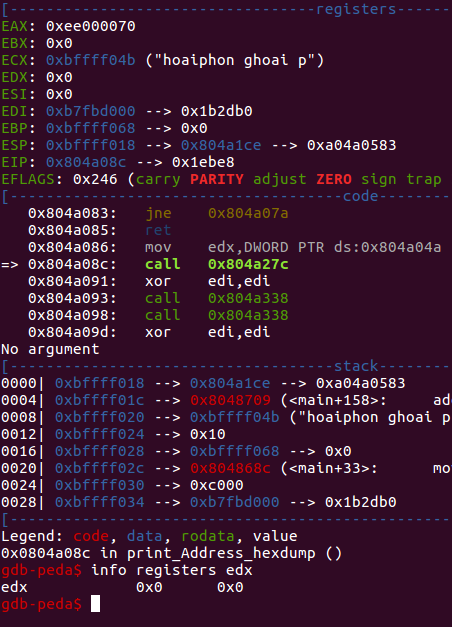
Hình 2.3.1‑1 Khởi tạo biến address



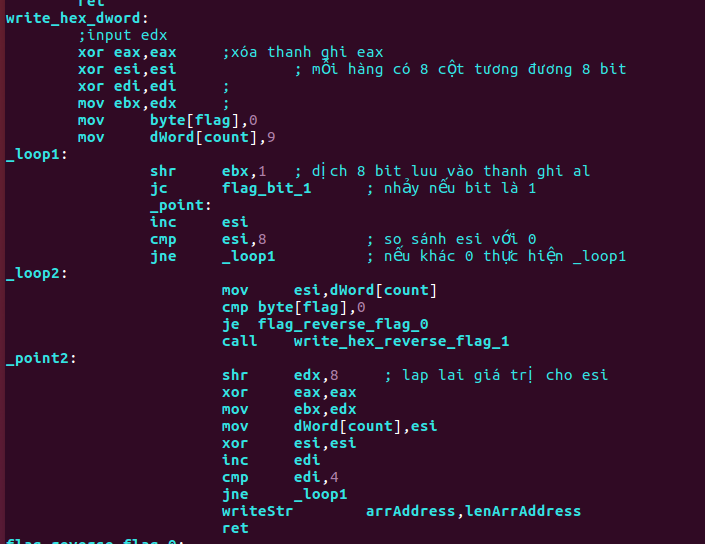
Hình 2.3.1‑2 Tăng địa chỉ address sau mỗi lần in



Hình 2.3.1‑3 Gán giá trị address cho thanh ghi edx



Hình 2.3.1‑4 Xem giá trị thanh ghi edx

* Vì là lần in đầu tiên trên giá trị ở edx là 1
* **Hàm write\_hex\_dword**: Hàm này được em viết trong thư viện display.asm

Hình 2.3.1‑5 Hình ảnh hàm write\_hex\_dword ở display.asm

* **Ý tưởng về write\_hex\_dword:** Bởi vì địa chỉ in ra địa chỉ của hexdump thì cần 32bit tương ứng với kiểu dữ liệu dword.
  + Ý tưởng của em sẽ shr 8 bit và lưu vào thanh ghi ax để in ra kí tự hex mà vì là 32 bit nên sẽ có 4 lần dịch bit mỗi lần 8 bit.
  + Kết quả của từng kí tự hex có được em sẽ lưu vào arrAdress được khai báo ở display.asm

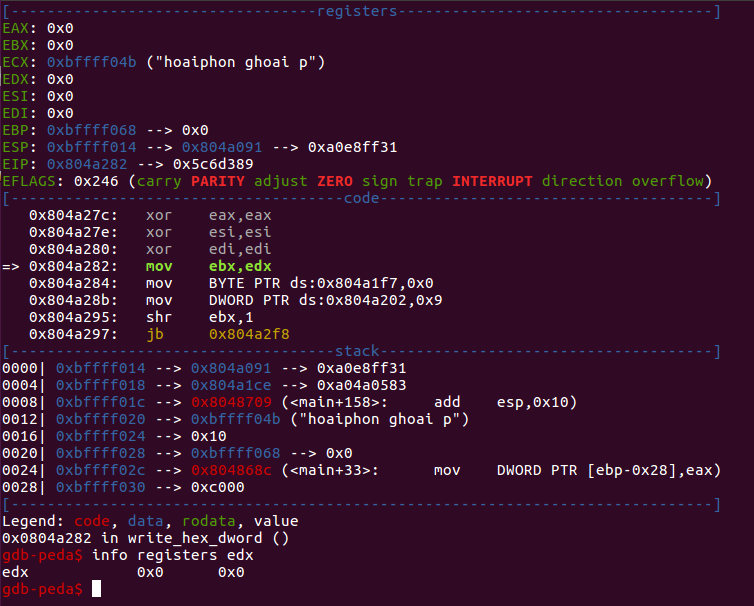


Hình 2.3.1‑6 Khai báo chuỗi kí tự arrAddress

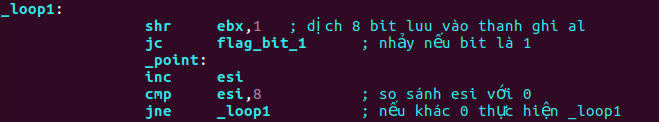
* **Chạy debug và giải thích code:**



Hình 2.3.1‑7 Chuẩn bị cho hàm write\_hex\_dword

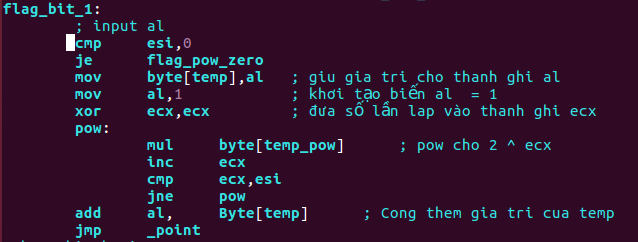


Hình 2.3.1‑8 Xem kết quả edx tại hàm write\_hex\_dwordỞ hình 2.6.1-7 em sẽ xóa 3 thanh ghi eax,esi,edi và đưa giá trị của thanh ghi edx cho ebx



Hình 2.3.1‑9 Nhãn loop\_1

* Ở loop 1 này em sẽ dịch 1 bit sang phải nếu bit bị dịch là 1 thì cờ CF =1 thì lúc này em sẽ nhãy xuống nhãn flag\_bit\_1

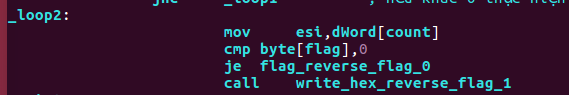


Hình 2.3.1‑10 Nhãn flag\_bit\_1

* Nhãn này có có nhiệm vụ là sẽ tính 2^n nếu bit được dịch là bit 1 và lưu vào biến temp
  + Đầu tiên em sẽ cho biến temp giữ giá trị al trường hợp xảy ra esi = 0

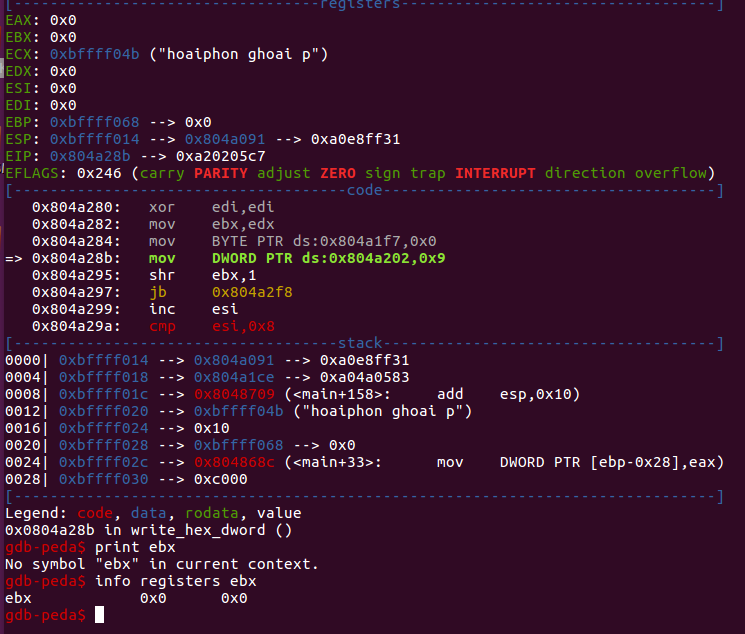


* + Tạo 1 biến temp kiểu byte để giữ giá trị cho thanh ghi al, vì trong quá trình dịch 8 bit có thể se có nhiều bit 1 và cờ CF được bật nhiều lần
  + Do khi dịch thanh ghi ebx khi bit được dịch là 1 thì em sẽ tính giá trị tại bit 1 đó và lặp theo esi và thanh ghi al sẽ có giá trị al = 2^esi
  + Đầu tiên em phải gán giá trị cho thanh ghi al là 1 để cứ mỗi lần lặp thì sẽ nhân với 2
  + Sau đó cộng với giá trị ban đầu được dịch nếu nhiều bit 1



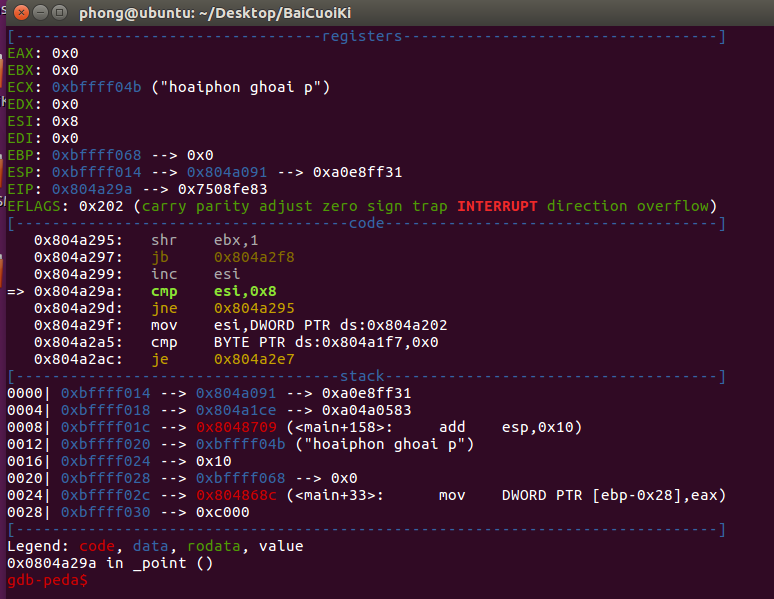
Hình 2.3.1‑11 Nhãn \_loop2

* Ở nhãn này em sẽ in ra giá trị địa chỉ hexdump về dạng hex theo biến cờ flag
* Ví dụ: khi lưu giá trị vào mảng địa chỉ arrAdress thì lúc này em sẽ lưu từ cuối mảng cho tới đầu mảng.



Hình 2.3.1‑12 Debug write\_hex\_dword

* Em sẽ đưa giá trị edx vào ebx và dịch 8 bit theo số lần lặp của thanh ghi esi



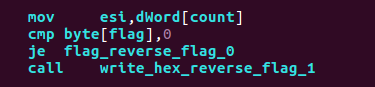
Hình 2.3.1‑13 Debug write\_hex\_dword

* Vì là 32 bit nên phải dịch 4 lần và 1 lần dịch là 8 bit qua trình loop:1 xảy ra đến khi thanh ghi esi bằng 8

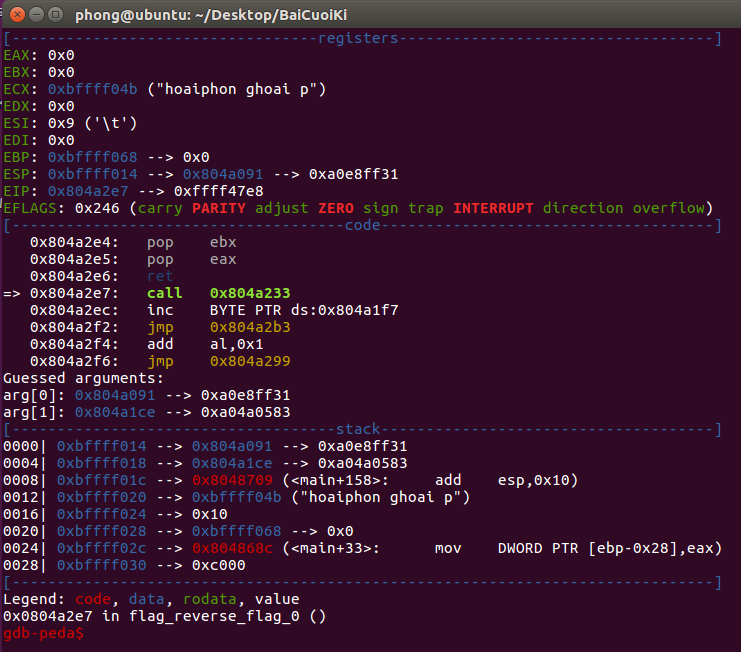


Hình 2.3.1‑14 Debug write\_hex\_dword

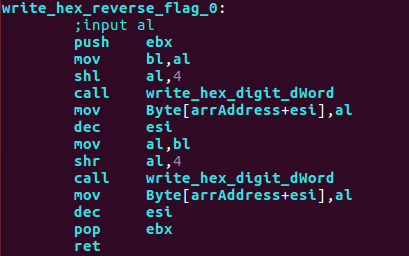
* Vì là em sẽ lưu kí tự hex từ cuối chuỗi arrAddress nên em sẽ gán cho biến count = 9 vì arrAdress có 10 ô nhớ
* Lúc này vì là lần lưu kí tự hex đầu tiên nên em sẽ đặt biến flag = 0 để lưu được giá trị theo dạng hex với 32 bit



Hình 2.3.1‑15 Debug write\_hex\_dword

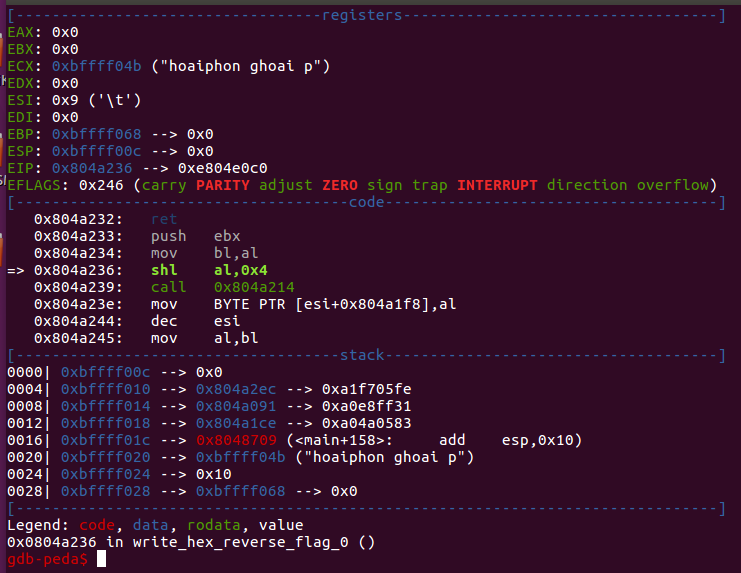


Hình 2.3.1‑16 Debug write\_hex\_dword

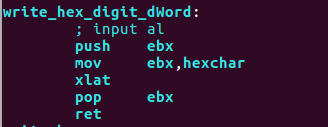


Hình 2.3.1‑17 Hàm write\_hex\_reverse\_flag\_0

* Hàm này sẽ nhận đầu vào là thanh ghi al lưu vào chuỗi arrAdress
  + Ban đầu em sẽ xoay trái 4 bit. Mục đích của việc này là em sẽ lưu giá trị kiểu hex ở 4 bit cao



Hình 2.3.1‑18 Hàm write\_hex\_reverse\_flag\_0

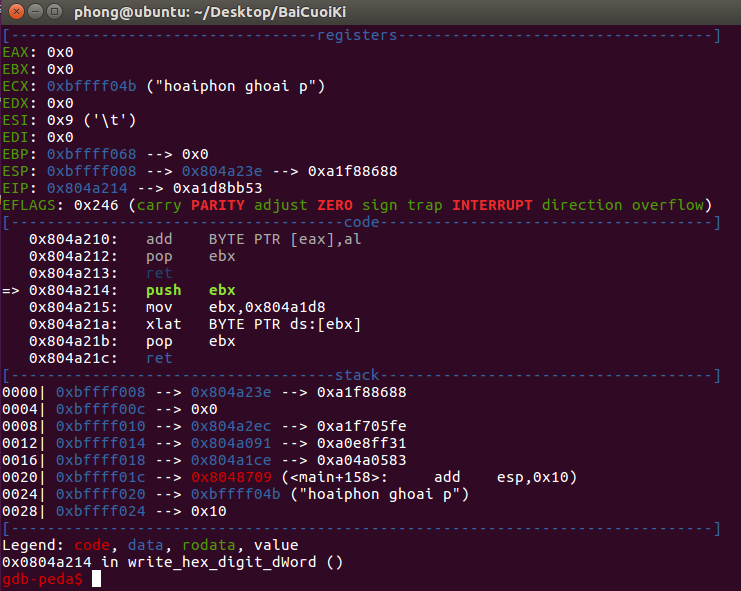


Hình 2.3.1‑19 Hàm write\_hex\_reverse\_flag\_0

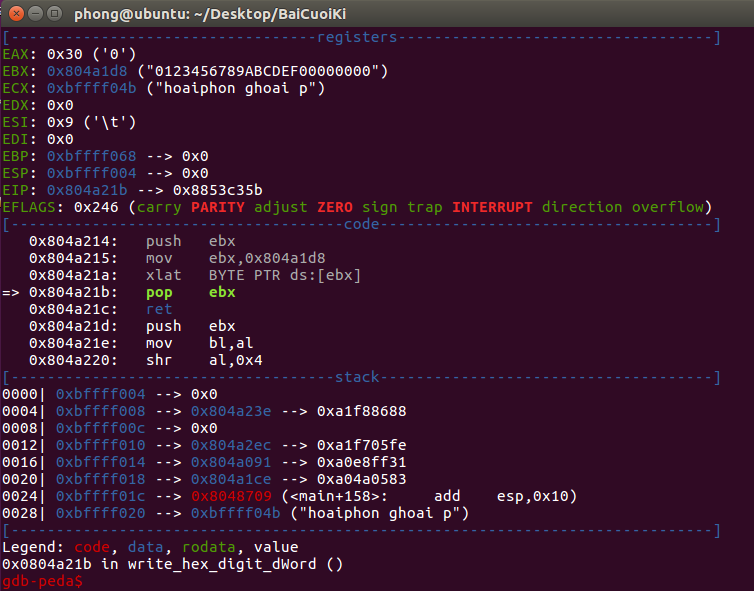
* Mục địch cùa hàm này sẽ tra kí tự hex trong chuỗi hexchar đã được khai báo bằng xlat



Hình 2.3.1‑20 Hàm write\_hex\_reverse\_flag\_0

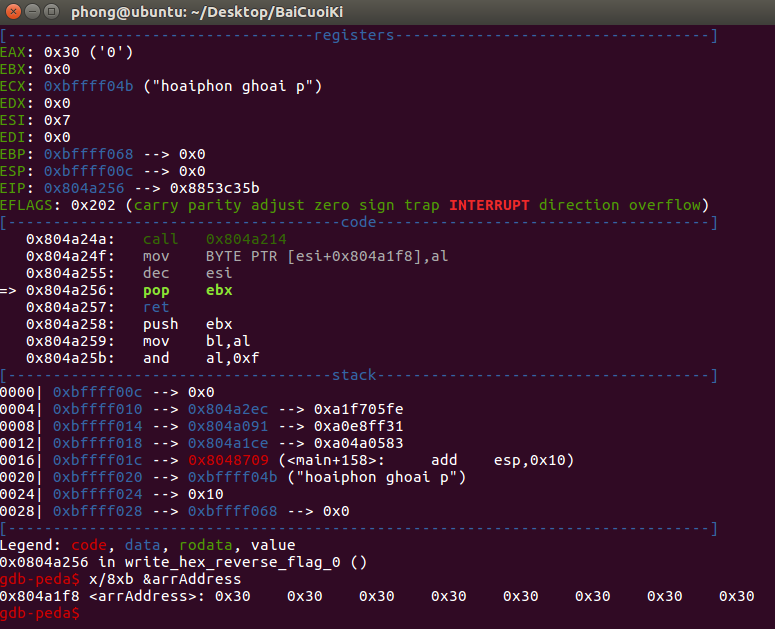


Hình 2.3.1‑21 Hàm write\_hex\_digit\_dWord



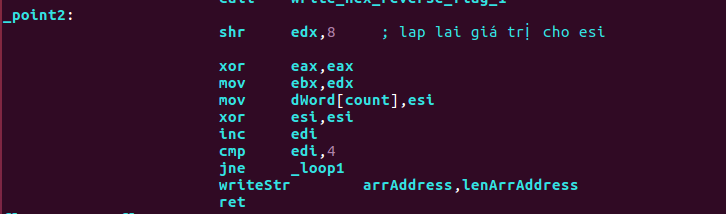
Hình 2.3.1‑22 Hàm write\_hex\_digit\_dWord

Từ hình 2.6.1-22 lúc này thanh ghi EAX được gán giá trị là 0x30 sau khi tra bảng hexchar.

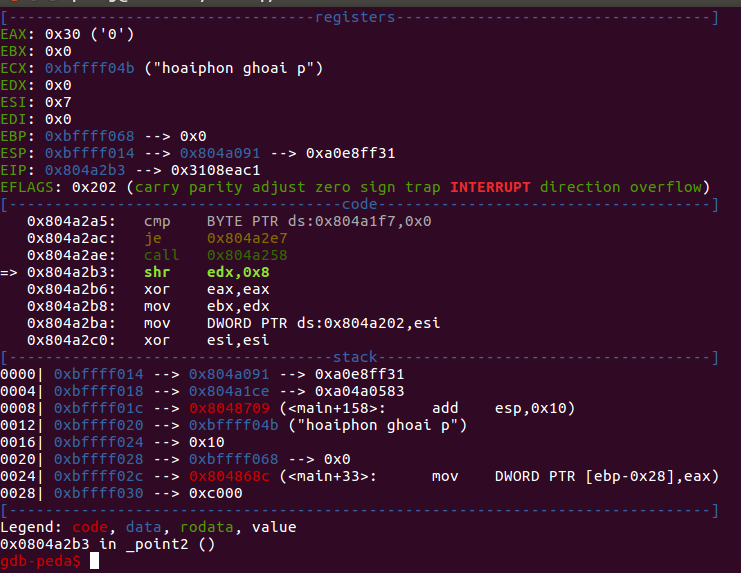


Hình 2.3.1‑23 Hàm write\_hex\_dword

* Sau khi chạy trong xòm write\_hex\_dword thì 2 byte cuối trong chuỗi kí tự của mảng arrAdress sẽ gán là kí tự 0 do địa chỉ ban đầu truyền vào thanh ghi edx là 00000000

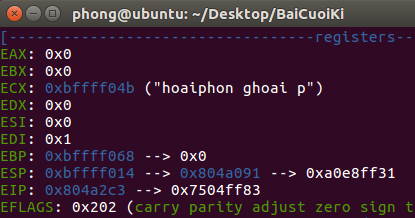


Hình 2.3.1‑24 Hàm write\_hex\_dword



Hình 2.3.1‑25 Hàm write\_hex\_dword

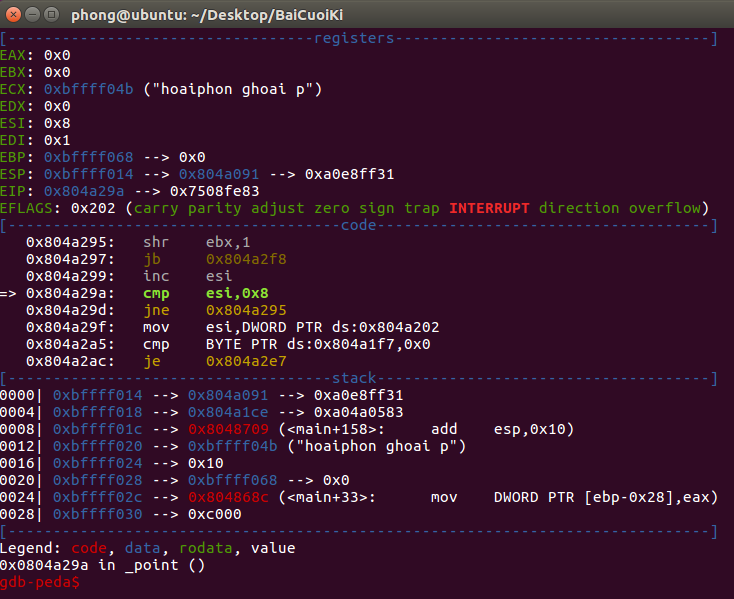
* Ở nhãn \_point2 như trên hình 2.6.1-24: em sẽ dịch phải 8 bit giá trị địa chỉ của hexdump có 32 bit với mỗi giá trị hex mang 8 bit thì phải dịch tối đa là 4
  + Cứ mỗi lần dịch như thế em lại tăng thanh ghi edi lên 1

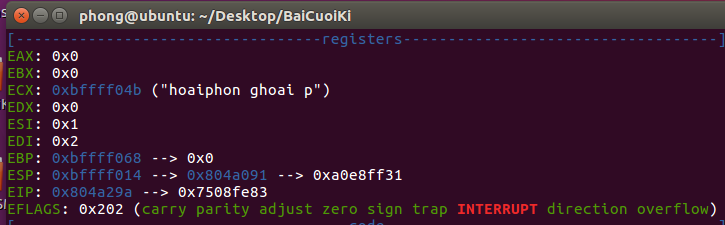


Hình 2.3.1‑26 Hàm write\_hex\_dword

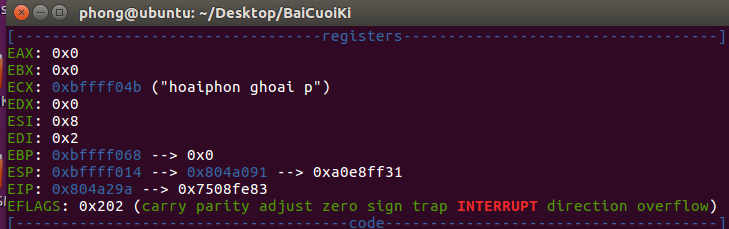
Đưa thanh ghi esi về 0 và cứ tiếp tục quá trình như vậy đến khi thanh ghi edi bằng 0x4 thì lúc này sẽ đủ 32 bit cho giá trị địa chỉ của hexdump

* Hình ảnh chạy debug GDB cho 4 quá trình tiếp theo

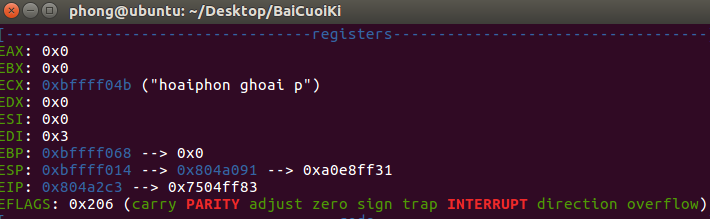


Hình 2.3.1‑27 Hàm write\_hex\_dword 

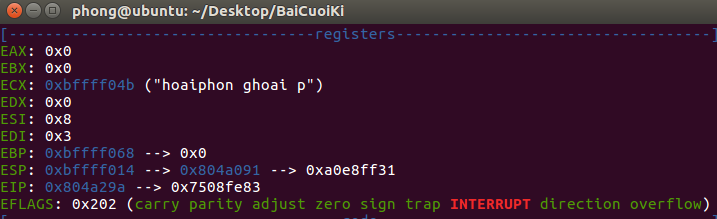
Hình 2.3.1‑28 Hàm write\_hex\_dword



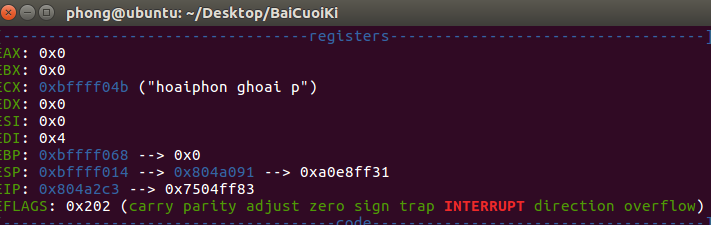
Hình 2.3.1‑29 Hàm write\_hex\_dword



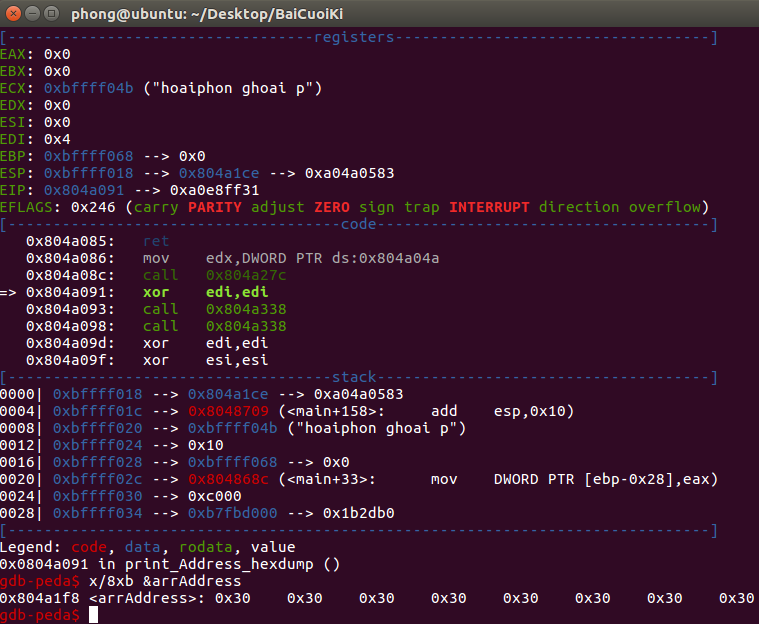
Hình 2.3.1‑30 Hàm write\_hex\_dword



Hình 2.3.1‑31 Hàm write\_hex\_dword



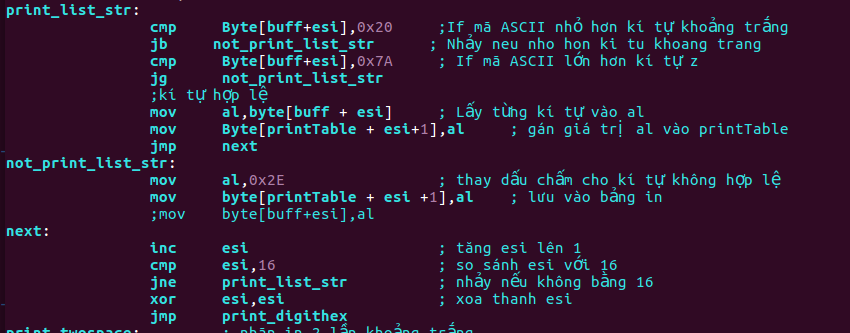
Hình 2.3.1‑32 Hàm write\_hex\_dword



Hình 2.3.1‑33 Kết quả sau khi ra khỏi hàm write\_hex\_dword

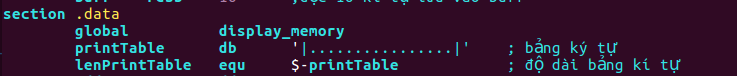
* Kết quả sau khi chạy xong hàm write\_hex\_dword khi truyền giá trị địa chỉ 32 bit là 0x00000000

### Chép 16 kí tự vào printTable



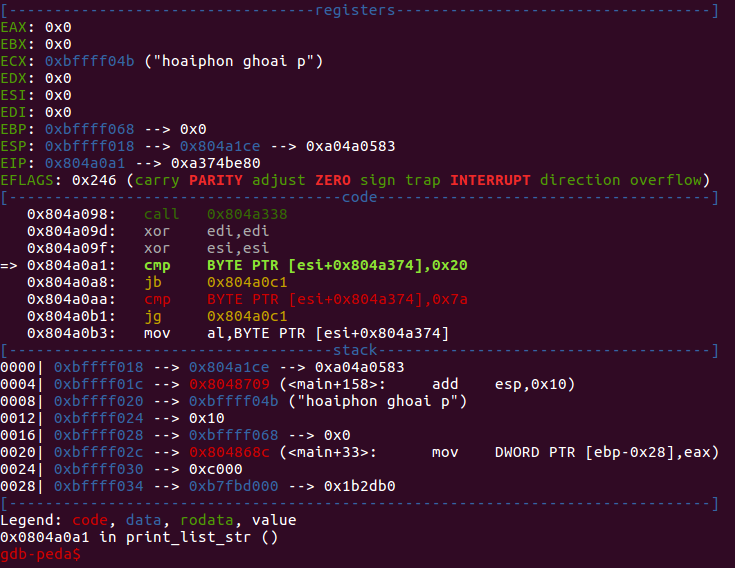
Hình 2.3.2‑1 In 16 bit kí tự về dạng hex

* **Sơ lược đoạn code trở hình 2.6.2-1**: Đầu tiên em sẽ xác định các kí tự hợp lệ và không hợp lệ.
  + Kí tự không hợp lệ được em quy định là nhỏ hơn cách tự khoảng cách hoặc lớn hơn kí tự z
  + Em sẽ xử lý các kí tự không hợp lệ đó bằng cách thay nó bằng dấu chấm để tránh bị xô lệch trong qua trình in
  + 16 kí tự được kiểm tra em sẽ lưu trong mảng đã được khao báo sẵn printable



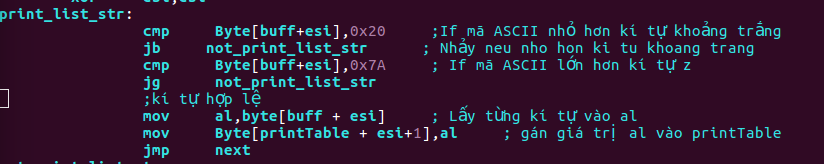
Hình 2.3.2‑2 In 16 bit kí tự về dạng hex

* + Tất cả 16 kí tự hợp lệ và không hợp lệ em sẽ lưu trong mảng printable dùng để in kết quả cột cuối cùng
* **Chạy debug bằng gdb và kiểm chứng:**

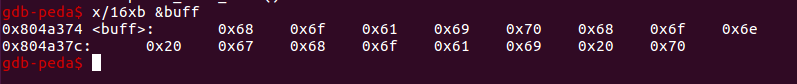


Hình 2.3.2‑3 In 16 bit kí tự

* Trên thanh ghi em dùng thanh ghi ESI làm biến đếm để kiểm tra từng kí tự và mảng buff lưu 16.
  + Kiểm tra kí tự có lớn ‘ ’ và nhỏ hon ‘z’ hay không. Nếu nằm ngoải khoảng này kí tự đó sẽ thay bằng ‘.’

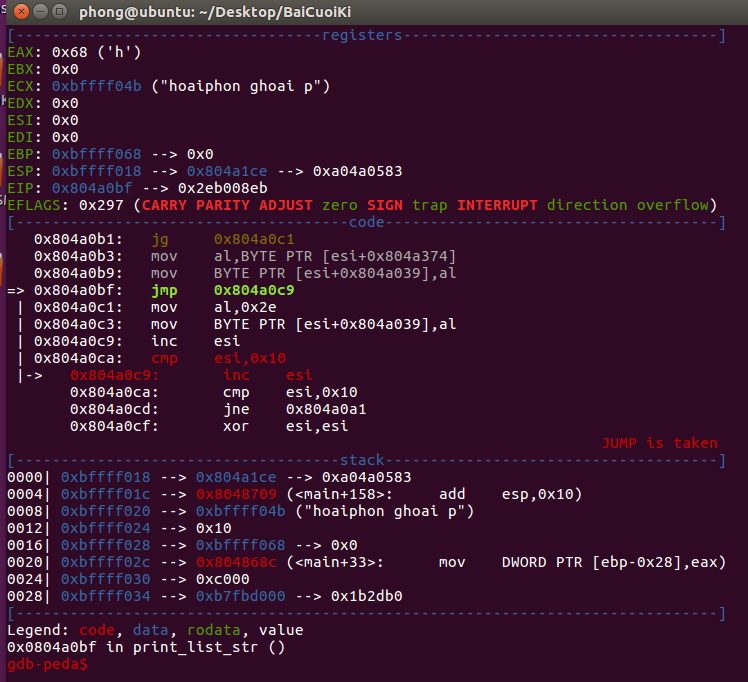


Hình 2.3.2‑4 In 16 bit kí tự

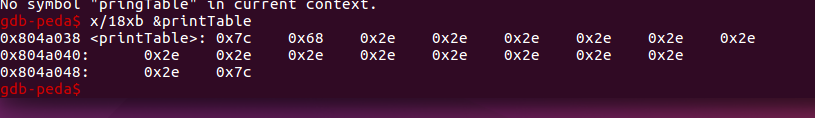


Hình 2.3.2‑5 In 16 bit kí tự

* + Do kí tự đầu nằm trong khoảng được cho phép là kí hợp lệ nên sẽ được lưu vào printable

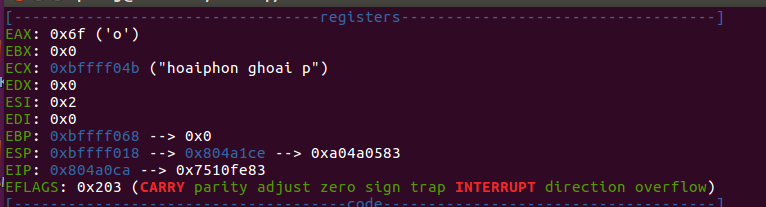


Hình 2.3.2‑6 In 16 bit ki tự



Hình 2.3.2‑7 In 16 bit kí tự

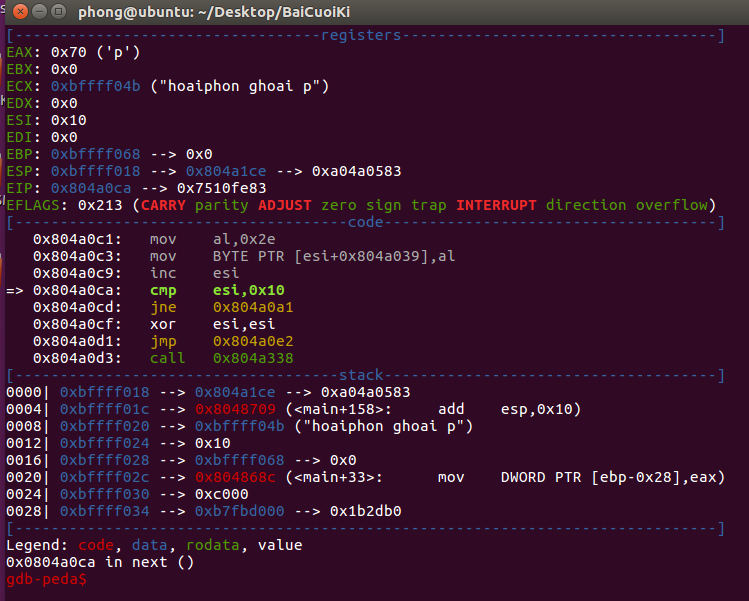
* + 0x7c là kí tụ ‘|’ nên em không cần thay thế nó em chỉ thay thế các kí tự từ esi+1 lúc này dấu chấm đã được đổi thành 0x68 mang kí tự ‘h’



Hình 2.3.2‑8 In 16 bit kí tự



Hình 2.3.2‑9 In 16 bit kí tự



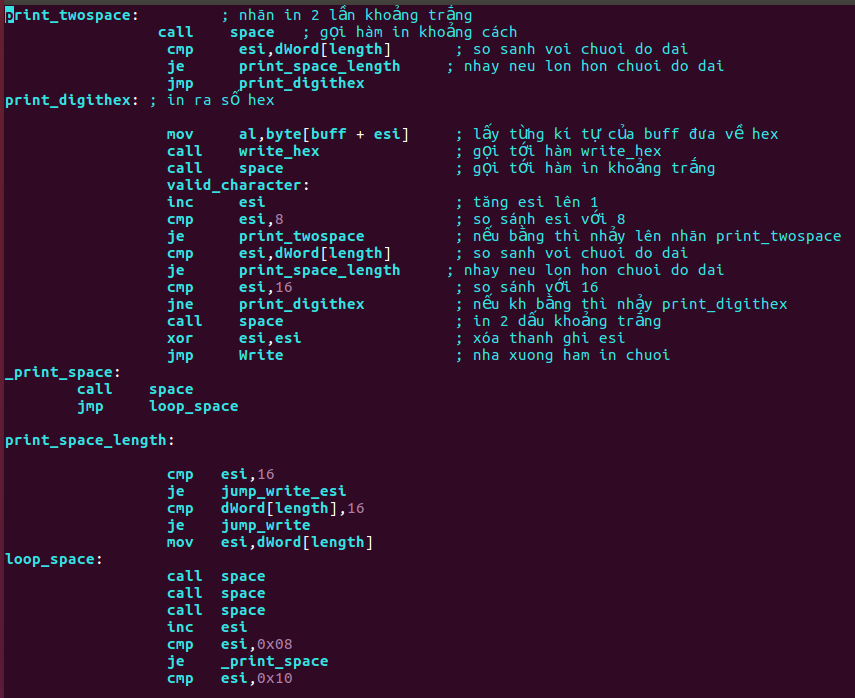
Hình 2.3.2‑10 In 16 bit kí tự

* Lúc này khi thanh ghi esi bằng 16 ta sẽ quan sát 16 kí tự sau khi được thay thế.



Hình 2.3.2‑11 In 16 bit kí tự

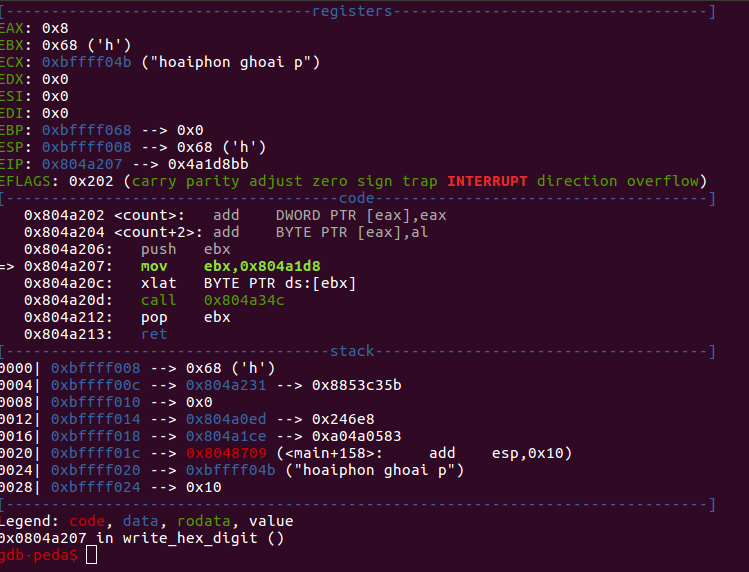
* Ta thu được kết quả sau khi thanh ghi esi = 16

In ra màn hình 16 kí tự bằng hệ hex

Hình 2.3.2‑12 In ra màn hình 16 kí tự bằng hệ hex



Hình 2.3.2‑13 In ra màn hình 16 kí tự bằng hệ hex

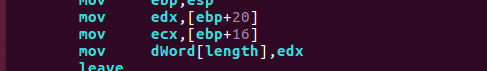


Hình 2.3.2‑14 In ra màn hình 16 kí tự bằng hệ hex

* **Sợ lược về đoạn code ở hình 2.6.3-1:**
* Ở đây em có xử lý cho cả 2 trương hợp
  + Độ dài chuỗi sẽ được truyền từ C vào là lưu vào biến length

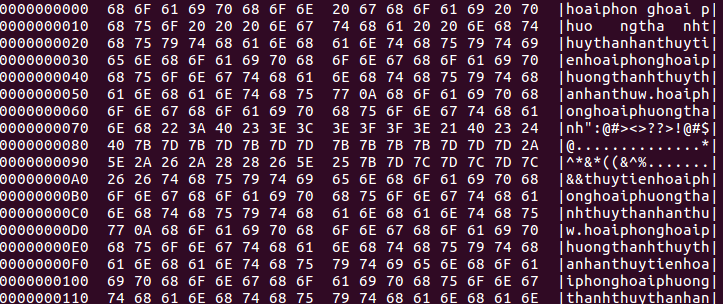


Hình 2.3.2‑15 In ra màn hình 16 kí tự bằng hệ hex



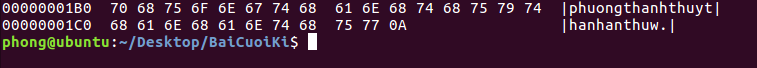
Hình 2.3.2‑16 In ra màn hình 16 kí tự bằng hệ hex

* + Trường hợp 1: Độ dài chuỗi truyền vào là bằng đúng 16 thì em sẽ không in kí tự khoảng trắng mà in ra đủ các kí tự khi truyền vào 16



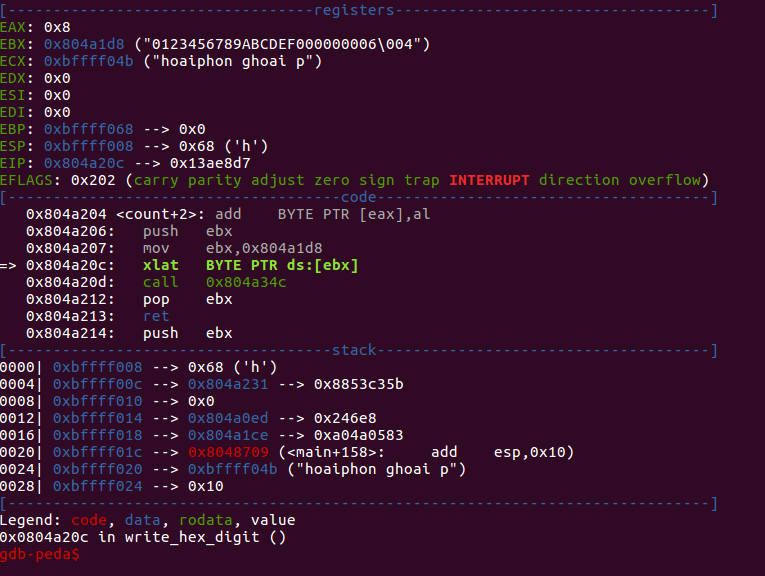
Hình 2.3.2‑17 Kết quả khi truyền vào đủ 16 kí tự

* + Trường hợp 2: Nếu độ dài chuỗi < 16 thì phải thay thế bằng các kí tự khoảng trắng để không làm bi lệch khi in cột thứ 4 là cột kí tự



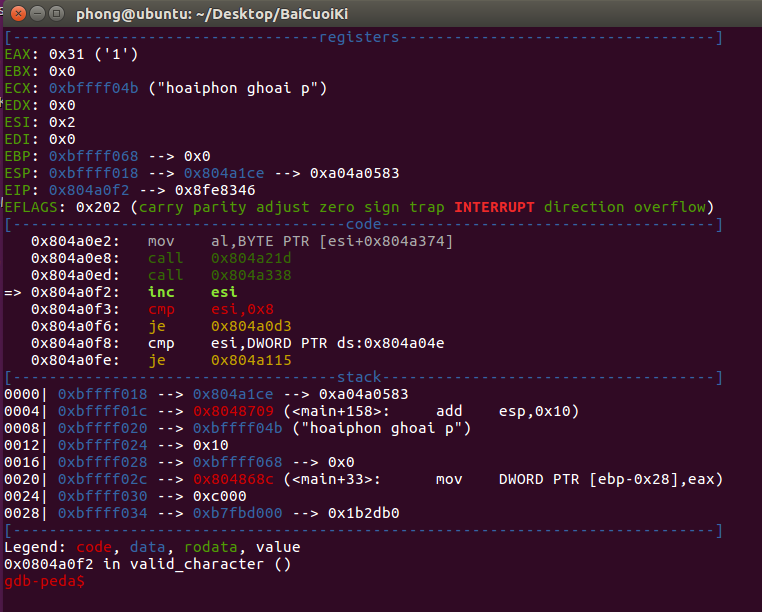
Hình 2.3.2‑18 Kết quả nếu không truyền đủ 16 kí tự

* **Chạy gdb để kiểm tra kết quả:**



Hình 2.3.2‑19 In ra màn hình 16 kí tự hệ hex

Các kí tự sẽ được tra trong xlat ở thư viện display.asm có trong hàm write hex mà em đã giải thích các phần code phía trên. Sau đó xuất ra các giá trị hex tương ứng

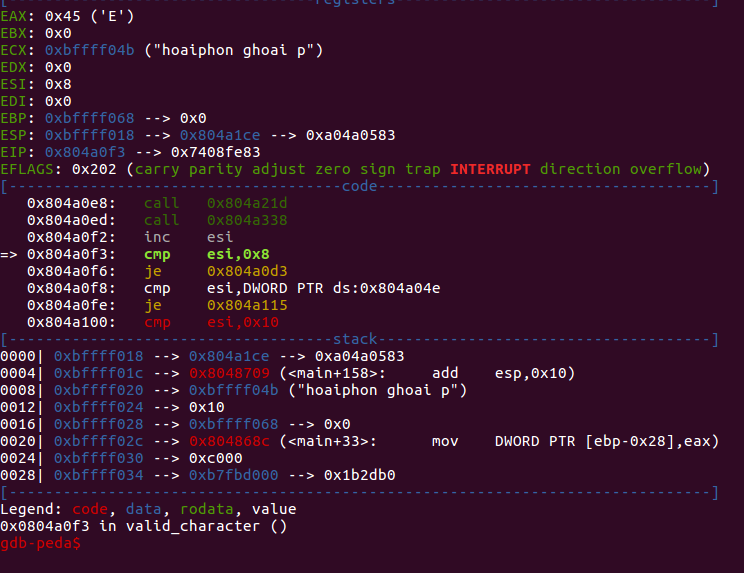


Hình 2.3.2‑20 In ra màn hình 16 kí tự bàng hex

* Ở hình trên nếu esi bằng 8 em sẽ nhảy đến nhãn in 2 khoảng cách. Vì cứ sau 8 và 16 thì phải in ra 2 khoảng cách.

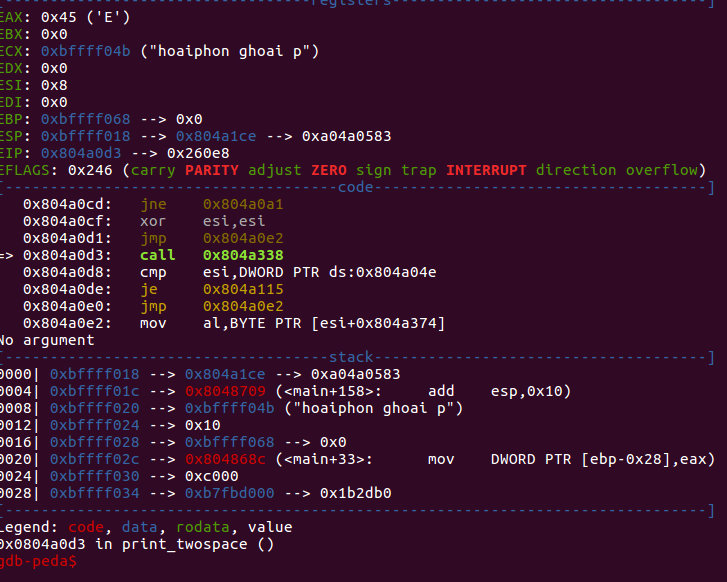


Hình 2.3.2‑21 In 2 lần khoảng cách



Hình 2.3.2‑22 In 2 lần khoảng cách

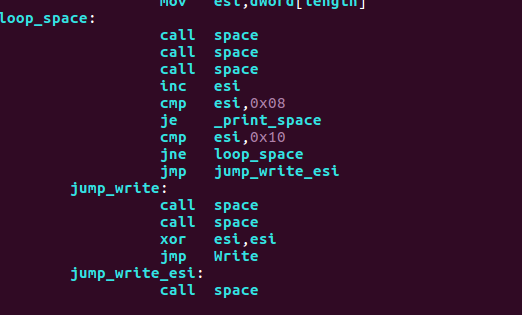
* Khi ESI bằng 8 thì sẽ in ra 2 lần khoảng cách



Hình 2.3.2‑23 In 2 lần khoảng cách



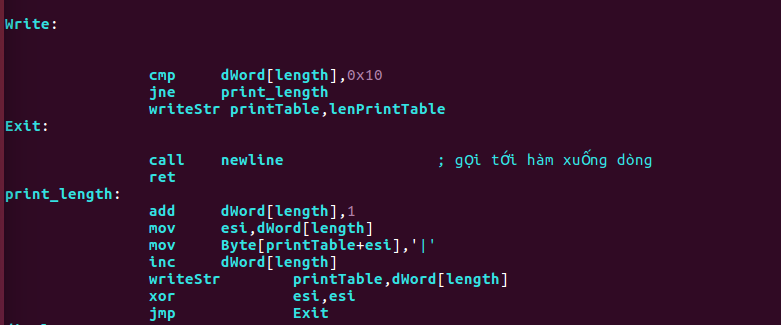
Hình 2.3.2‑24 Trường hợp nhỏ hơn 16 kí tự



Hình 2.3.2‑25 Trường hợp nhỏ hơn 16 kí tự

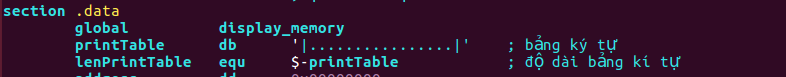
* Ở trường hợp nảy nếu như nhỏ hơn 16 kí tự thì sẽ in ra các khoảng trắng để không bị xô lệch khi in cột cuối cùng

### In ra màn hình cột kí tự

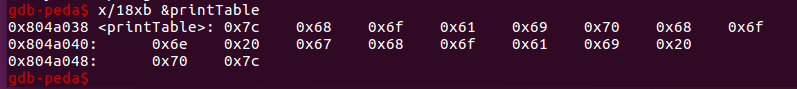


Hình 2.3.3‑1 In ra màn hình cột kí tự

* **Sở lược đoạn code in ra cột kí tự:** Ở đây em cũng chia ra 2 trường hợp
  + Trường hợp 1: Nếu trong trường hợp biến length bằng 16 thì lúc này các dấu chấm trong printTable đã được thay thế hết và mình có thể in ra với độ dài của lenPrintTable



Hình 2.3.3‑2 In ra màn hình cột kí tự



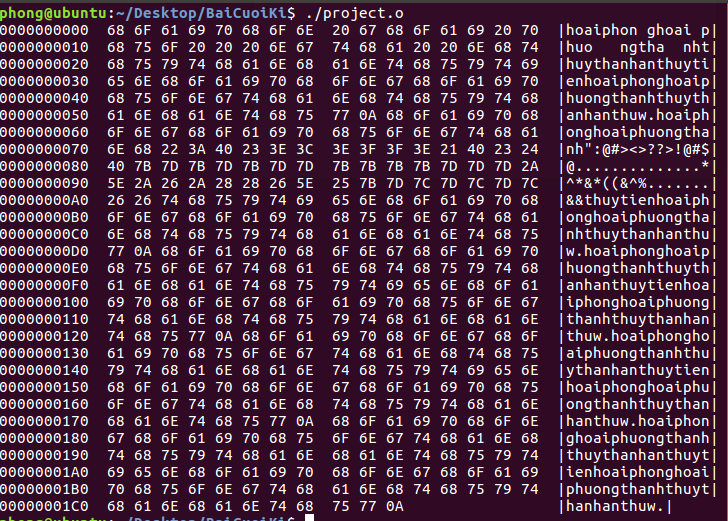
Hình 2.3.3‑3 In ra màn hình cột kí tự



Hình 2.3.3‑4 In ra màn hình cột kí tự

* Do giá trị length được truyền vào bằng 16 nên không cần thay dấu khoảng cách nào cho kí tự hex
  + Trường hợp 2: Nếu trong trường hợp biến length khác bằng 16 kí tự lúc các kí tự khoảng trắng sẽ được in ra để bảo đảm không xô lệch ở cột thứ 4





Hình 2.3.3‑5 Xử lý kiểu này sẽ giúp cột kí tự không bị xô lệch

# KẾT LUẬN

Sau quá trình hoàn thành bài project 1, em đã tích lũy cho mình những kiến thức cơ bản và có một chút nâng cao về hợp ngữ, điều đặc biệt ở đây là lập trình hợp ngữ mang tính tỉ mĩ rất cao. Nhờ vậy giúp em rèn luyện việc code cẩn thận và xử lý luồng tư duy của mình một cách hiệu quả và linh hoạt hơn. Nhờ đó mà em biết được mọi khi mình code một đoạn code ở chương trình cấp cao thì rất đơn giản và tự nhiên nhưng đối với hợp ngữ thì nó hoàn toàn ngược lại. Nhờ thế em mới biết được từ code ở ngôn ngữ cấp cao khi đưa xuống ngôn ngữ máy thì nó phức tạp và đòi hỏi sự chặt chẽ tới mức nào.

Quá trình code của em cũng chưa tối ưu về mặt logic, dữ liệu, code vẫn còn quá dài khi xử lý vấn đề nào đó. Nhờ trong quá trình làm project em có thể tổng hợp được các kiến thức mà mình học được, và tìm hiểu thêm một số cái mới mà mình còn chưa biết

Cuối cùng em xin cảm ơn thầy **Nguyễn Đăng Quang** nhờ có những chia sẻ và góp ý từ thầy em đã tổng hợp kiến thức và hoàn thành project một cách thuận lợi.

Sinh viên thực hiện

Phong

Hồ Hoài Phong

# Tài Liệu Tham Khảo

1. **ThS Nguyễn Đăng Quang.** *Kiến trúc máy tính & hợp ngữ.* Thành Phố Hồ Chí Minh : https://www.youtube.com/channel/UCHvwAqtCYl0nRKu\_jg2PsNA, 2021.

2. **William Stallings.** *Computer Organization and Architecture, 9th Edition, Pearson Education, Prentice-Hall.* 2013.

3. **https://www.eecg.utoronto.ca/~amza/www.mindsec.com/files/x86regs.html.**

4. **https://www.tutorialspoint.com/assembly\_programming/index.htm.**

5. **https://stackoverflow.com/questions/tagged/nasm.**