Lab 2 : DEBUG CHƯƠNG TRÌNH DÙNG TRÌNH GDB

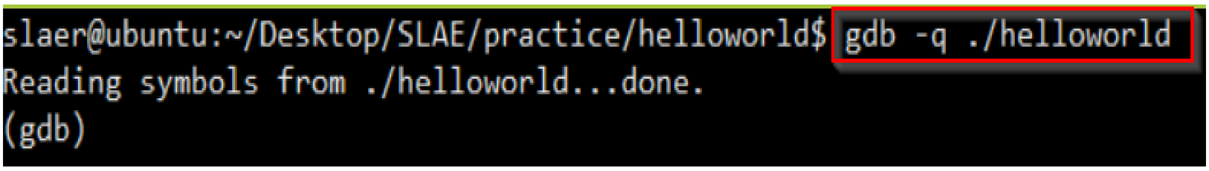
1. **Chuẩn đầu ra :** Sau bài này, người học có thể :
   - ✓ Debug được chương trình
2. **Chuẩn bị :** Đọc trước phần lý thuyết về các lệnh của Debug.
3. **Phương tiện :**
   - ✓ Máy vi tính.
   - ✓ Chương trình gdb của hệ điều hành.
4. **Thời lượng : 4 tiết**
5. **Tóm tắt lý thuyết : Giới thiệu về trình debug trên linux**

   GNU Debugger (GDB) is one of the most important tool while writing any low-level program. We will see the basic usage of GDB. Using GDB is simple. Here see yourself

   $ gdb <options> <executable>

   Well not really!!! Let me help you to understand some important options of GDB.

   First type "$ gdb -h" to get all the available options. Out of all that the only option use is "-q" which is quit start. It just suppresses licensing info. Another option which can be helpful is "-p" which is used for attaching already running process to GDB.



```
slaer@ubuntu:~/Desktop/SLAE/practice/helloworld$ gdb -q ./helloworld
Reading symbols from ./helloworld...done.
(gdb)
```

   So now the program is loaded, we will see some internal options.

Inspecting loaded executable inside GDB:

- "**info**" (i) command is used for extracting information of various kind. Just type "(gdb) help info" to get every available option. Some useful options are:
   2. info registers — List of integer registers and their contents
   3. info symbol — Describe what symbol is at location ADDR
   4. info breakpoints — Status of specified breakpoints (all user-settable breakpoints if no argument)
   5. info files — Names of targets and files being debugged
- "**break**" (b) command is used for setting a break point. This breakpoint can be set against "address", "function" etc. GDB also offers facility of conditional breakpoints, which we will be using multiple times
- "**run**" (r) command will run the loaded executable inside GDB.
- "**disassemble**" (disas) is the command to disassemble the pointed instruction.
- "**stepi**" command will help to execute step by step execution.

- "**x**" command is used for Examining the memory in various formats. We will look it more details in future posts.
- "**print**" (p) command will print register values.
-

6. Tóm tắt các lệnh trong debug

Command	**Example**	**Description**
run		start program
quit		quit out of gdb
cont		continue execution after a break
break [addr]	break *_start+5	sets a breakpoint
delete [n]	delete 4	removes nth breakpoint
delete		removes all breakpoints
info break		lists all breakpoints
stepi		execute next instruction
stepi [n]	stepi 4	execute next n instructions
nexti		execute next instruction, stepping over function calls
nexti [n]	nexti 4	execute next n instructions, stepping over function calls
where		show where execution halted

disas [addr]	disas _start	disassemble instructions at given address
info registers		dump contents of all registers
print/d [expr]	print/d $ecx	print expression in decimal
print/x [expr]	print/x $ecx	print expression in hex
print/t [expr]	print/t $ecx	print expression in binary
x/NFU [addr]	x/12xw &msg	Examine contents of memory in given format
display [expr]	display $eax	automatically print the expression each time the program is halted
info display		show list of automatically displays
undisplay [n]	undisplay 1	remove an automatic display

7. Nội dung thực hành

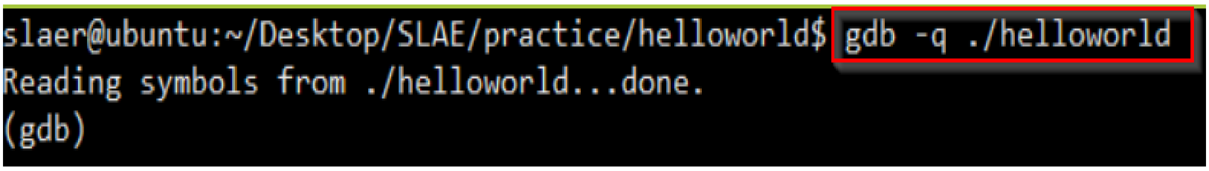
7.1. Nạp chương trình sau vào

```
; This is simple hello world code
; Author: SLAER (Shashank Gosavi)

global _start

    section .text
    _start:

    xor ecx, ecx ; Clearing ECX
    xor ebx, ebx ; Clearing EBX
    mul ecx      ; Clearing EAX, EDX

    ; Write subroutine

    mov eax, 0x4 ; Moving Write syscall number into EAX
    mov ebx, 0x1 ; Moving file descriptor into EBX
    mov ecx, $msg ; Moving actual buffer into ECX
    mov edx, $len ; Moving the count into EDX
    int 0x80 ; Interrupt 80

    ; Graceful Exit
    mov eax, 0x1 ; Moving Exit sysscall number into EAX
    mov ebx, 0x0 ; Moving status number = 0 in EBX
    int 0x80 ; Interrupt 80

    section .data
    msg: db "Hello World!",0x0A
    len: equ $-msg
```

- **lưu chương trình với tên là helloworld.nasm**
- **biên dịch chương trình với nasm**
- **liên kết chương trình với ld**
-

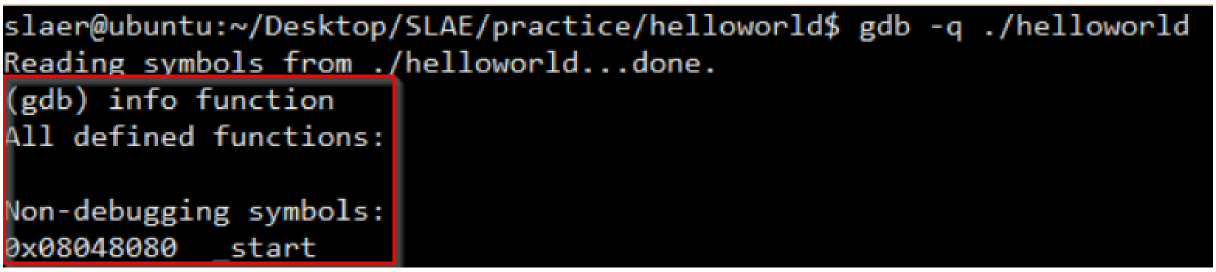
7.2. Khởi động chương trình Debug :

- Load program in GDB



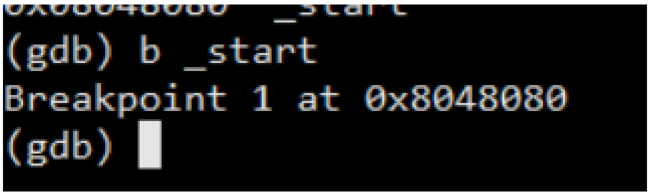
```
slaer@ubuntu:~/Desktop/SLAE/practice/helloworld$ gdb -q ./helloworld
Reading symbols from ./helloworld...done.
(gdb)
```

- Type following command to get details about list of symbols in the executable.



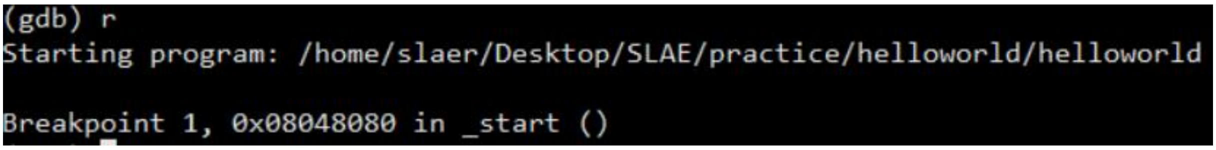
```
slaer@ubuntu:~/Desktop/SLAE/practice/helloworld$ gdb -q ./helloworld
Reading symbols from ./helloworld...done.
(gdb) info function
All defined functions:

Non-debugging symbols:
0x08048080  _start
```

- Now set breakpoint for _start function.



```
0x08048080  _start
(gdb) b _start
Breakpoint 1 at 0x8048080
(gdb) █
```

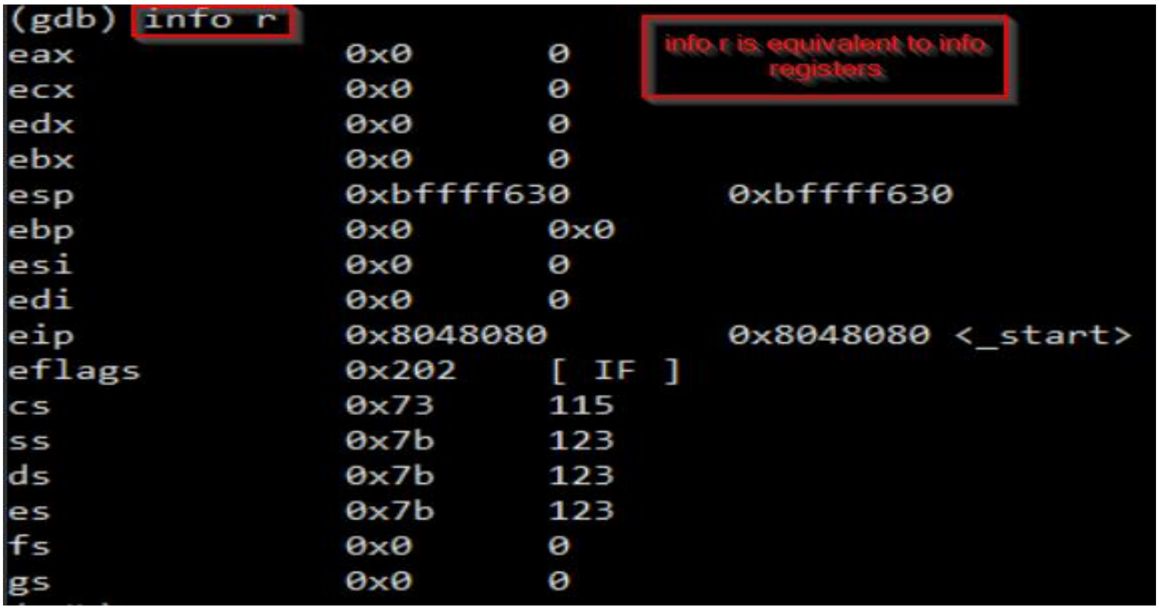
- Now run the program.



```
(gdb) r
Starting program: /home/slaer/Desktop/SLAE/practice/helloworld/helloworld

Breakpoint 1, 0x08048080 in _start ()
```

You can observe that on running the program the break point is hit. This is because execution starts with _start.

- Now we can check the registers etc. Lets do that

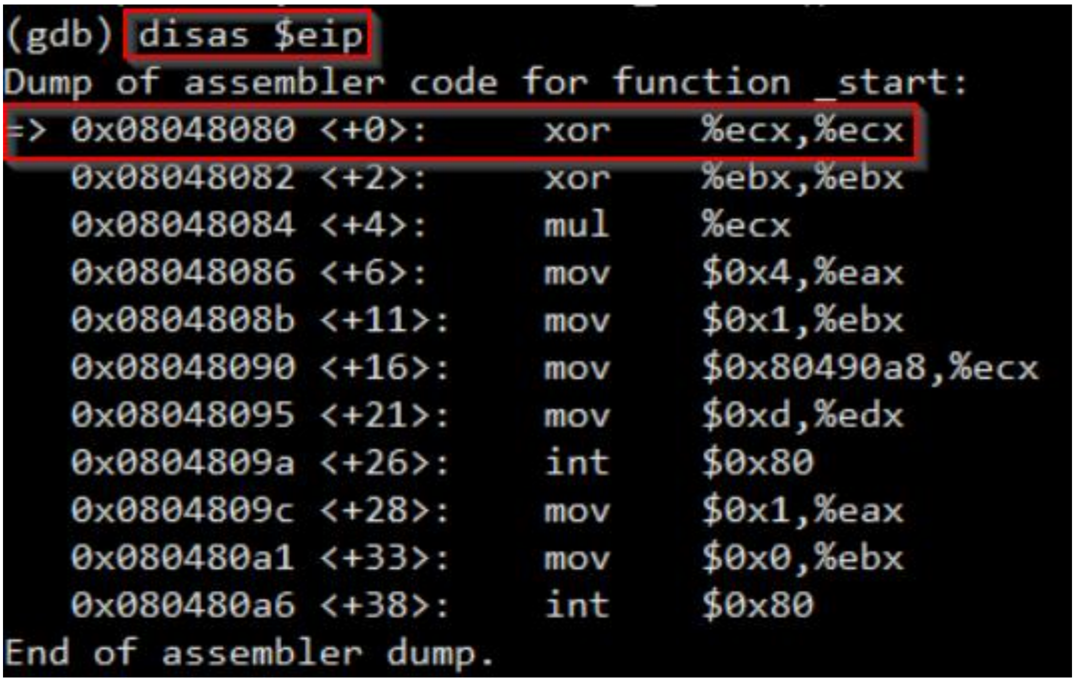


```
(gdb) info r
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbffff630       0xbffff630
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048080        0x8048080 <_start>
eflags         0x202    [ IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

info r is equivalent to info registers

As we can see since program is not running register values are mostly zero. Mind you that these values are relative.

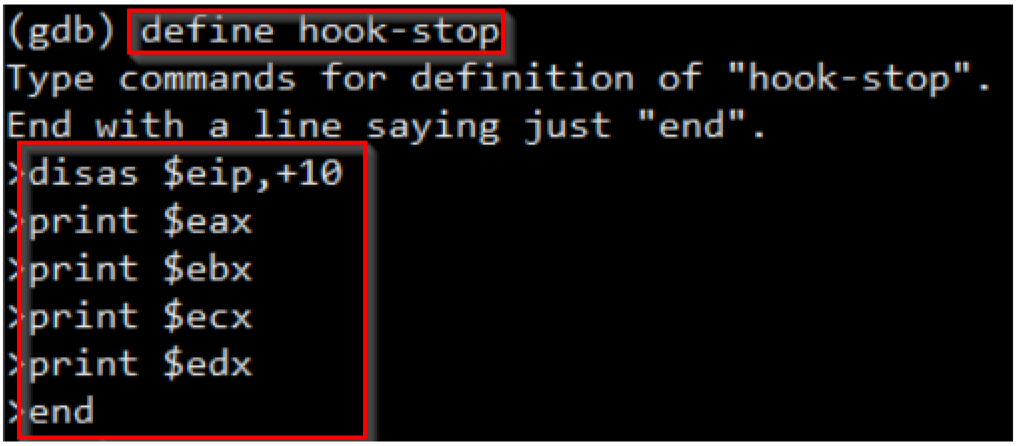
- Lets disassemble the code. Note that disas command can disassemble address, function or register value. In our case we have disassembled EIP register value, which is address of next instruction



```
(gdb) disas $eip
Dump of assembler code for function _start:
=> 0x08048080 <+0>:     xor    %ecx,%ecx
   0x08048082 <+2>:     xor    %ebx,%ebx
   0x08048084 <+4>:     mul    %ecx
   0x08048086 <+6>:     mov    $0x4,%eax
   0x0804808b <+11>:    mov    $0x1,%ebx
   0x08048090 <+16>:    mov    $0x80490a8,%ecx
   0x08048095 <+21>:    mov    $0xd,%edx
   0x0804809a <+26>:    int    $0x80
   0x0804809c <+28>:    mov    $0x1,%eax
   0x080480a1 <+33>:    mov    $0x0,%ebx
   0x080480a6 <+38>:    int    $0x80
End of assembler dump.
```

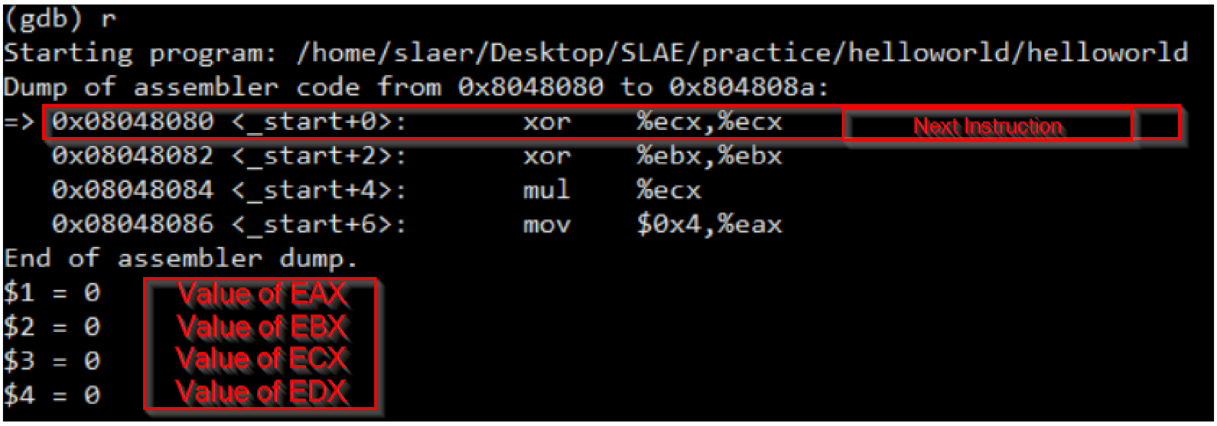
The arrow (=>) is showing next instruction to be executed.

- Now lets see one interesting feature of GDB called Hook. Hook is basically used for binding number of instruction to be executed per instruction. So lets “define hook-stop”



```
(gdb) define hook-stop
Type commands for definition of "hook-stop".
End with a line saying just "end".
>disas $eip,+10
>print $eax
>print $ebx
>print $ecx
>print $edx
>end
```

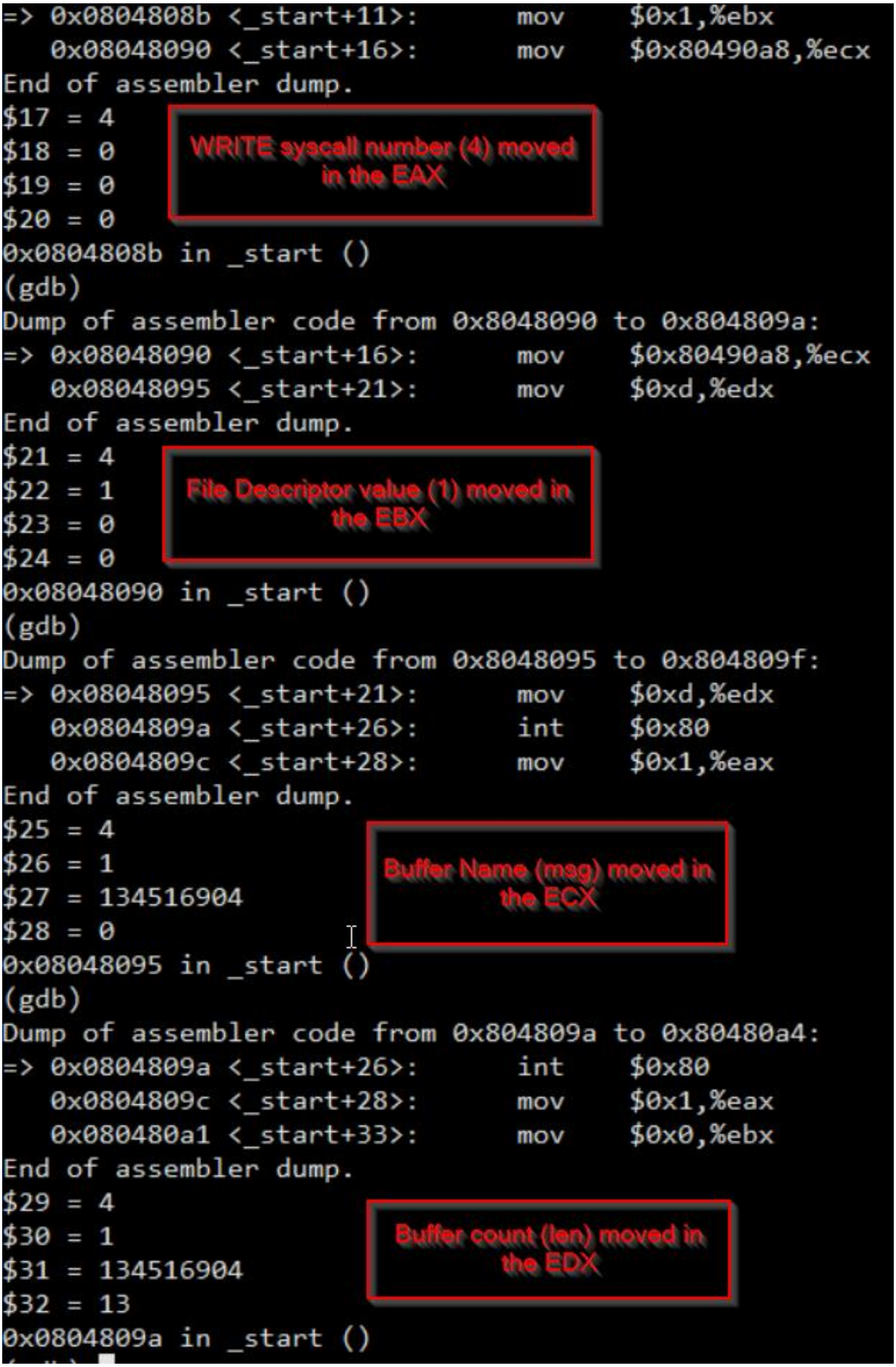
Here I have defined very simple hook. It will disassemble $eip and next 10 instructions, then display value of EAX, EBX, ECX, EDX respectively. On running program, you’ll get following output:



```
(gdb) r
Starting program: /home/slaer/Desktop/SLAE/practice/helloworld/helloworld
Dump of assembler code from 0x8048080 to 0x804808a:
=> 0x08048080 <_start+0>:       xor    %ecx,%ecx
   0x08048082 <_start+2>:       xor    %ebx,%ebx
   0x08048084 <_start+4>:       mul    %ecx
   0x08048086 <_start+6>:       mov    $0x4,%eax
End of assembler dump.
$1 = 0
$2 = 0
$3 = 0
$4 = 0
```

We can observe the step by step changes in the value of registers. So after couple of “stepi”s it will be something like below.

- Gõ liên tiếp một số lần lệnh stepi
- Kết quả sẽ như hình vẽ



```
=> 0x0804808b <_start+11>:      mov    $0x1,%ebx
   0x08048090 <_start+16>:      mov    $0x80490a8,%ecx
End of assembler dump.
$17 = 4
$18 = 0
$19 = 0
$20 = 0
0x0804808b in _start ()
(gdb)
```

WRITE syscall number (4) moved in the EAX

```
Dump of assembler code from 0x8048090 to 0x804809a:
=> 0x08048090 <_start+16>:      mov    $0x80490a8,%ecx
   0x08048095 <_start+21>:      mov    $0xd,%edx
End of assembler dump.
$21 = 4
$22 = 1
$23 = 0
$24 = 0
0x08048090 in _start ()
(gdb)
```

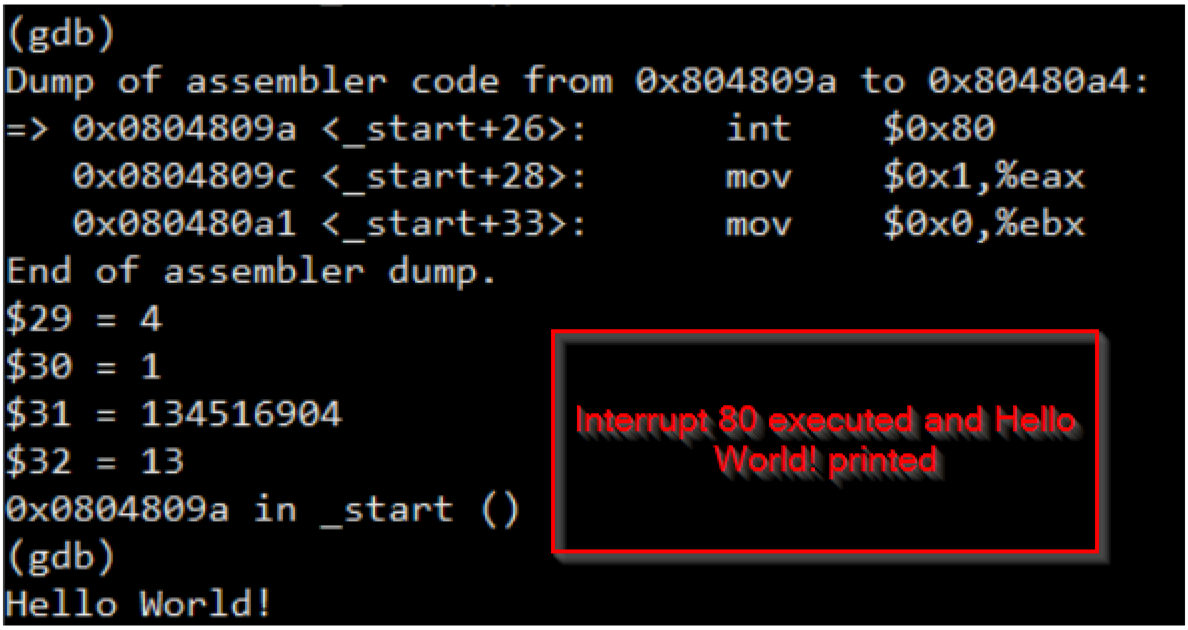
File Descriptor value (1) moved in the EBX

```
Dump of assembler code from 0x8048095 to 0x804809f:
=> 0x08048095 <_start+21>:      mov    $0xd,%edx
   0x0804809a <_start+26>:      int    $0x80
   0x0804809c <_start+28>:      mov    $0x1,%eax
End of assembler dump.
$25 = 4
$26 = 1
$27 = 134516904
$28 = 0
0x08048095 in _start ()
(gdb)
```

Buffer Name (msg) moved in the ECX

```
Dump of assembler code from 0x804809a to 0x80480a4:
=> 0x0804809a <_start+26>:      int    $0x80
   0x0804809c <_start+28>:      mov    $0x1,%eax
   0x080480a1 <_start+33>:      mov    $0x0,%ebx
End of assembler dump.
$29 = 4
$30 = 1
$31 = 134516904
$32 = 13
0x0804809a in _start ()
```

Buffer count (len) moved in the EDX



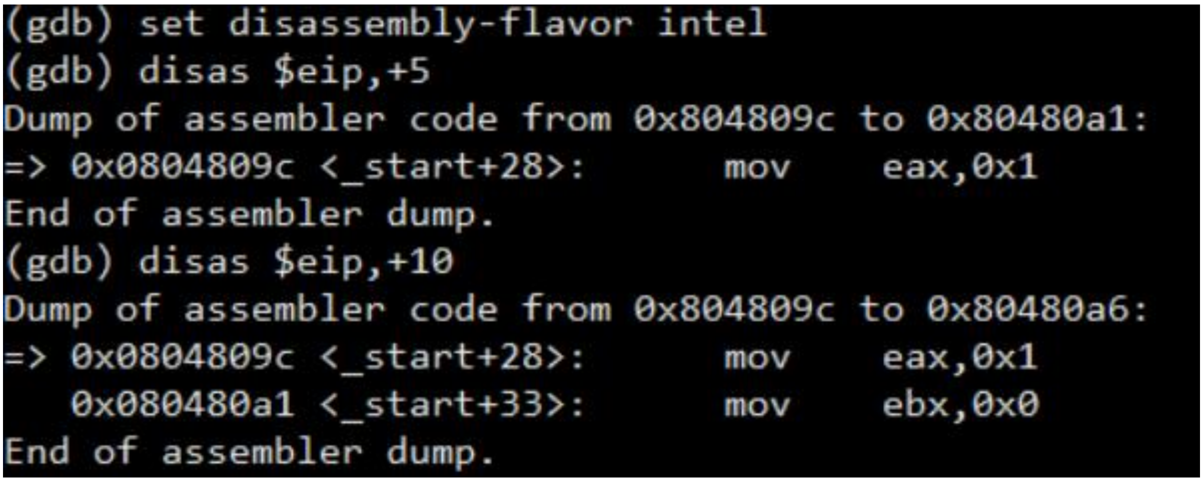
```
(gdb)
Dump of assembler code from 0x804809a to 0x80480a4:
=> 0x0804809a <_start+26>:      int     $0x80
   0x0804809c <_start+28>:      mov     $0x1,%eax
   0x080480a1 <_start+33>:      mov     $0x0,%ebx
End of assembler dump.
$29 = 4
$30 = 1
$31 = 134516904
$32 = 13
0x0804809a in _start ()
(gdb)
Hello World!
```

Interrupt 80 executed and Hello World! printed

-

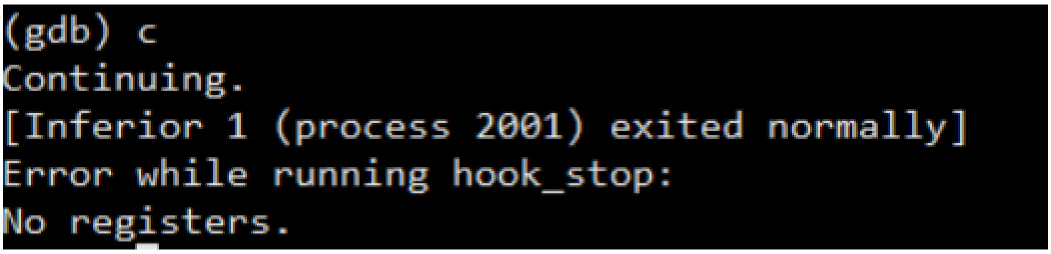
7.3. Lưu ý :

- Update: I forgot to tell you one very important thing. By default follows ATT convention disassembled code. Above disassembly convention is ATT (full of $ and %). To change the convention to Intel, use following command:
- (gdb) set disassembly-flavor intel
- Now if you run "disas" command you can see following:



```
(gdb) set disassembly-flavor intel
(gdb) disas $eip,+5
Dump of assembler code from 0x804809c to 0x80480a1:
=> 0x0804809c <_start+28>:      mov     eax,0x1
End of assembler dump.
(gdb) disas $eip,+10
Dump of assembler code from 0x804809c to 0x80480a6:
=> 0x0804809c <_start+28>:      mov     eax,0x1
   0x080480a1 <_start+33>:      mov     ebx,0x0
End of assembler dump.
```

- Finally just type "c" to continue execution. It will execute the program to the end, if no other breakpoint present.



```
(gdb) c
Continuing.
[Inferior 1 (process 2001) exited normally]
Error while running hook_stop:
No registers.
```

7.4. Luyện tập thêm

7.4.1. Nạp chương trình sau

```
; Title: Data Types
; Description: Simple code to understand datatypes and representation in IA-32
; Author: Shashank "SLAER" Gosavi
 global _start
 section .text
     _start:
 ; Graceful Exit code
     mov eax, 1
     mov ebx, 0
     int 80h
 section .data
     var1: db 0x55            ; Just byte 0x55
     var2: db 0x55, 0x56, 0x57     ; three bytes in succession
     var3: db 'a', 0x55            ; character constant
     var4: db 'hi', 14, 15, '$'    ; string constant
     var5: dw 0x1234               ; 0x34 0x12 due to Little Endianness
     var6: dw 'a'             ; 0x61 0x00 (just number)
     var7: dw 'ab'            ; Character constant
     var8: dw 'abc'           ; 0x61 0x62 0x63 0x00 (string)
     var9: dd 0x12345678      ; 0x78 0x56 0x34 0x12
     var10: dd 1.234567e20         ; floating-point constant
     var11: dq 0x123456789abcdef0  ; eight-byte constant
     var12: dq 1.234567e20         ; double-precision float
     var13: dt 1.234567e20         ; extended-precision float
```

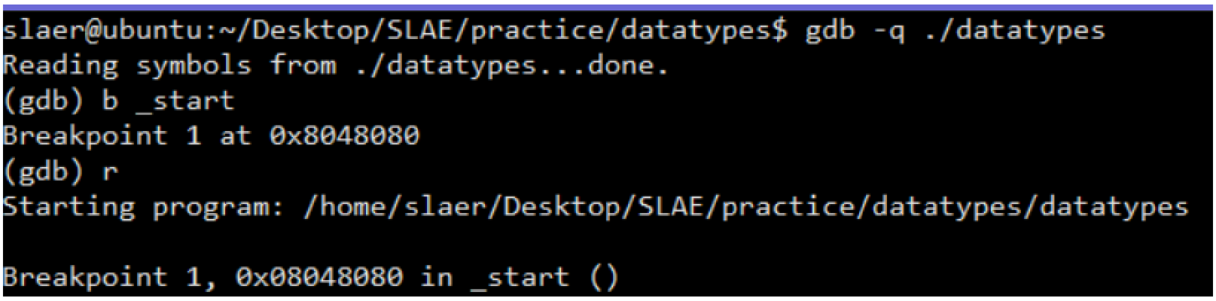
```
section .bss

    buffer: resb 64                ; reserve 64 byte

    wordvar: resw 1           ; reserve a word
```

7.4.2. Các bước tiếp theo

- **Lưu tập tin với tên datatypes.nasm**
- **Biên dịch file nguồn bằng lệnh nasm**
- **Liên kết mã đối tượng bằng lệnh ld**

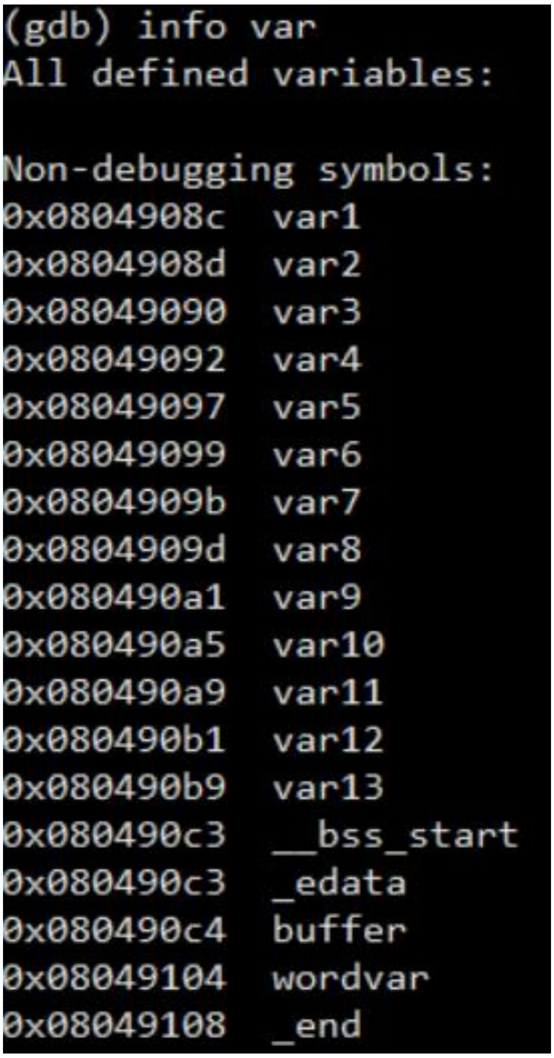
7.4.3. Debug chương trình theo trình tự sau

- **Ra lệnh gdb –q ./datatypes**



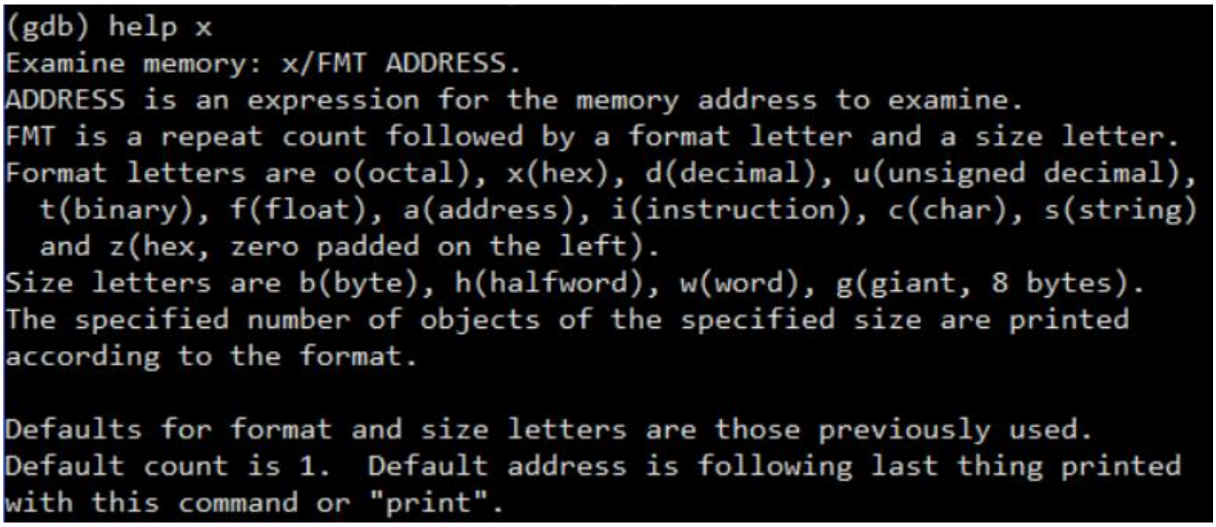
```
slaer@ubuntu:~/Desktop/SLAE/practice/datatypes$ gdb -q ./datatypes
Reading symbols from ./datatypes...done.
(gdb) b _start
Breakpoint 1 at 0x8048080
(gdb) r
Starting program: /home/slaer/Desktop/SLAE/practice/datatypes/datatypes

Breakpoint 1, 0x08048080 in _start ()
```

- (Executable Loaded in GDB with Breakpoint set to "_start" symbol. "r" for starting execution of code)
- First, we will list out all the variables present in the code with "info variables" command.



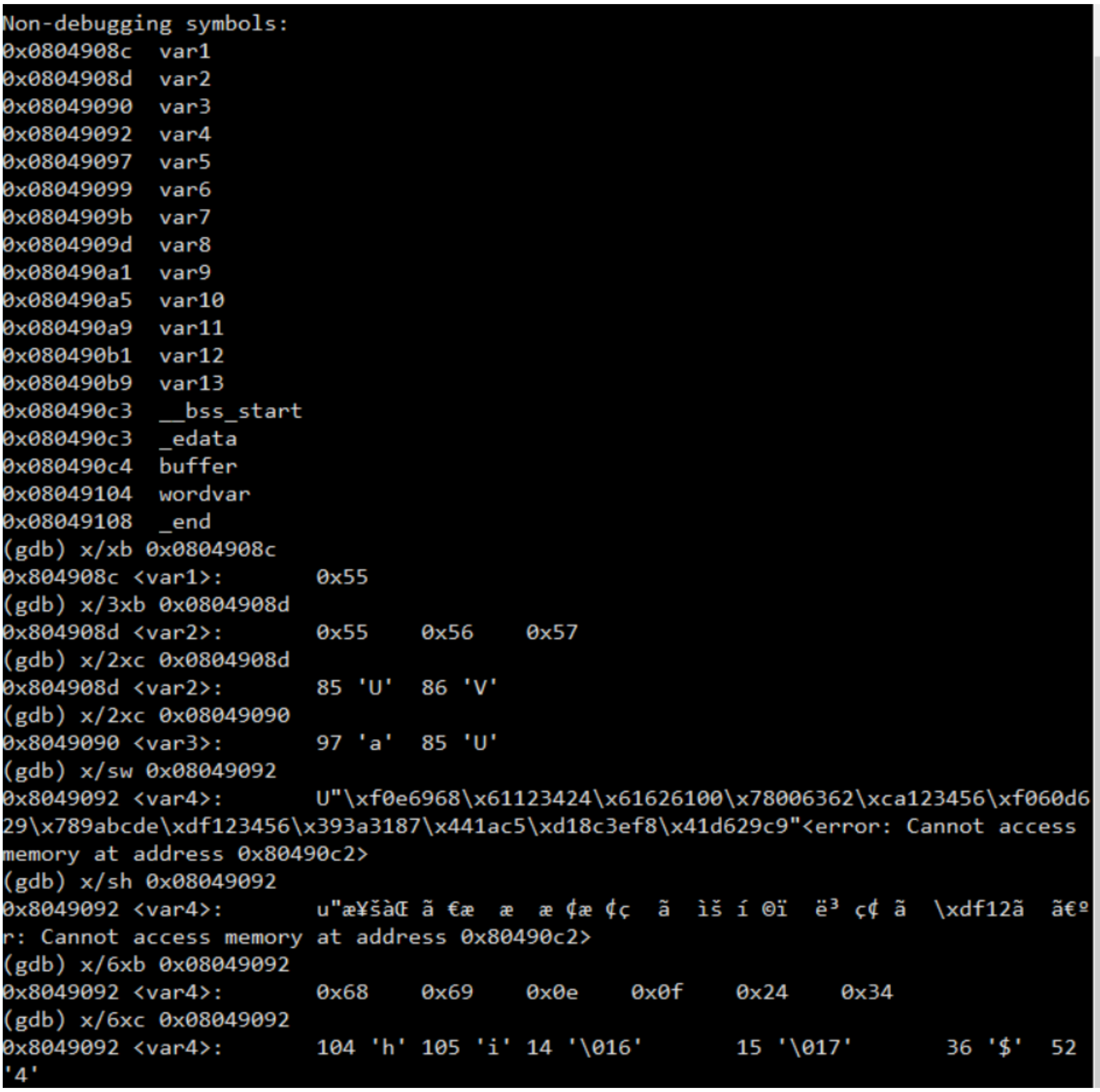
```
(gdb) info var
All defined variables:

Non-debugging symbols:
0x0804908c  var1
0x0804908d  var2
0x08049090  var3
0x08049092  var4
0x08049097  var5
0x08049099  var6
0x0804909b  var7
0x0804909d  var8
0x080490a1  var9
0x080490a5  var10
0x080490a9  var11
0x080490b1  var12
0x080490b9  var13
0x080490c3  __bss_start
0x080490c3  _edata
0x080490c4  buffer
0x08049104  wordvar
0x08049108  _end
```

- To examine contents of variables and registers, “x” command is used in GDB. “help x” command will show following output.



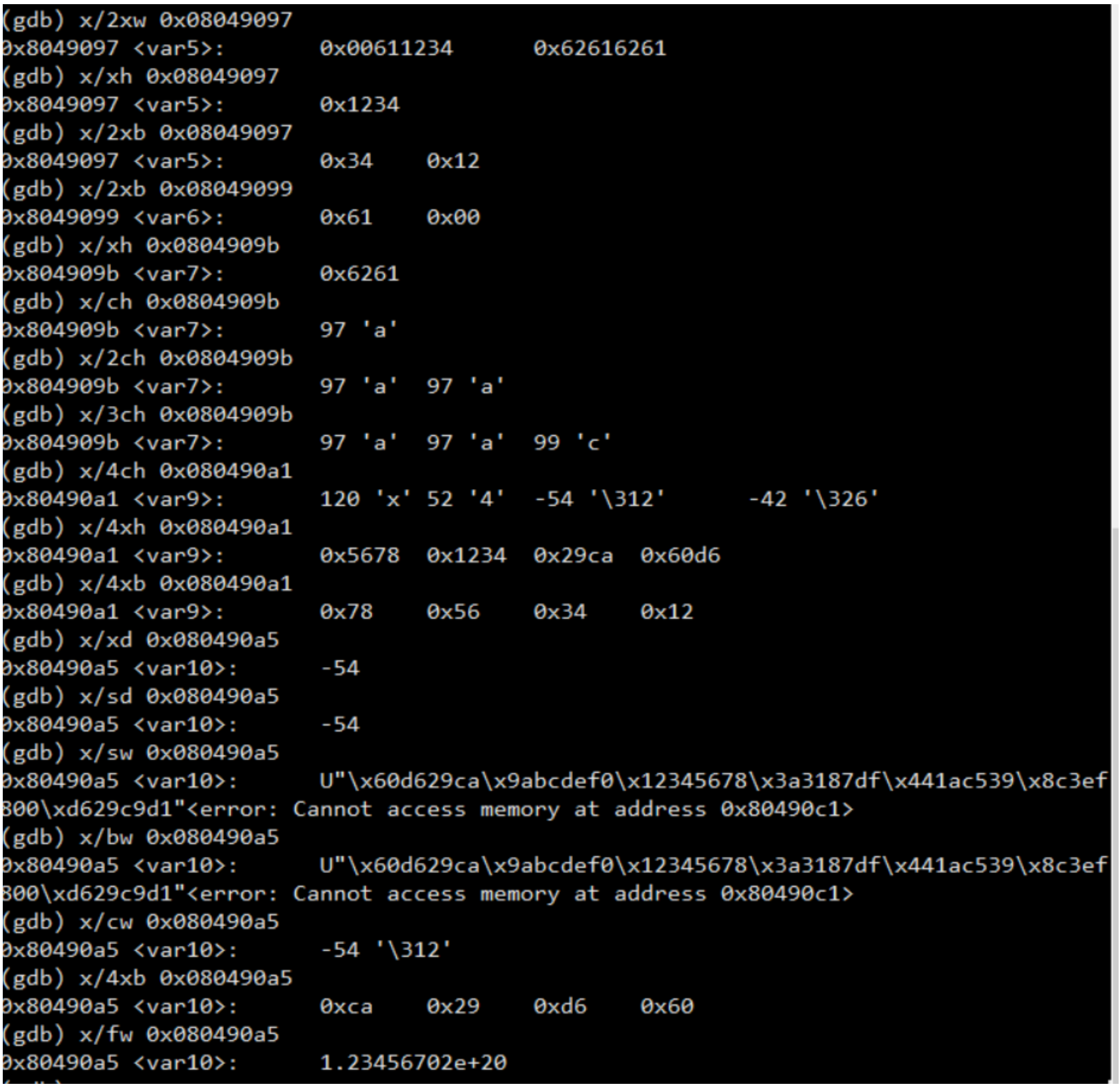
```
(gdb) help x
Examine memory: x/FMT ADDRESS.
ADDRESS is an expression for the memory address to examine.
FMT is a repeat count followed by a format letter and a size letter.
Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal),
  t(binary), f(float), a(address), i(instruction), c(char), s(string)
  and z(hex, zero padded on the left).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).
The specified number of objects of the specified size are printed
according to the format.

Defaults for format and size letters are those previously used.
Default count is 1.  Default address is following last thing printed
with this command or "print".
```

- Now based on above screenshot, you can understand below screenshots.



```
Non-debugging symbols:
0x0804908c  var1
0x0804908d  var2
0x08049090  var3
0x08049092  var4
0x08049097  var5
0x08049099  var6
0x0804909b  var7
0x0804909d  var8
0x080490a1  var9
0x080490a5  var10
0x080490a9  var11
0x080490b1  var12
0x080490b9  var13
0x080490c3  __bss_start
0x080490c3  _edata
0x080490c4  buffer
0x08049104  wordvar
0x08049108  _end
(gdb) x/xb 0x0804908c
0x804908c <var1>:       0x55
(gdb) x/3xb 0x0804908d
0x804908d <var2>:       0x55    0x56    0x57
(gdb) x/2xc 0x0804908d
0x804908d <var2>:       85 'U'  86 'V'
(gdb) x/2xc 0x08049090
0x8049090 <var3>:       97 'a'  85 'U'
(gdb) x/sw 0x08049092
0x8049092 <var4>:       U"\xf0e6968\x61123424\x61626100\x78006362\xca123456\xf060d6
29\x789abcde\xdf123456\x393a3187\x441ac5\xd18c3ef8\x41d629c9"<error: Cannot access
memory at address 0x80490c2>
(gdb) x/sh 0x08049092
0x8049092 <var4>:       u"æ¥šàŒ ã €æ  æ  æ ¢æ ¢ç  ã  ìš í ©ï  ë³ ç¢ ã  \xdf12ã  ã€º
r: Cannot access memory at address 0x80490c2>
(gdb) x/6xb 0x08049092
0x8049092 <var4>:       0x68    0x69    0x0e    0x0f    0x24    0x34
(gdb) x/6xc 0x08049092
0x8049092 <var4>:       104 'h' 105 'i' 14 '\016'       15 '\017'       36 '$'  52
'4'
```

- You can observe that I have tried couple of options here. You also have to do trial and error to get intended output. Same goes with next screenshot.



```
(gdb) x/2xw 0x08049097
0x8049097 <var5>:       0x00611234      0x62616261
(gdb) x/xh 0x08049097
0x8049097 <var5>:       0x1234
(gdb) x/2xb 0x08049097
0x8049097 <var5>:       0x34    0x12
(gdb) x/2xb 0x08049099
0x8049099 <var6>:       0x61    0x00
(gdb) x/xh 0x0804909b
0x804909b <var7>:       0x6261
(gdb) x/ch 0x0804909b
0x804909b <var7>:       97 'a'
(gdb) x/2ch 0x0804909b
0x804909b <var7>:       97 'a'  97 'a'
(gdb) x/3ch 0x0804909b
0x804909b <var7>:       97 'a'  97 'a'  99 'c'
(gdb) x/4ch 0x080490a1
0x80490a1 <var9>:       120 'x' 52 '4'  -54 '\312'      -42 '\326'
(gdb) x/4xh 0x080490a1
0x80490a1 <var9>:       0x5678  0x1234  0x29ca  0x60d6
(gdb) x/4xb 0x080490a1
0x80490a1 <var9>:       0x78    0x56    0x34    0x12
(gdb) x/xd 0x080490a5
0x80490a5 <var10>:      -54
(gdb) x/sd 0x080490a5
0x80490a5 <var10>:      -54
(gdb) x/sw 0x080490a5
0x80490a5 <var10>:      U"\x60d629ca\x9abcdef0\x12345678\x3a3187df\x441ac539\x8c3ef
800\xd629c9d1"<error: Cannot access memory at address 0x80490c1>
(gdb) x/bw 0x080490a5
0x80490a5 <var10>:      U"\x60d629ca\x9abcdef0\x12345678\x3a3187df\x441ac539\x8c3ef
800\xd629c9d1"<error: Cannot access memory at address 0x80490c1>
(gdb) x/cw 0x080490a5
0x80490a5 <var10>:      -54 '\312'
(gdb) x/4xb 0x080490a5
0x80490a5 <var10>:      0xca    0x29    0xd6    0x60
(gdb) x/fw 0x080490a5
0x80490a5 <var10>:      1.23456702e+20
```

- You must try this all by yourself. That’s the only way to learn… Anyways, I hope this is sufficient to help you understand how to use x command.

-