HOW TO DEBUG ASSEMBLY WITH GDB:

I am sure that many who have wanted to learn assembly have started off from the book "Assembly Language step by step" by Jeff Duntemann. This book , even though made for assembly language is also a great introduction to the complete bare workings of how computers work, and that is not easy. I have mentioned this book because it shows you how to use a debugger called "Insight", which is a very good GUI front-end for the GNU debugger. The only thing is, Debian has taken this package off of its repository, which means people running Debian or Ubuntu(?) cannot download the package, but you can (try, but after trying myself have found countless dependency problems) to install it from the source code which you can find from the official website.

Because of this, when I was learning from this book, I began cursing Debian, but then just though about which debugger to use. I came across DDD, another GUI front-end to gdb, but then found it stupid with all the windows, where resizing was a pain, and couldn't actually get really into the sticky stuff because the amount of tutorials on assembly debugging assembly with DDD is close to 0. So then I decided to use the gdb back-end on the very well unified terminal emulator.

I could also find many tutorials on this myself, giving information on commands you can use, but the information on those commands is limited and does not extend out to a full explanation of those commands. This is what I wish to do with commands that need more explanation, to the best of my ability.

First of all, the program code in the terminal text below is from Jeff Duntemann's book, and Jeff Duntemann himself (just to make it clear). The copied terminal text is more of a trial and error, first experience of gdb from what I have done with it. It is more there to show you what a session of debugging may look like with gdb.

The command "info registers" gives you all the values of all of the registers in hex and will be the mostly used command when debugging, especially, assembly. this can be printed out when you have single stepped an instruction and want to see the changes of that instruction. Another one of the mostly used commands is the "x" command which eXamines a given amount of memory starting from an address, and allows you to interpret the values from memory as hex values, characters, strings or instructions. The printed information from gdb's help for the x command displays this:

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".

This shows the format for this command, so by typing:
(gdb) x/10cb 0x80490a4
will show 10 (after "/") bytes ("b") interpreted as characters ("c") starting at the 32-bit memory

address "0x80490a4". I believe this command can be used with the variables of memory instead of the exact memory address, where such a variable would exist in the .data section of your assembly code. In this example, this command could be used:
x/10cb EatMsg
This command should print out the contents of memory starting off where the address "EatMsg" points to. This format of the command would be used for many purposes, but mainly when us humans cannot remember 32-bit addresses in hex and use our predefined variables instead.

Another command that you will definitely use is "list", which, as you may see below displays the source code that the debugging information in the executable points to, which can then be used to set breakpoints. another command disassembles the executable, which while used for C debugging, is a little less useful as in assembly, the source code command "list" also prints out the comments you have used (never forget them, or I will find you and disassemble your computer...), so when you may get a little lost in your code, the results of disassemble command may tell you where you are (as you can find out by the default instruction stepping output, which will be talked about later), using list, you can refer to your very friendly words of advice every semi-colon..

The next command that will be assessed is the "break" command. This command allows you to set a breakpoint, which is kind of the whole meaning of a debugger, where the program is halted and you are allowed to examine the effects of each instruction of your program after the set breakpoint is made. A breakpoint is a specific instruction that can be identified by a line number of memory address where your program's instruction are kept. In the terminal text, near the beginning, you can see me use the command to set a breakpoint at line 27 of the source code. Gdb then, after the breakpoint is set, gives you back its interface, after it has just (very metaphorically speaking) shouted "STOP HERE AND DO NOT EXECUTE THIS INTRUCTION 'UNTIL I WANT YOU TO!!" to the processor. Note that the instruction where the breakpoint has been set has not been executed yet. You as the human debugger can then step through each instruction and see its effects on the registers using the "info registers" command, and if you program is very memory orientated, you can then use the "x" command to see the instructions effects on the memory.

The "si" command lets you execute the next instruction of assembly code, and this will be the command that actually does something with your program: actually gets it running, sort-of. This command will and only will execute the next instruction. the "break" command only INITIALLY stops the program from being run fully, but the si instruction is used for you to actually debug the program. Look at the results from executing this command below, when "si" has been typed. what "si" actually means is "step one instruction, and ONLY on instruction". Stepi's cousin is actually Step, but this is mainly for C, where stepping one line of code may be many actual machine instructions.

The "ni" command is used primarily for when your assembly code has any functions outside of the actual source code, or subroutine calls. This kind of programming is something that will be done when you graduate far from a "newbie", but still is something I have not done, but if you know your C, and good for you, you will know at least what I try to talk about. For example, say your program has an instruction that calls some other piece of code that has already been thoroughly debugged, like a subroutine call, you may not want to go through that function as it has already been debugged, so, before you enter that subroutine, you would execute the "ni" command, where you step over the subroutine, executing the whole of it along the way, but where you do not have to debug it. If you wanted to enter that subroutine, you would then execute "si" which enters it and will allow you to debug the subroutine, one, instruction, at, a, time.

The next command I will briefly explain will save a fair bit of time, and allow you to display something, be it the contents of a register or an address in memory, but this will be printed to screen

EVERY time you execute on of the "si" or "ni" commands. such a usage woulod be where you would print the contents of the instruction pointer's address in memory, and interpret the contents of that address as machine instructions in assembly. This command will show you the next instruction to be carried out by "si" in the future, which is very useful and can be done like this:

display/i $eip (note the "i" which interprets the contents of the memory address as Instructions)

Debugging an assembly program may be like seeing your assembly code again, but in action, but with assembly being assembly, why do we need to, because assembly shows us what it does anyway with registers and memory? The truth is, debugging will definitely be used for BIG programs in assembly, and for ones dealing with the stack and also ones dealing with complex arrays. For any case, some stuff assembly does not hide, but by no means makes it clear to you as a programmer by simply letting you view its source code. For example, the little endian nature of x86 processors may stab you in the back and laugh in you face, and debugging and examining the contents of memory and registers may help by far, especially with BIG assembly programs that actually do cool stuff.

But before anyone cries and realises their program doesn't have an exit code and you went straight from the last assembly instruction and typed "si", and good old Linux has terminated your program and shoved a segmentation error in front of your face, I will try to explain why. What this means is that your last instruction that doesn't exit the program is more of a trial and error with some assembly test code (good, this will honestly help you be creative with your programs!) and your last instruction was, for example, in Intel syntax, "mov eax, ebx", or something else useful, when you execute that instruction, the next instruction in memory is not that of your program, because all of it has already run, and your naughty program is trying to get the instruction off of some other programs memory, which is highly not allowed! Linux however will just terminate your program and shove it right back at you, no harm done. But sometimes, your last instruction may not be executed after the segmentation fault, disallowing you to look at the effects of it in gdb, when that instruction may be a really interesting one, so, nothing is to the rescue!! What I really mean is "nop" this is a very useful instruction that is used for debugging, and primarily does nothing with anything for or in your program (it technically does this: "xchg eax, eax", or something like that). Anyway, this instruction can be used as a means of adding padding to the beginning and end of your code, as Jeff Duntemann has done in his program below, and as you can see, his comment states that it keeps gdb happy! What this does is stop your program counter going off the cliff of your program and from being naughty, and gives you time to execute that precious last instruction without having to add the exit syscalls and with the assurance that that you have time to examine the last instruction's effects and kill the program with "kill" and stop debugging.

Below I have put in a copied session in gdb from one of the first trial and error style debugs of this source code (yes, I am inexperienced). As I will leave you time to have a look, I will say bye for now and hope I have done something helpful.

someone@debian-box:~/Documents/assembly/eatsyscall$ gdb ./eatsyscall
GNU gdb (Debian 7.7.1+dfsg-5) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.

```
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./eatsyscall...done.
(gdb) list
1       ;  Executable name : EATSYSCALL
2       ;  Version         : 1.0
3       ;  Created date    : 1/7/2009
4       ;  Last update     : 2/18/2009
5       ;  Author          : Jeff Duntemann
6       ;  Description     : A simple program in assembly for Linux, using NASM 2.05,
7       ;    demonstrating the use of Linux INT 80H syscalls to display text.
8       ;
9       ;  Build using these commands:
10      ;    nasm -f elf -g -F stabs eatsyscall.asm
(gdb)
11      ;    ld -o eatsyscall eatsyscall.o
12      ;
13
14      SECTION .data                   ; Section containing initialised data
15
16              EatMsg: db "Eat at Joe's!",10
17              EatLen: equ $-EatMsg
18
19      SECTION .bss                    ; Section containing uninitialized data
20
(gdb)
21      SECTION .text                   ; Section containing code
22
23      global  _start                  ; Linker needs this to find the entry point!
24
25      _start:
26              nop                     ; This no-op keeps gdb happy...
27              mov eax,4               ; Specify sys_write call
28              mov ebx,1               ; Specify File Descriptor 1: Standard Output
29              mov ecx,EatMsg                  ; Pass offset of the message
30              mov edx,EatLen                  ; Pass the length of the message
(gdb)
31              int 80H                         ; Make kernel call
32
33              MOV eax,1               ; Code for Exit Syscall
34              mov ebx,0               ; Return a code of zero
35              int 80H                         ; Make kernel call
36
37
38
39
40
(gdb) break 27
Breakpoint 1 at 0x8048081: file ./eatsyscall.asm, line 27.
(gdb) list
```

```
41
42
43
44
45
46
47
48
49
50
(gdb) display/i $eip
(gdb) run
Starting program: /home/someone/Documents/assembly/eatsyscall/eatsyscall

Breakpoint 1, 0x08048081 in _start ()
1: x/i $eip
=> 0x8048081 <_start+1>:    mov    $0x4,%eax
(gdb) si
0x08048086 in _start ()
1: x/i $eip
=> 0x8048086 <_start+6>:    mov    $0x1,%ebx
(gdb) info registers
eax            0x44
ecx            0x00
edx            0x0         0
ebx            0x0         0
esp            0xffffd420  0xffffd420
ebp            0x0         0x0
esi            0x0 0
edi            0x0 0
eip            0x8048086   0x8048086 <_start+6>
eflags         0x202       [ IF ]
cs             0x23        35
ss             0x2b        43
ds             0x2b        43
es             0x2b        43
fs             0x0  0
gs             0x0 0
(gdb) x
0x804808b <_start+11>:      0x0490a4b9
(gdb) si
0x0804808b in _start ()
1: x/i $eip
=> 0x804808b <_start+11>:  mov    $0x80490a4,%ecx
(gdb) info registers
eax            0x44
ecx            0x00
edx            0x0         0
ebx            0x1         1
esp            0xffffd420  0xffffd420
ebp            0x0         0x0
esi            0x0 0
```

```
edi            0x0 0
eip            0x804808b 0x804808b <_start+11>
eflags         0x202      [ IF ]
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
fs             0x0  0
gs             0x0 0
(gdb) x help
No symbol "help" in current context.
(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".
(gdb) si
0x08048090 in _start ()
1: x/i $eip
=> 0x8048090 <_start+16>:  mov    $0xe,%edx
(gdb) info registers
eax            0x44
ecx            0x80490a4 134516900
edx            0x0        0
ebx            0x1        1
esp            0xffffd420 0xffffd420
ebp            0x0        0x0
esi            0x0  0
edi            0x0 0
eip            0x8048090 0x8048090 <_start+16>
eflags         0x202      [ IF ]
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
fs             0x0  0
gs             0x0 0
(gdb) x/10xb
0x8048095 <_start+21>:       0xcd    0x80    0xb8    0x01    0x00    0x00    0x00    0xbb
0x804809d <_start+29>:       0x00    0x00
(gdb) x/10xb 0x80490a4
0x80490a4 <EatMsg>:          0x45    0x61    0x74    0x20    0x61    0x74    0x20    0x4a
0x80490ac:     0x6f    0x65
```

```
(gdb) x/10cb 0x80490a4
0x80490a4 <EatMsg>:         69 'E'   97 'a'    116 't'   32 ' '    97 'a'    116 't'   32 ' '    74 'J'
0x80490ac:     111 'o'  101 'e'
(gdb) where
#0  0x08048090 in _start ()
(gdb) si
0x08048095 in _start ()
1: x/i $eip
=> 0x8048095 <_start+21>:  int    $0x80
(gdb) info registers
eax            0x44
ecx            0x80490a4134516900
edx            0xe14
ebx            0x1           1
esp            0xffffd420 0xffffd420
ebp            0x0           0x0
esi            0x0 0
edi            0x0 0
eip            0x8048095 0x8048095 <_start+21>
eflags         0x202       [ IF ]
cs             0x23          35
ss             0x2b          43
ds             0x2b          43
es             0x2b          43
fs             0x0  0
gs             0x0 0
(gdb) x/15is 0x8048095
0x8048095 <_start+21>:       "\270\001"
0x804809a <_start+26>:       ""
0x804809b <_start+27>:       ""
0x804809c <_start+28>:       "\273"
0x804809e <_start+30>:       ""
0x804809f <_start+31>:       ""
0x80480a0 <_start+32>:       ""
0x80480a1 <_start+33>:       "
0x80480a4:     "Eat at Joe's!\n"
0x80480b3:     ""
0x80480b4:     "\001"
0x80480b6:     ""
0x80480b7:     ""
0x80480b8:     ""
0x80480b9:     ""
(gdb) x/15ib 0x8048095
=> 0x8048095 <_start+21>:  int    $0x80
   0x8048097 <_start+23>:    mov    $0x1,%eax
   0x804809c <_start+28>:    mov    $0x0,%ebx
   0x80480a1 <_start+33>:    int    $0x80
   0x80480a3:  add    %al,0x61(%ebp)
   0x80480a6:  je     0x80480c8
   0x80480a8:  popa
   0x80480a9:  je     0x80480cb
   0x80480ab:  dec    %edx
```

```
   0x80480ac:  outsl  %ds:(%esi),(%dx)
   0x80480ad:  gs
   0x80480ae:  daa
   0x80480af:  jae    0x80480d2
   0x80480b1:  or     (%eax),%al
   0x80480b3:  add    %al,(%ecx)
(gdb) si
Eat at Joe's!
0x08048097 in _start ()
1: x/i $eip
=> 0x8048097 <_start+23>:  mov    $0x1,%eax
(gdb) info registers
eax            0xe14
ecx            0x80490a4134516900
edx            0xe14
ebx            0x1          1
esp            0xffffd420 0xffffd420
ebp            0x0          0x0
esi            0x0 0
edi            0x0 0
eip            0x8048097 0x8048097 <_start+23>
eflags         0x202        [ IF ]
cs             0x23         35
ss             0x2b         43
ds             0x2b         43
es             0x2b         43
fs             0x0  0
gs             0x0 0
(gdb) si
0x0804809c in _start ()
1: x/i $eip
=> 0x804809c <_start+28>:  mov    $0x0,%ebx
(gdb) si
0x080480a1 in _start ()
1: x/i $eip
=> 0x80480a1 <_start+33>:  int    $0x80
(gdb) si
[Inferior 1 (process 3285) exited normally]
(gdb) kill
The program is not being run.
(gdb) quit
someone@debian-box:~/Documents/assembly/eatsyscall$ ./eatsyscall
Eat at Joe's!
someone@debian-box:~/Documents/assembly/eatsyscall$
```