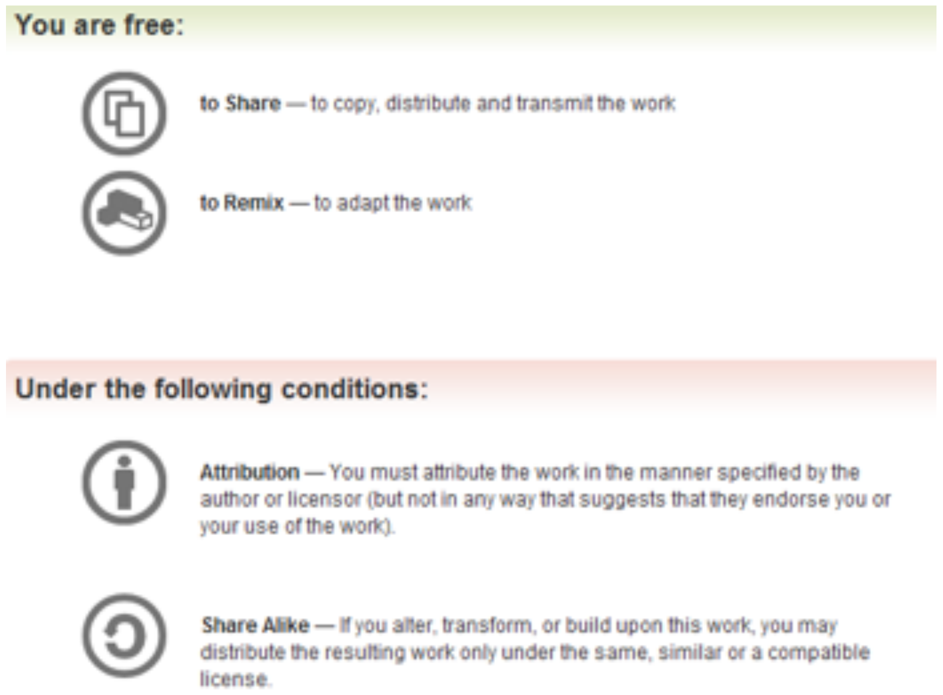
Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2014

xkovah at gmail

All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Attribution condition: You must indicate that derivative work
"Is derived from Xeno Kovah's 'Intro x86-64’ class, available at http://OpenSecurityTraining.info/IntroX86-64.html”

Attribution condition: You must indicate that derivative work
"Is derived from Xeno Kovah's ‘Intro x86-64’ class, available at http://OpenSecurityTraining.info/IntroX86-64.html"

gcc - GNU project C and C++ compiler

- Available for many *nix systems (Linux/BSD/OSX/Solaris)
- Supports many other architectures besides x86
- Some C/C++ options, some architecture-specific options
  - Main option we care about is building debug symbols. Use “-ggdb” command line argument.
- Basically all of the VisualStudio options in the project properties page are just fancy wrappers around giving their compiler command line arguments. The equivalent on *nix is for to developers create “makefile”s which are a configuration or configurations which describes which options will be used for compilation, how files will be linked together, etc. We won’t get that complicated in this class, so we can just specify command line arguments manually.

Book p. 53

More details in the manpage, available by typing “man gcc” on a *nix system

gcc basic usage

- gcc -o <output filename> <input file name>
  - gcc -o hello hello.c
  - If -o and output filename are unspecified, default output filename is “a.out” (for legacy reasons)
- So we will be using:
  - gcc -ggdb -o <filename> <filename>.c
  - gcc -ggdb -o Example1 Example1.c

objdump - display information from object files

- Where ”object file” can be an intermediate file created during compilation but before linking, or a fully linked executable
  - For our purposes means any ELF file - the executable format standard for Linux
- The main thing we care about is -d to disassemble a file.
- Can override the output syntax with “-M intel”
  - Good for getting an alternative perspective on what an instruction is doing, while learning AT&T syntax

Book p. 63

More details in the manpage, available by typing “man objdump” on a *nix system
Aside: the equivalent on Mac OS X is otool.

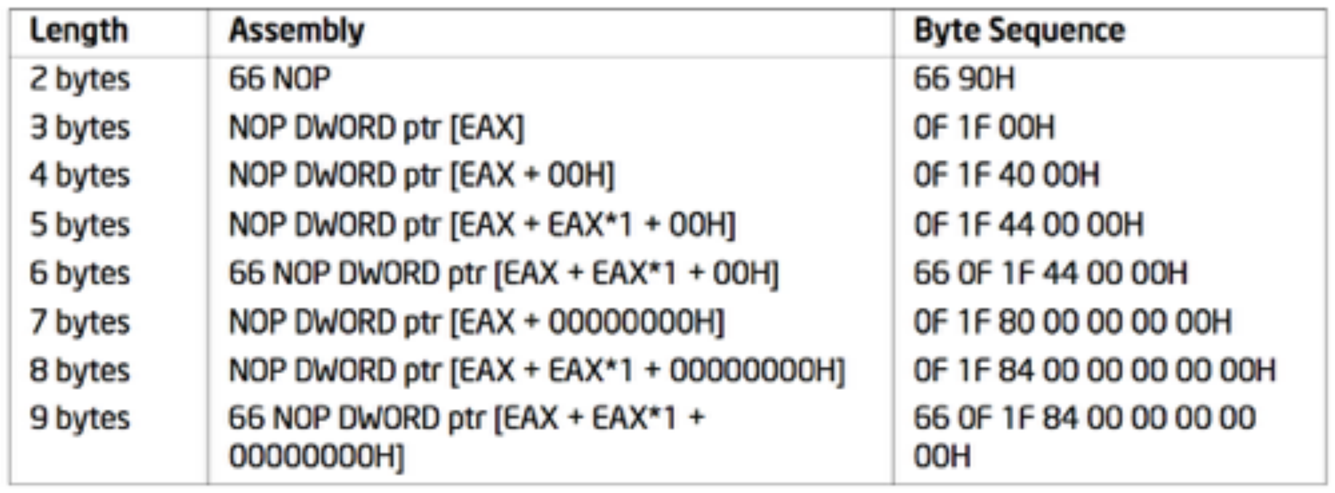
objdump -d hello

```
hello:     file format elf64-x86-64

Disassembly of section .init:

00000000004003e0 <_init>:
  4003e0:48 83 ec 08                sub    $0x8,%rsp
  4003e4:48 8b 05 0d 0c 20 00       mov    0x200c0d(%rip),%rax # 600ff8 <_DYNAMIC+0x1d0>
  4003eb:48 85 c0                   test   %rax,%rax
  4003ee:74 05                      je     4003f5 <_init+0x15>
  4003f0:e8 3b 00 00 00             callq  400430 <__gmon_start__@plt>
  4003f5:48 83 c4 08                add    $0x8,%rsp
  4003f9:c3                         retq
…
000000000040052d <main>:
  40052d:55                         push   %rbp
  40052e:48 89 e5                   mov    %rsp,%rbp
  400531:bf d4 05 40 00             mov    $0x4005d4,%edi
  400536:e8 d5 fe ff ff             callq  400410 <puts@plt>
  40053b:b8 34 12 00 00             mov    $0x1234,%eax
  400540:5d                         pop    %rbp
  400541:c3                         retq
  400542:66 2e 0f 1f 84 00 00       nopw   %cs:0x0(%rax,%rax,1)
  400549:00 00 00
  40054c:0f 1f 40 00                nopl   0x0(%rax)
…
```

Wait…whut? “nopl/nopw”?

“There are more [NOPS] under heaven and earth, Horatio, than are dreamt of in your philosophy” :)

Table 4-12. Recommended Multi-Byte Sequence of NOP Instruction



Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

GCC is clearly using some multi-byte NOPs to pad the end of main() so that the next function starts on a 0x10-aligned boundary

objdump -d -M intel hello

```
hello:     file format elf64-x86-64

Disassembly of section .init:

00000000004003e0 <_init>:
  4003e0:       48 83 ec 08             sub    rsp,0x8
  4003e4:       48 8b 05 0d 0c 20 00    mov    rax,QWORD PTR [rip+0x200c0d] # 600ff8
   <_DYNAMIC+0x1d0>
  4003eb:       48 85 c0                test   rax,rax
  4003ee:       74 05                   je     4003f5 <_init+0x15>
  4003f0:       e8 3b 00 00 00          call   400430 <__gmon_start__@plt>
  4003f5:       48 83 c4 08             add    rsp,0x8
  4003f9:       c3                      ret
…
000000000040052d <main>:
  40052d:       55                      push   rbp
  40052e:       48 89 e5                mov    rbp,rsp
  400531:       bf d4 05 40 00          mov    edi,0x4005d4
  400536:       e8 d5 fe ff ff          call   400410 <puts@plt>
  40053b:       b8 34 12 00 00          mov    eax,0x1234
  400540:       5d                      pop    rbp
  400541:       c3                      ret
  400542:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
  400549:       00 00 00
  40054c:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]
…
```

hexdump & xxd & strings

- Sometimes useful to look at a hexdump to see opcodes/operands or raw file format info
- hexdump, hd - ASCII, decimal, hexadecimal, octal dump
  - hexdump -C for “canonical” hex & ASCII view
  - Use for a quick peek at the hex
- xxd - make a hexdump or do the reverse
  - Use as a quick and dirty hex editor
  - xxd hello > hello.dump
  - Edit hello.dump
  - xxd -r hello.dump > hello

– strings - dump out all the ASCII strings for a binary

More details in the manpage, available by typing “man hexdump” and “man xxd” on a *nix system

GDB - the GNU debugger

- A command line debugger - quite a bit less user-friendly for beginners.
  - There are wrappers such as ddd but I tried them back when I was learning asm and didn't find them to be helpful. YMMV
- Syntax for starting a program in GDB in this class:
  - gdb <program name> -x <command file>
  - gdb Example1 -x myCmds

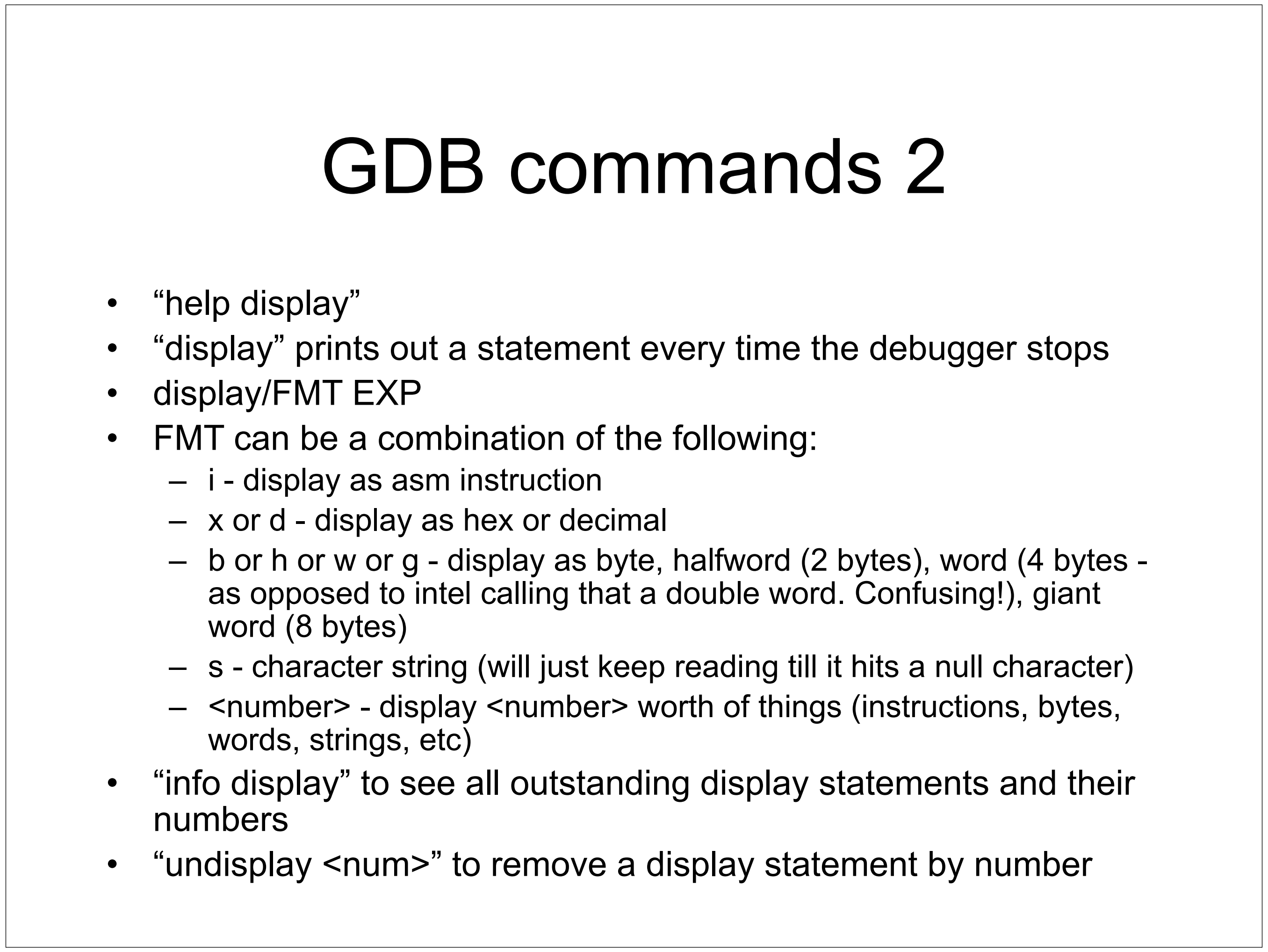
Book p. 57

About GDB -x <command file>

- Somewhat more memorable long form is “--command=<command file>”
- <command file> is a plaintext file with a list of commands that GDB should execute upon starting up. Sort of like scripting the debugger.
- Absolutely **essential** to making GDB reasonable to work with for extended periods of time (I used GDB for many years copying and pasting my command list every time I started GDB, so I was super ultra happy when I found this option)

GDB commands

- “help” - internal navigation of available commands
- “run” or “r” - run the program
- “r <argv>” - run the program passing the arguments in <argv>
  - I.e. for Example 2 “r 1 2” would be what we used in windows



GDB commands 2

- “help display”
- “display” prints out a statement every time the debugger stops
- display/FMT EXP
- FMT can be a combination of the following:
  - i - display as asm instruction
  - x or d - display as hex or decimal
  - b or h or w or g - display as byte, halfword (2 bytes), word (4 bytes - as opposed to intel calling that a double word. Confusing!), giant word (8 bytes)
  - s - character string (will just keep reading till it hits a null character)
  - <number> - display <number> worth of things (instructions, bytes, words, strings, etc)
- “info display” to see all outstanding display statements and their numbers
- “undisplay <num>” to remove a display statement by number

Full list of format specifiers

http://sources.redhat.com/gdb/current/onlinedocs/gdb.html#SEC71

GDB commands 3

- “x/FMT EXP” - x for “Examine memory” at expression
  - Always assumes the given value is a memory address, and it dereferences it to look at the value **at** that memory address
- “print/FMT EXP” - print the value of an expression
  - Doesn’t try to dereference memory
- Both commands take the same type of format specifier as display
- Example:

```
(gdb) x/x $rbp
0x7fffffffde70:   0x00000000
(gdb) print/x $rbp
$1 = 0x7fffffffde70
(gdb) x/x $rbx
0x0: Cannot access memory at address 0x0
(gdb) print/x $rbx
$2 = 0x0
```

Full list of format specifiers
http://sources.redhat.com/gdb/current/onlinedocs/gdb.html#SEC71
From "help x":
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) and s(string).
Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).

GDB commands 4

- For all breakpoint-related commands see “help breakpoints”
- “break” or “b” - set a breakpoint
  - With debugging symbols you can do things like “b main”. Without them you can do things like

    “b *<address>” to break at a given memory address.
  - Note: gdb’s interpretation of where a function begins may exclude the function prolog like “push ebp”…
- “info breakpoints” or “info b” - show currently set breakpoints
- "delete <num> - deletes breakpoint number <num>, where <num> came from "info breakpoints"

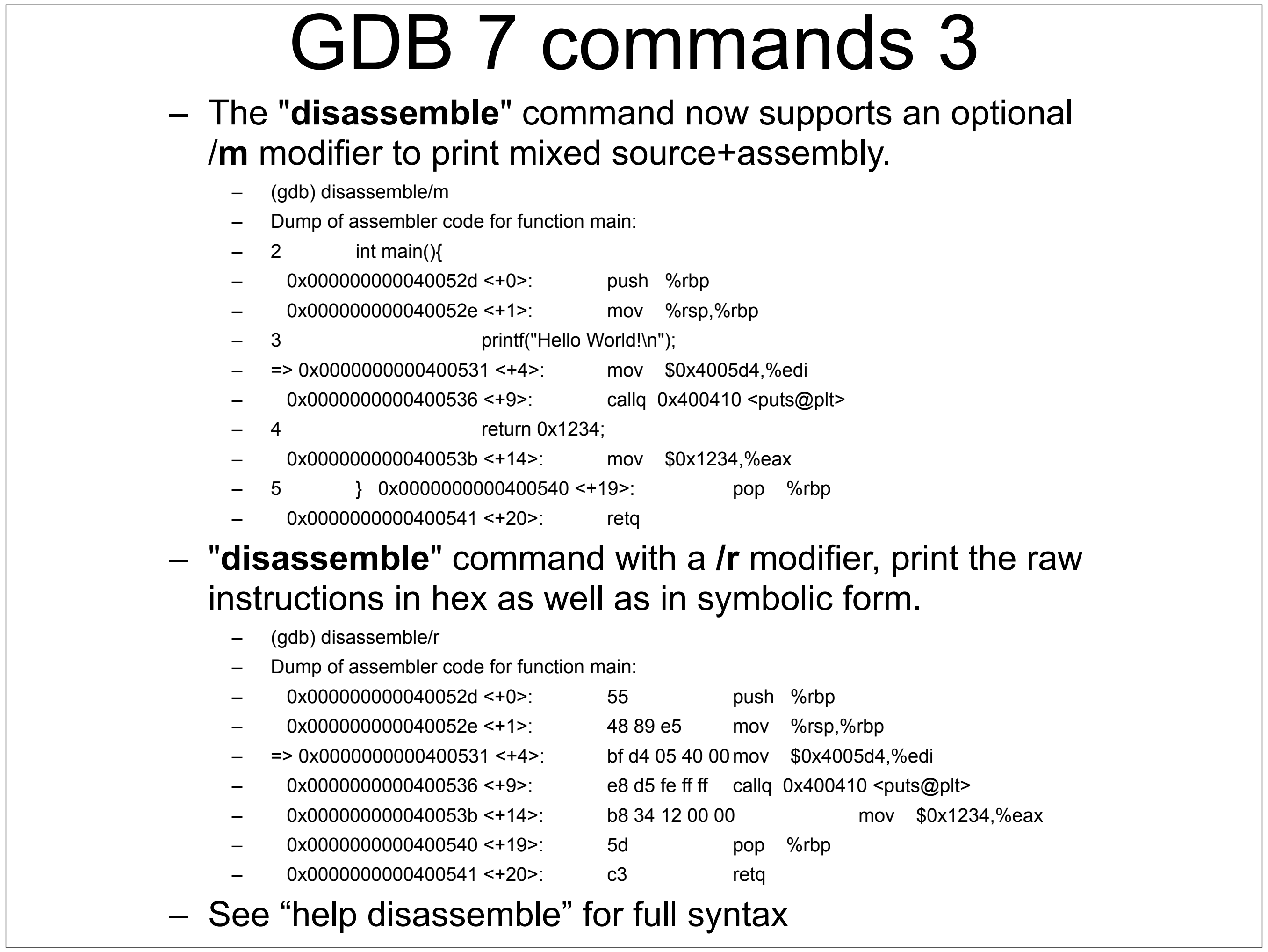
GDB 7 commands

- New for GDB 7, released Sept 2009
  - Thanks to Dave Keppler for notifying me of the availability of these new commands (even if they don’t work in this lab ;))
  - **reverse-step** ('rs') -- Step program backward until it reaches the beginning of a previous source line
  - **reverse-stepi** -- Step backward exactly one instruction
  - **reverse-continue** ('rc') -- Continue program being debugged but run it in reverse
  - **reverse-finish** -- Execute backward until just before the selected stack frame is called

GDB 7 commands 2

- **reverse-next** ('rn') -- Step program backward, proceeding through subroutine calls.
- **reverse-nexti** ('rni') -- Step backward one instruction, but proceed through called subroutines.
- **set exec-direction (forward/reverse)** -- Set direction of execution. All subsequent execution commands (continue, step, until etc.) will run the program being debugged in the selected direction.

Currently can’t use /r and /m together



GDB 7 commands 3

- The "**disassemble**" command now supports an optional /**m** modifier to print mixed source+assembly.

```
(gdb) disassemble/m
Dump of assembler code for function main:
2       int main(){
   0x000000000040052d <+0>:     push   %rbp
   0x000000000040052e <+1>:     mov    %rsp,%rbp
3                   printf("Hello World!\n");
=> 0x0000000000400531 <+4>:     mov    $0x4005d4,%edi
   0x0000000000400536 <+9>:     callq  0x400410 <puts@plt>
4                   return 0x1234;
   0x000000000040053b <+14>:    mov    $0x1234,%eax
5       }   0x0000000000400540 <+19>:       pop    %rbp
   0x0000000000400541 <+20>:    retq
```

- "**disassemble**" command with a **/r** modifier, print the raw instructions in hex as well as in symbolic form.

```
(gdb) disassemble/r
Dump of assembler code for function main:
   0x000000000040052d <+0>:     55              push   %rbp
   0x000000000040052e <+1>:     48 89 e5        mov    %rsp,%rbp
=> 0x0000000000400531 <+4>:     bf d4 05 40 00  mov    $0x4005d4,%edi
   0x0000000000400536 <+9>:     e8 d5 fe ff ff  callq  0x400410 <puts@plt>
   0x000000000040053b <+14>:    b8 34 12 00 00          mov    $0x1234,%eax
   0x0000000000400540 <+19>:    5d              pop    %rbp
   0x0000000000400541 <+20>:    c3              retq
```

- See “help disassemble” for full syntax

Currently can’t use /r and /m together

initial GDB commands file

- display/10i $rip
- display/x $rax
- display/x $rbx
- display/x $rcx
- display/x $rdx
- display/x $rdi
- display/x $rsi
- display/x $r8
- display/x $r9
- display/x $rbp
- display/16xg $rsp
- break main

```
(gdb) r
Starting program: /mnt/hgfs/vmshare/IntroToAsm_code_for_class/HelloWorld/hello

Breakpoint 1, main () at Hello.c:3
3                    printf("Hello World!\n");
11: x/16xg $rsp
0x7fffffffde70:      0x0000000000000000      0x00007ffff7a35ec5
0x7fffffffde80:      0x0000000000000000      0x00007fffffffdf58
0x7fffffffde90:      0x0000000100000000      0x000000000040052d
0x7fffffffdea0:      0x0000000000000000      0x39f79df94699a772
0x7fffffffdeb0:      0x0000000000400440      0x00007fffffffdf50
0x7fffffffdec0:      0x0000000000000000      0x0000000000000000
0x7fffffffded0:      0xc6086206fb99a772      0xc60872bffa63a772
0x7fffffffdee0:      0x0000000000000000      0x0000000000000000
10: /x $rbp = 0x7fffffffde70
9: /x $r9 = 0x7ffff7dea560
8: /x $r8 = 0x7ffff7dd4e80
7: /x $rsi = 0x7fffffffdf58
6: /x $rdi = 0x1
5: /x $rdx = 0x7fffffffdf68
4: /x $rcx = 0x0
3: /x $rbx = 0x0
2: /x $rax = 0x40052d
1: x/10i $rip
=> 0x400531 <main+4>:     mov    $0x4005d4,%edi
   0x400536 <main+9>:     callq  0x400410 <puts@plt>
   0x40053b <main+14>:    mov    $0x1234,%eax
   0x400540 <main+19>:    pop    %rbp
   0x400541 <main+20>:    retq
```

Source code line printed here if source is available (e.g. compiled with -ggbd)

Stepping

- “stepi” or “si” - steps one asm instruction at a time
  - Will always “step into” subroutines
- “nexti” or “ni” - steps over one asm instruction at a time
  - Will always “step over” subroutines
- “step” or “s” - steps one source line at a time (if no source is available, works like stepi)
- “until” or “u” - steps until the next source line, not stepping into subroutines
  - If no source available, this will work like a stepi that will “step over” subroutines
- “finish” - steps out of the current function

GDB misc commands

- “set disassembly-flavor intel” - use intel syntax rather than AT&T
  - Again, not using now, just good to know
- “continue” or “c” - run until you hit another breakpoint or the program ends
- “backtrace” or “bt” - print a trace of the call stack, showing all the functions which were called before the current function

Lab time:
Running all the examples we ran earlier with Windows/VS with Linux/GDB