Debugging Assembly Code with *gdb*

*gdb* is the GNU source-level debugger that is standard on Linux (and many other Unix) systems. It can be used both for programs written in high-level languages like C and C++ and for assembly code programs; this document concentrates on the latter.

For detailed information on the use of *gdb*, consult the documentation. Unfortunately, this is not in form of a `man` page, and even the `info` page does not seem to be installed on the Linuxlab machines; the best thing is to read it on the web at [http://sourceware.org/gdb/current/onlinedocs/gdb/](http://sourceware.org/gdb/current/onlinedocs/gdb/).

*gdb* will work in an ordinary terminal window, and this is fine for debugging assembly code. For use with higher-level source code, it is more convenient to use *gdb* from within the *emacs* editor (a good one to learn!) or using a graphical front-end like *ddd*. The basic commands remain the same.

To use *gdb* with high-level language programs, you should compile with the `-g` option. This will include information in the object file to relate it back to the source file. When assembling `.s` files to be debugged, the `-g` option is not necessary, but it is harmless.

```
gcc -m64 -g -o foo fooDriver.c fooRoutine.s
```

To invoke the debugger on `foo`, type

```
gdb foo
```

This loads program `foo` and brings up the *gdb* command line interpreter, which then waits for you to type commands. Program execution doesn’t begin until you say so.

Here are some useful commands. Many can be abbreviated, as shown. Hitting return generally repeats the last command, sometimes advancing the current location.

```
h[elp] [keyword]                      Displays help information.
r[un] [args]                        Begin program execution. If the program normally takes command-line arguments (e.g., `foo hi 3`), you should specify them here (e.g., `run hi 3`).
b[reak] [address]                   Set a breakpoint at the specified address (or at the current address if none specified). Addresses can be given symbolically (e.g., `foo`) or numerically (e.g., `0x10a38`). When execution reaches a breakpoint, you are thrown back into the *gdb* command line interpreter.
c[ontinue]                        Continue execution after stopping at a breakpoint.
i[nfo] b[reak]                    Display numbered list of all breakpoints currently set.
```
d[ele]te b[reakpoints] number
  Delete specified breakpoint number.

p[rint]/[format] expr
  Print the value of an expression using the specified format (decimal if unspeci-
  fied). Expressions can involve program variables or registers, which are specified
  using a $ rather than a % sign. Useful formats include:

  • d decimal
  • x hex
  • t binary
  • f floating point
  • i instruction
  • c character

  For example, to display the value of register %rdi in decimal, type p/x $rdi. Note
  that you need to use the 64-bit (%r) forms of register names. To see the
  value of the current program counter, type p/x $rip.

i[nfo] r[egisters] register
  An alternative way to print the value of a register (or, if none is specified, of all
  registers) in hex and decimal. Specify the register without a leading %, e.g., rdi.

x/[count]/[format] [address]
  Examine the contents of a specified memory address, or the current address if
  none specified. If count is specified, displays specified number of words. Ad-
  dresses can be symbolic (e.g., main) or numeric (e.g., 0x10a44). Formats are
  as for print. Particularly useful for printing the program text, e.g., x/100i
  foo disassembles and prints 100 instructions starting at foo.n

disas[semble] address [address]
  Another way to print the assembly program text surrounding an address, or be-
  tween two addresses.

set var = expr
  Set specified register or memory location to value of expression. Examples: set
  $rdi=0x456789AB or set myVar=myVar*2.

s[tep]i
  Execute a single instruction and then return to the command line interpreter.

n[ext]i
  Like stepi, except that if the instruction is a subroutine call, the entire subrou-
  tine is executed before control returns to the interpreter.

whe[re]
  Show current activation stack.

q[uit]
  Exit from gdb.