# Debugging Assembly Code with gdb

gdb is the GNU source-level debugger that is standard on the CS department sparcs and on linux 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; instead, it may be read using the info command. Type

```
/pkgs/gnu/bin/info gdb
```

and then type h to get an introduction on how to use info. 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 xxgdb or /pkgs/gnu/bin/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, -g option is not necessary, but it is harmless.

```
gcc -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[elete] b[reakpoints]` *number*
: Delete specified breakpoint number.

`p[rint]`*[/format] expr*
: Print the value of an expression using the specified format (decimal if unspecified). 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 `%i5` in decimal, type `p/x $i5`. To see the value of the current program counter, type `p/x $pc`.

`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., `i4`.

`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. Addresses 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`.

`set` *var* `=` *expr*
: Set specified register or memory location to value of expression. Examples: `set $g4=0x456789AB` or `set myVar=myVar*2`.

`si`

Execute a single instruction and then return to the command line interpreter.

`ni`

Like `si`, except that if the instruction is a subroutine call, the entire subroutine is executed before control returns to the interpreter.

`where`

Show current activation stack.

`q[uit]`

Exit from `gdb`.