Debugging assembly programs

When your assembly program crashes, the exact problem is often tough to diagnose. In this note, I'll discuss some common assembly coding bugs, and then some debugging tools/ techniques.

Before I start, I want to reiterate the most important debugging technique of all - getting a good start. You will benefit greatly by being very organized and coding small chunks of your program at a time.

The sections are: 1. Common bugs, 2. Good, ole printf(), 3. Using gdb, and 4. Using Netbeans.

1. Common bugs

Here are some bugs that I have run into that are specific to assembly programming.

- **Screwing up the stack** - Perhaps you forget to place the entry/exit guards on a function. If you have two ret statements in your functions (I almost never do this), did you place the exit guard on each to
- **Screwing up the stack 2** - Did you make space for your local variables by decrementing `%esp`? If not, you may wreck your stack within your function.
- **Screwing up the stack 3** - Did you restore your stack after calling a function by adding to `%esp`?
- **Using the wrong parameter or local variable** - Our parameter and local variable names are gone in assembly, so it's common to use the wrong frame pointer offset.
- **Wrong name** - Are you using the wrong name? Do you want a value or an address: $N or N? Often, both are acceptable to the assembler, but can have negative effects on your program working correctly.
- **Registers changed by a function** - If you're storing a value in a register, that value may be erased when you call another function. An easy example is `%eax`. If you use `%eax` to store a value that you need, that value will be replaced by the function's return value. The same is true with other registers. If you need a value to survive a function call, then 1) use a local variable, or 2) push the register value onto the stack before the call and pop it off after. I usually just use a local variable.

An aside - If you mess up your stack, the impact (your program crashing or giving bad results) will likely be many lines after the offending code. This makes these bugs especially tricky to find and squash.

A mysterious assembler error is to use two memory references in one command. That's an Intel assembly no-no. Move one of your memory values to a register and proceed.

2. Good, ole printf()

I wonder...

- Did that function really get called?
- Are my function parameters correct?
- Did that local variable get changed?

You're running your program and it's not working. Let your investigation begin! You can answer questions like the ones above using debugging `printf()` calls to questionable code. Print "I'm in function X()" to show that function X got called. Print its parameters and local variables to verify that they have the values you think they should have.

Once you've fixed your problem, you can either remove any debugging `printf()` calls or comment them out.

3. Using gdb

First, you must compile your assembly code with the -g option to fully use the debugger. This option directs gcc to leave many user names in the executable so they can be referenced within the debugger.

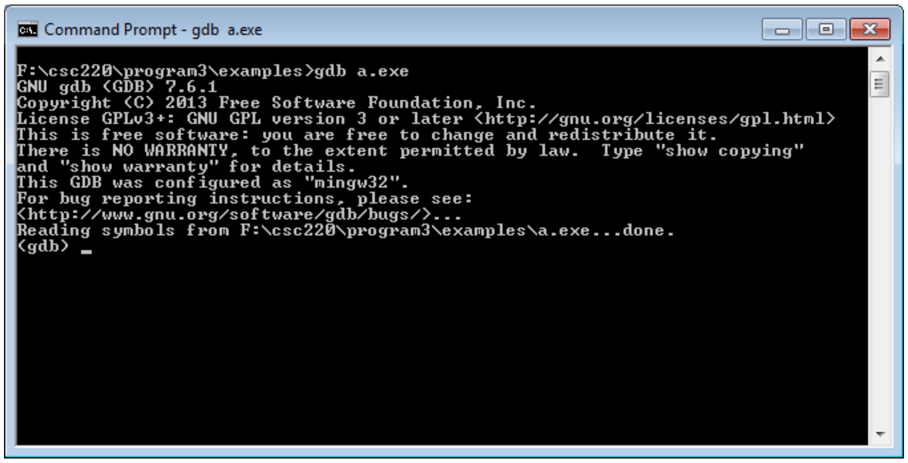
This command creates an `a.exe` suitable for debugging:

```
gcc -g bill.s
```

OK, start gdb for your -g executable and we're off!

```
gdb a.exe
```

Here's my PC console window for this. `(gdb)` is the gdb command prompt.



```
F:\csc220\program3\examples>gdb a.exe
GNU gdb (GDB) 7.6.1
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from F:\csc220\program3\examples\a.exe...done.
(gdb) _
```

Here are some common/useful gdb commands to control the execution of your program:

- **break** - Stop execution of your program at the specified label. Example: `break main`. This will stop your program right away.
- **run** - Run your program. It will stop at the first breakpoint that you have set.
- **next** - execute one line
- **step** - step gdb into a call, we move “into” a function
- **continue** - resume running program after a breakpoint
- **list** - List the code at the current point in your program
- **list <line_num>** - list code at line line_num
- **disassemble** - similar to list, but includes memory addresses for code

You can use gdb to view the state of registers and memory as your program executes:

- **info register** - prints the value of all registers
- **print $<register>** - prints the value of one register… notice the $ syntax. Example: `print $edx`
- **print/x $<register>** - print the register value in hex
- **print/t $<register>** - print the register value in binary
- **x/<format> <address>** - examines contents of memory at <address>. The <format> controls the output style: x for hex, d for decimal, etc. A couple of examples here:
    - `x ($ebp+8)` - examines/prints value of the 1st function parameter
    - `x/d ($ebp-4)` - examines/prints value of the 1st local variable, as a decimal
    - `x $esp` - examines/prints the value at top of the stack

Let’s show off a bit:

- You can use **set** to change the value of a register while you are running. Example: `set $edx = 17`
- You can use **print** to see frame pointer values like function parameters and local variables. Example: `print *(int)($ebp+8)` shows the integer value at `ebp`+8, which is the first function parameter. Example: `print *(int)($ebp-4)` would show the value of the first local variable.

Finally... Enter **quit** to leave gdb. There is a **help** command available as well. Most commands can be abbreviated: “q” means “quit”, “n” is “next” and so on.

Here are a couple of other quick summaries of using gdb for assembly code:

- www.ece.unm.edu/~jimp/310/nasm/gdb.pdf - some nice examples, print show-offs!
- http://www.csee.umbc.edu/~cpatel2/links/310/nasm/gdb_help.shtml -

4. Using NetBeans

You can debug your assembly program in Netbeans, but it's a little tricky. Personally, I prefer using gdb, but here goes…

1. **Create a new project** - File/New Project. Select C/C++ and C/C++ Application as your choices. Name your project and de-select the automatic creation of main.c.
2. **Add your ASM file** - Right-click on "Source Files" for your new project. Add the *.s files that comprise your assembly program.
3. **Compile, Run** - Compile and run should be pretty much the same. Netbeans will warn you that a console window will show your I/O, rather than a window in Netbeans.
4. **Debug!** - Here are some steps/options:
    a. Set a breakpoint in your program
    b. Debug/Debug project
    c. Next/step works
    d. Right-click and you can see your registers and memory

After this, play! I haven't really played with this much myself.
Thanks, Bill