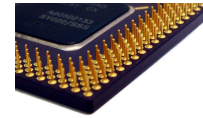COMP-573A Microcomputers

Programming and Debugging
Assembly under Linux
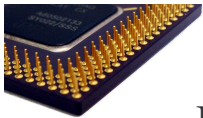slides by Alexandre Denault

---

Simple C Program

Helloworld.c

```
int main(int argc, char *argv[]) {

        printf("Hello World");
}
```

Using GCC to produce Assembly

- From the prompt, type :
  $ **gcc –S HelloWorld.c**
- This will produce a file named HelloWorld.s
- You can produce cleaner assembly code using the –oN flag, N being the optimization level
  $ **gcc –S –o3 HelloWorld.c**

# Assembly Ouput

- Assembly produced by GCC is easy to recognize:

```
.file   "HelloWorld.c"
        .section        .rodata
.LC0:
        .string "Hello World"
        .text
.globl main
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        …
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
.Lfe1:
        .size   main,.Lfe1-main
        .ident  "GCC: (GNU) 3.2.2 20030222 (Red Hat Linux 3.2.2
```

Mixing C and Assembly

- Sometimes, only a particular function needs to be written in assembly.
- GCC allows several way to mix C code and assembly code.
  - Seperate files for C code and assembly code
  - Inlining assembly code directly in the C code
- Adding assembly code to your C code makes your project less portable. Unless special precaution are taken, you can only compile on one architecute.
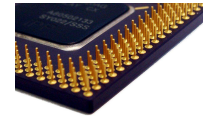
Seperate File Compilation

- C code should be stored in .c files.
- Assembly code should be stored in .s files. (this is case sensitive, .S are different types of files)
- The C code should have the function prototype of any function called from assembly.
- Functions in the assembly code that will called from C should be declared global.
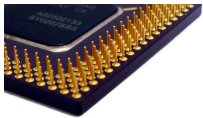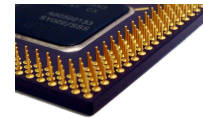
C program that uses assembly

cprogram.c

```
int add(int x, int y);

int main(void) {
        int i,j,k;

        i = 2;
        j = 3;
        k = add(i, j);
        return k;
}
```

Assembly function called from C

add.s

```
        .text

.globl add
        .type   add, @function

add:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        popl    %ebp
        ret
```
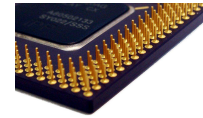
Compiling .s and .c files

- Once both your .c and .s files are ready, you need to compile them togheter.
- This can be done by specifying both filenames to the gcc command.

  gcc cprogram.c add.s

- Don't forget the -g flag if you want to debug your new executable.

Inlining assembly in C code

- If you only want to add a few lines of assembly in your C code, it might be preferable to inlining the assembly code.
- This can be done using the asm function.
- GCC will have a harder time optimizing your code if you use the asm directive.
- You can use the volatile keyword to prevent GCC from optimizing out your code.
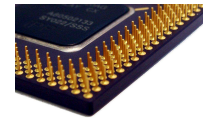
Using the asm function

add.c

```
int add(int x, int y) {

      asm volatile ("movl    12(%ebp), %eax");
      asm volatile ("addl     8(%ebp), %eax");
}
```

- It is possible to have more than one assembly instruction in the asm function.
- More information asm is available on the Internet.

Using GDB to trace program execution

- First, you must compile the executable with the –g flag
  $ gcc HelloWorld.s –g –o HelloWorld
- Please note that there **is** a difference between s and capital S in the extension
- As taken from the man page:
  file.s : Assembly code.
 **file.S :Assembly code which must be preprocessed.**

Using GDB on Solaris 8 (Sparc)

- When working on the Sparc architecture, please try to use the **gcc** and **as** command found in the /usr/local/pkgs if you are planning to use the gdb debugger.
- First, you must compile the assembler with g**as** to produce an object file with debugging information. The **as** normally found on Solaris does not produce debugging information we can use with gdb.

  $ /usr/local/pkgs/binutils-2.14/bin/as
  -gstabs addition.s -o addition.o

- Second, you must use gcc to generate an executable from that object file. You should also add any C files (if needed) that are needed to produce an executable.

  $ /usr/local/pkgs/gcc3.2.2/bin/gcc
  cprogram.c addition.o -o addition
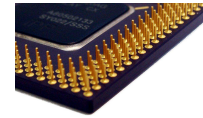
Starting GDB

- Any executable compiled with the –g flag can be used with gdb.

  ```
  $ gdb HelloWorld
  GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
  Copyright 2003 Free Software Foundation, Inc.
  GDB is free software, covered by the GNU General Public
    License, and you are welcome to change it and/or
    distribute copies of it under certain conditions. Type
    "show copying" to see the conditions. There is absolutely
    no warranty for GDB.  Type "show warranty" for details.
    This GDB was configured as "i386-redhat-linux-gnu"...
  (gdb)
  ```
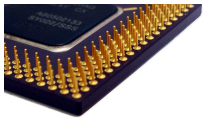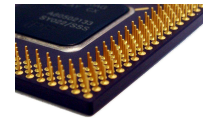
Getting Help

- When using GDB, you can always type in the "help" command to get a list of availible commands.

  ```
  (gdb) help
  ```

- You can also get help on a specific command by typing "help" and the name of that command.
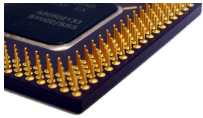
  ```
  (gdb) help breakpoints
  Making program stop at certain points.
  ```

Breakpoints

- First step is set a breakpoint so we can slowly trace the execution of the application.

- This can be done using the "break" command.

  ```
  (gdb) break 1
  Breakpoint 1 at 0x8048328: file HelloWorld.s, line
    1.
  ```
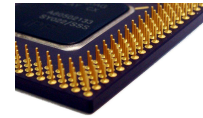
Running step-by-step

- We next to start the application. This is done using the "run" command.

```
(gdb) run
Starting program: /home/adenau/comp573/HelloWorld

Breakpoint 1, main () at HelloWorld.s:10
10              pushl   %ebp
Current language:  auto; currently asm
```

- We can advance to the next instruction using the "nexti" command.

```
(gdb) nexti
11              movl    %esp, %ebp
```

Content of registers

- The "print" command can be used to print out values of registers and variables.
- (gdb) print $ebp
- $1 = (void *) 0xbffff6b8
- The "x" command allows you to examine a block of memory. It has several options.

```
(gdb) x /8xw 0xbffff6b8
0xbffff6b8:     0xbffff6d8      0x42015574      …
0xbffff6c8:     0xbffff70c      0x4001582c…
```
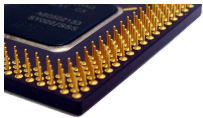
Complete Register Information

- The "info register" command gives us a quick overview of the content of every register.

```
(gdb) info registers
eax            0x1             1
ecx            0x42015554      1107383636
edx            0x40016bc8      1073834952
ebx            0x42130a14      1108544020
esp            0xbffff6b8      0xbffff6b8
ebp            0xbffff6d8      0xbffff6d8
esi            0x40015360      1073828704
edi            0x8048370       134513520
eip            0x8048329       0x8048329
eflags         0x346           838
cs             0x23            35
ss             0x2b            43
ds             0x2b            43
es             0x2b            43
fs             0x0             0
gs             0x33            51
```
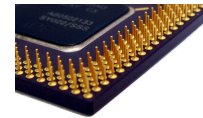
Continual tracing information

- Sometimes, it's more convenient to have trace information printed out after every instruction. This can be done using the display command.

```
(gdb) display $ebp
1: $ebp = (void *) 0xbffff6d8
(gdb) display $esp
2: $esp = (void *) 0xbffff6b8
(gdb) nexti
main () at HelloWorld.s:12
12              subl    $8, %esp
2: $esp = (void *) 0xbffff6b8
1: $ebp = (void *) 0xbffff6d8
```

- The command "undisplay" cancels a display request.
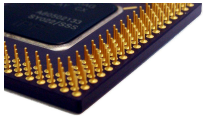
For more information on GDB

- More information about GDB is available on the Internet (tutorials, examples, etc).
- Many other commands allow you to fine tune the debugging process.
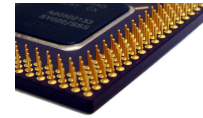
ABI Compliance

- Certain architecture allow a great deal of freedom on how registers and memory are used. (ex: Sparc, PowerPC).
- The System V Application Binary Specification (ABI) are use to establish standards on how system resources are used.
- The ABI usually defines conventions on register usage, stack format, etc.
- For x86, we can avoid having to look at the ABI specification because we have the "The Floppy Textbook".
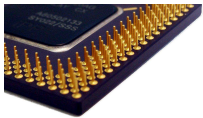
References

- The Floppy Textbook (1999 edition)
- Using the GNU Assembler (gas)
  http://www.gnu.org/manual/gas/html_chapter/as_toc.html
- Man Pages : Gcc and Gdb
- GDB Tutorial
  http://www.cs.princeton.edu/%7Ebenjasik/gdb/gdbtut.html