CS220 – Logic Design
AS03-Developer Tools

• Outline
  – Parts of a Program
  – The CS220 Library
  – The First Program
  – Using the Debugger
  – Using C/C++ Library Functions

• References
  – AL: Using as
  – AL: GAS IA-32 Ass. Language Programming
  – Online: gdb man page, Debugging with gdb
AS03-Developer Tools
The Parts of a Program

- An assembly program (all programs actually) consists of sections. The three most commonly used sections are the text, data, and bss sections.

- The required text section contains the executable code. The optional data section contains data that are assigned an initial value. The optional bss section contains data that are created with an initial zero value (typically data buffers).
The Parts of a Program

- Here are a few examples of data definitions:

```assembly
.sect .data
msg1: .asci "Hello world!"
       .byte 0
msg2: .asciz "Enter an int: 
buffer: .fill 40, 2, 0xFFFF
count: .int 1000
x:    .double 23.987e+12
.sect .bss
strbuf: .fill 80
ival:   .int 0
```
AS03-Developer Tools
The Parts of a Program

- Note that the `.section` directive is used to indicate the start of a section.

- Although the same directives that are used to initialize data in the `data` section (.int, .float, etc.) can be used in the `bss` section, they only allocate space in the `bss` section. All space in the `bss` section is initialized to 0.

- Typically all space will be assigned a `label` so that it can be accessed.
We will be using a library to simplify input and output from assembly programs.

The library is named `libcs220.a`. To link `foo.s` with the library:

```
g++ -o foo foo.s -LLIBDIR -lcs220
```

LIBDIR should be replaced with the path to the directory containing the library.
Program `foo.s` should have this form:

```assembly
.globl _asm_main
.section .text
_asm_main:
  enter $0, $0   # set up stack
  pusha         # save registers
  # User-written code goes here.
  popa
  movl $0, %eax # return status
  leave         # restore stack
  ret
```
The CS220 library output routines are: `print_int`, `print_uint`, `print_char`, `print_str`, `print_nl`. The EAX register contains the data (or the address of the string) to be displayed:

```assembly
movl $12345, %eax
call print_int   # display 12345
call print_nl
movl count, %eax   # EAX <- Value
call print_int
movl $msg1, %eax   # EAX <- Address
call print_str   # display string
```
The input routines are: `read_int`, `read_uint`, `read_char`, `read_str`, `read_line`, `read_nl`. The EAX register contains the data read (or the address of the string).

```
movl $msg2, %eax
call print_str # prompt
call read_int  # get value
movl %eax, ival  # store in mem

movl $strbuf, %eax # buffer addr
call read_line # get input
```
NOTE: We could also use the C `stdio` routines (printf and scanf) for I/O. The `stdio` routines are more difficult to use (especially the input routines) than the CS220 library routines. Using the library will allow us to write more interesting programs more quickly.
Our first program will use the CPUID instruction to display information about the processor. CPUID returns different results depending on the value in the EAX register. (See the GAS IA-32 Assembly Language Programming reference.)

With EAX equal to 0, CPUID returns the Vendor ID string in registers EBX, EDX, and ECX. (The ID is a 12 character string.)
.globl _asm_main

.section .data
output:
    .asciz "The processor Vendor ID is 'xxxxxxxxxxxxxxxxx\n"

.section .text
_asm_main:
    enter $0, $0
    pusha

    # Continued on next slide
# Continued from previous slide

movl $0, %eax

cpuid

movl $output, %edi

movl %ebx, 28(%edi)

movl %edx, 32(%edi)

movl %ecx, 36(%edi)

movl $output, %eax

call print_str

popa

movl $0, %eax

leave

ret
The "movl $output, %edi" moves (copies) the address of the first byte of the output string into the EDI register. (In contrast, “movl output, %edi” would move the first four bytes of the output string into EDI.)

“movl %ebx, 28(%edi)” moves the data in EBX to the address in EDI plus 28.
A debugger allows us to step through a program instruction by instruction. We can examine the registers and/or memory locations after each instruction.

To use the debugger we need to assemble our program using the `-g` or `-gstabs` option. This will add extra information to the object file to aid the debugger:

```
g++ -g -o cpuid cpuid.s
   -L../cs220lib -lcs220
```
To run the program under the debugger:

```
gdb cpuid
```

This will start the debugger and load the program, but the program is not run. You run the program by typing `run` at the `gdb` prompt. You will usually want to set one or more breakpoints first. Set a breakpoint at `_asm_main` by entering:

```
(gdb) break asm_main
```
AS03-Developer Tools
Using the Debugger

- Now type **run** at the gdb prompt to run the program under the debugger until the breakpoint is reached. **gdb** will then display the instruction to be executed **next**:
  9  enter  $0, $0

- You then use either the **step** or **next** commands to run the program line-by-line. **step** steps into a function call, while **next** executes the function and then stops at the next line after the function call.
AS03-Developer Tools
Using the Debugger

• You can also use the `cont` (continue) command to continue running the program until the end (or the next breakpoint).

• The great power in using a debugger comes in being able to examine registers and/or data values after executing an instruction.

• The `print` command is used to display register or variable values. The `x` command is used to display memory contents. You can display all registers by typing `info registers`. 
Here are a few examples:

```plaintext
print /d $eax    # display in dec
desc
print /x $ebp    # display in hex
desc
info registers
x /dw &yval     # integer
desc
x /cb &xval     # byte
desc
x /s &output    # string
desc
```
The list command is used to display lines of source code:

```
(gdb) list
1  .globl _asm_main
2
3  .section .data
4  output: .asciz "The processor Vendor ID is 'xxxxxxxxxxxxxxx'\n"
5
6         .section .text
```
**AS03-Developer Tools**

**Using the Debugger**

- **list** will display 10 lines of code. Successive **list** commands will display the next 10 lines. **list 1** will display the first 10 lines again.

- List is useful for displaying source code line numbers, breakpoints can also be specified by **line number**:

  
  (gdb) **break 8**

- **quit** will quit the debugger.
• **help** will display a list of commands. Type **help** followed by a command name to get information on the command.

• Refer to the **gdb info** pages for complete documentation on the debugger.

• Today's in-class exercise will lead you through the use of the debugger.
• We will cover C++/assembly interfacing in more depth later. Here is a preview.

• To call a C/C++ function from assembly you must understand the standard C calling convention:
  - Function arguments are passed on the stack. The rightmost arg is pushed onto the stack first.
  - The function return is stored in EAX.
  - The calling routine is responsible for popping the arguments off of the stack.
The following C++ code:

```cpp
// seed random number generator
// with current time
srand(time(0));
// Retrieve a random number
int x = rand();
cout << x << endl;
```

could be written in assembly as ...
pushl $0        # Push time arg
call  _time
addl  $4, %esp  # “Pop” time arg
pushl %eax      # Push srand arg
call  _srand
addl  $4, %esp  # “Pop” srand arg
call  _rand     # Call rand
movl  %eax, x   # Store number
call  print_int # Display number
call  print_nl