Homework

• Reading
  – Professional Assembly Language, pp 73-106
  – Also, study website references:
    • “Using gas” and “gas i386 Assembly Language Guide”

• Machine Projects
  – Turn in mp1 tonight
  – Get mp2warmup/Do it for practice (no turn-in)
  – Start looking at mp2

• Labs
  – Continue with labs in your assigned section
C versus Assembly Language

• C is called a “portable assembly language”
  – Allows low level operations on bits and bytes
  – Allows access to memory via use of pointers
  – Integrates well with assembly language functions

• Advantages over assembly code
  – Easier to read and understand source code
  – Requires fewer lines of code for same function
  – Doesn’t require knowledge of the hardware
C versus Assembly Language

• Good reasons for learning assembly language
  – It is a good way to learn how a processor works
  – In time-critical sections of code, it is possible to improve performance with assembly language
  – In writing a new operating system or in porting an existing system to a new machine, there are sections of code which must be written in assembly language such as the cpuid example in this lecture
Best of Both Worlds

• Integrating C and assembly code
• Convenient to let C do most of the work and integrate with assembly code where needed
• Make our gas routines callable from C
  – Use C compiler conventions for function calls
  – Preserve registers that C compiler expects saved
Instruction Four Field Format

• Label:
  – Can be referred to as a representation of the address
  – Usual practice is to place these on a line by themselves
• Mnemonic to specify the instruction and size
  – Makes it unnecessary to remember instruction code values
• Operand(s) on which the instruction operates (if any)
  – Zero, one, or two operands depending on the instruction
• Comment contains documentation
  – It begins with a # anywhere and goes to the end of the line
  – It is very important to comment assembly code well!!
Assembly Framework for a Function

- General form for a function in assembly is:
  
  ```
  .globl _mycode
  .text
  _mycode:
    . . .
  ret
  .data
  mydata:
    .long 17
  .end
  ```
Assembler Directives

• Defining a label for external reference (call)
  .globl _mycode

• Defining the code section of program (ROM)
  .text

• Defining the static data section of program (RAM)
  .data

• End of the Assembly Language
  .end
Assembler Directives for Sections

- These directives designate sections where we want our assembler output placed into memory
  - `.text` places the assembler output into program memory space (e.g. where PROM will be located)
  - `.data` places the assembler output into a static initialized memory space (e.g. where RAM will be located)
  - `.bss` allows assembler to set labels for uninitialized memory space (we won’t be using this section)
  - `.section` ignore/omit this directive with our assembler

- In builds, ld is given addresses for the sections
Assembler Directives

• Defining / initializing static storage locations:

  label1:
    .long 0x12345678  # 32 bits

  label2:
    .word 0x1234       # 16 bits

  label3:
    .byte 0x12         # 8 bits
Assembler Directives

- Defining / initializing a string

```assembly
label1:
    .ascii "Hello World\n\0"

label2:
    .asciz "Hello World\n"
```
Defining Constant Values

- Constant definitions follow C conventions:
  - `$123`  # decimal constant
  - `$0x123`  # hex constant
  - `$'a'`  # character constant
  - `$'\n'`  # character escape sequence

- With the following exception:
  - `$'\0'`  # character octal constant
    # just use `$0` instead
Symbolic Constant Names

• Allow use of symbols for numeric values
  - Perform same function as C preprocessor \#define
  - Unfortunately not the same format as used in C preprocessor so can’t just include .h files to define symbols across combination of C/assembly code
  - Format is: SYMBOL = value
  - Example: NCASES = 8
    movl $NCASES, %eax
Addressing Memory

• Direct addressing for memory
  – Gas allows use of hard coded memory addresses
  – Not recommended except for HW based addresses
  – Examples: .text
    movb %al, 0x1234
    movb 0x1234, %dl
    ...
Addressing Memory

• Direct addressing for memory
  – Gas allows use of a label for memory address
  – Examples:

    ```
    .text
    movb %al, total
    movb total, %dl
    ...
    .data
    total: .byte 0
    ```
Addressing Memory

- Indirect - like *pointer in C
  - Defined as using a register as the address of the memory location to access in an instruction

\[
\begin{align*}
  \text{movl } \$0x1234, \%ebp \\
  \text{movb } (\%ebp), \%al
\end{align*}
\]
Addressing Memory

• Indirect with Offset - like *(pointer+4) in C
  – May also be done with a fixed offset, e.g. 4

\[
\text{movl } \$0x1234, \%ebp \\
\text{movb } 4(\%ebp), \%al
\]

%ebp  0x00001234

%al

Memory

One byte
Through the Looking Glass,
by Lewis Carroll

"The name of the song is called 'Haddocks' Eyes.'"
"Oh, that's the name of the song, is it?" Alice said trying to feel interested.

"No, you don't understand," the Knight said, looking a little vexed. 'That's what the name is called. The name really is 'The Aged Aged Man.'"
"Then I ought to have said 'That's what the song is called'?" Alice corrected herself.

"No, you oughtn't: that's quite another thing! The song is called 'Ways and Means': but that's only what it's called, you know!"
"Well, what is the song, then?" said Alice, who was by this time completely bewildered.

"I was coming to that," the Knight said. 'The song really is ‘A-sitting On A Gate': and the tune's my own invention."
Through the Looking Glass
Lewis Carroll in C code

- Defining an array and initializing a pointer to it:
  ```c
  char WaysandMeans[100]; // At address TheAgedAgedMan
  strcpy (WaysandMeans, "A Sitting on a Gate");
  char *HaddocksEyes = WaysandMeans; // TheAgedAgedMan
  
  WaysandMeans(@TheAgedAgedMan) "A Sitting on a Gate"
  HaddocksEyes TheAgedAgedMan which is the &WaysandMeans
  ```

- Dereferencing the pointer:
  ```c
  printf("%s\n", HaddocksEyes);
  ```

- Prints what?
Addressing Memory

• Memory-memory addressing restrictions
  – Why can’t we write instructions such as these?
    movl first, second  # direct
    movl (%eax), (%ebx) # indirect

  – Intel instruction set does not support instructions to move a value from memory to memory!

• Must always use a register as an intermediate location for the value being moved, e.g.
  movl first, %eax       # direct from mem
  movl %eax, second      # direct to mem
Integrating C and Assembly

- Pick up the makefile from $pcbook
- Always read the makefile for a program first!
- The makefile in $pcbook expects a “matched pair”
  - C driver filename is mycodec.c
  - Assembly filename is mycode.s
- The make file uses macro substitutions for input:
  - The format of the make command is:
    make A=mycode
Example: Function cpuid

- C “driver” in file cpuidc.c to execute code in cpuid.s

```c
/* cpuidc.c - C driver to test cpuid function
 * bob wilson - 1/15/2012
 */
#include <stdio.h>

extern char *cpuid(); /* our .s file is external*/

int main(int argc, char **argv)
{
    printf("The cpu ID is: %s\n", cpuid());
    return 0;
}
```
Example: Function cpuid

- Assembly code for function in file cpuid.s

```assembly
; cpuid.s C callable function to get cpu ID value
.data
buffer:
  .asciz "Overwritten!"  # overwritten later
.text
.globl _cpuid
_cpuid:
  movl $0,%eax          # zero to get Vendor ID
  cpuid                 # get it
  movl $buffer, %eax    # point to string buffer
  movl %ebx, (%eax)     # move four chars
  movl %edx, 4(%eax)    # move four chars
  movl %ecx, 8(%eax)    # move four chars
  ret                   # string pointer is in %eax
.end
```

Self Modifying Code

• Our assembler does not actually support cpuid instruction, so I made the code self-modifying:

```assembly
... 
_cpuid:
    movb $0x0f, cpuid1  # patch in the cpuid first byte
    movb $0xa2, cpuid2  # patch in the cpuid second byte
    movl $0,%eax       # input to cpuid for ID value
    cpuid1:            # hex for cpuid instruction here
        nop            # 0x0f replaces 0x90
    cpuid2:            # 0xa2 replaces 0x90
        nop
... 
```
Self Modifying Code

• Obviously, the self modifying code I used for this demonstration would not work if:
  – The code is physically located in PROM/ROM
  – There is an O/S like UNIX/Linux that protects the code space from being modified (A problem that we avoid using Tutor on our SAPC’s)

• Try justifying this “kludge” to the maintenance programmer!!
Here is self-modifying code in C:

```c
int main(int argc, char **args)
{
    // array to hold the machine code bytes of the function
    static char function[100];

    function[0] = 0xb8; // move address of the string to %eax
    ....
    function[5] = 0xc3; // and return
    ....
    // execute the function whose address is the array
    printf("%s\n", (* (char * (*)()) function)());

    return 0;
}
```