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
  - Use C compiler convention for passing return value

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:
          . . .         # comments
          ret
          .data
mydata:
          .long   17    # comment
          .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

```
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'  # assembles as '0' instead of 0
       # just use $0 to avoid problem
```

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

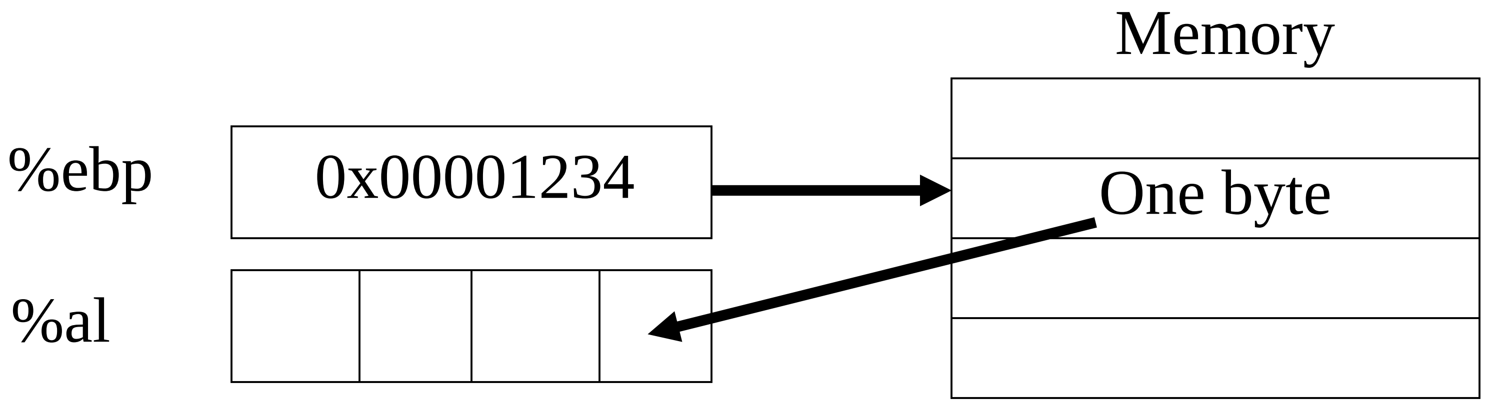
- Direct addressing for memory
  - Gas allows use of a label for memory address
  - Examples:

```
        .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

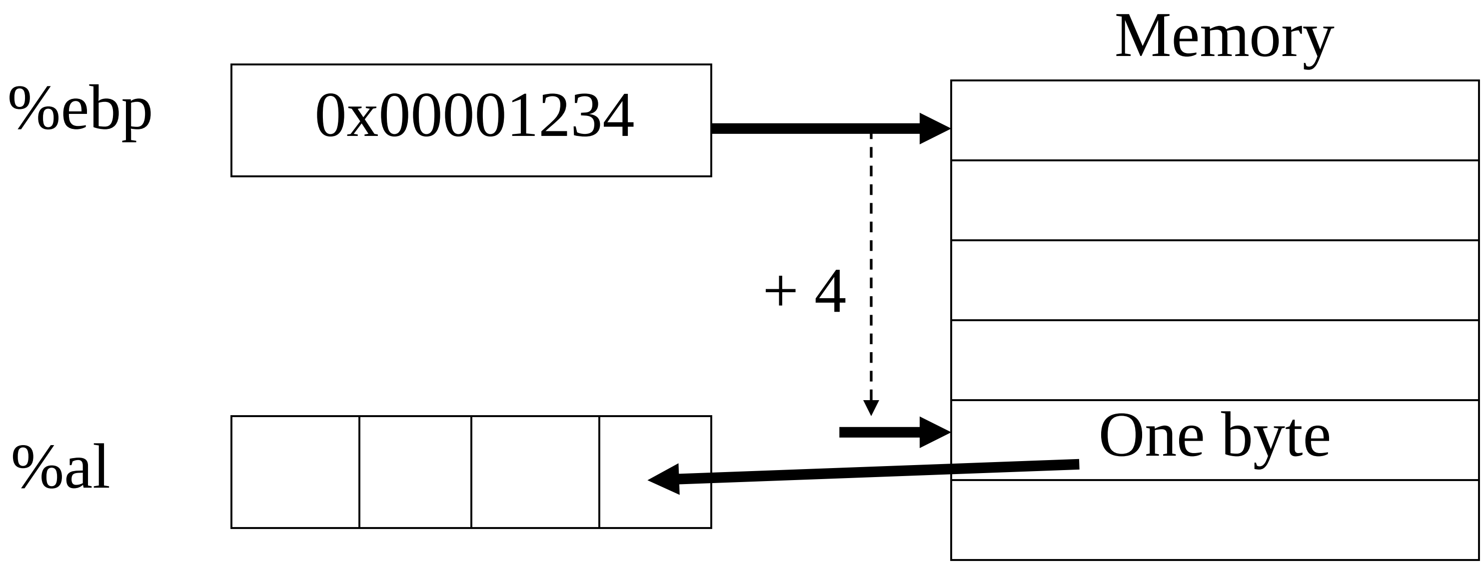
```
movl $0x1234, %ebp
movb (%ebp), %al
```



Addressing Memory

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

```
movl $0x1234, %ebp
movb 4(%ebp), %al
```



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:

```
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 (@SomeAddress)     TheAgedAgedMan which
                                    is the &WaysandMeans
```

- Dereferencing the pointer:

```
printf(“%s\n”, HaddocksEyes);
```

- Prints what?

Through the Looking Glass
Lewis Carroll in i386 Assembly

- Defining an array and initializing a pointer to it:

```
        .data
WaysandMeans: .asciz "A Sitting on A Gate"
HaddocksEyes: .long 0      # pointer to WayandMeans
        .text
        movl $WaysandMeans, HaddocksEyes
```

```
WaysandMeans (@TheAgedAgedMan) "A Sitting on a Gate"

HaddocksEyes (@SomeAddress)    TheAgedAgedMan which
                               is the &WaysandMeans
```

- Dereferencing the pointer:

```
        pushl HaddocksEyes
        call _printf      # 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
```

Introduction to MP2

- In mp2, you will write four assembly language functions in their respective source files (*.s)
- I have provided C driver code for testing (*c.c) and a makefile for doing a combined build to produce an executable file (*.lnx) for Tutor