Today

- Data section (cont)
- Addressing (intro)

Assignments

- HW5 out
- Quiz on Thurs
The `imul` instruction ... similar to `mul`, but for signed multiplication

```
imuls amt                   ... similar to mul

imuls src, dst              ... two operand version dst = src * dst
  - similar to add and sub
  - dst must be a register

imuls imm, src, dst         ... three operand version dst = imm * src
  - imm must be a literal (immediate) value
  - src must be a memory location or a register
  - dst must be a register

• example:

  movl   $8096, %eax         # eax = 8096
  imull  $64, %eax           # eax = 64 * eax
  xorl   %eax, %eax          # eax = 0
  movl   $8096, %ebx         # ebx = 8096
  imull  $64, %ebx, %eax     # eax = 64 * ebx
```
The **div instruction** ... unsigned integer division

Basic syntax:

\[
\text{div } s \text{ divisor} \\
\text{– } s \text{ is the divisor size} \\
\text{– } dx:ax \text{ is dividend for 32-bit divisor (see below)} \\
\text{– } ax \text{ is the resulting quotient} \\
\text{– } dx \text{ is the resulting remainder} \\
\text{– thus, result can be twice the size of original}
\]

- example:

\[
\begin{align*}
\text{movw } & \quad 0, %dx \\
\text{movw } & \quad 32, %ax \quad \# dx:ax = 32 \\
\text{movw } & \quad 3, %cx \quad \# cx = 3 \\
\text{divw } & \quad %cx \quad \# ax = 32/3, dx = 32 \, \%\, 3
\end{align*}
\]

Unsigned integer operands for unsigned division:

<table>
<thead>
<tr>
<th>Divisor</th>
<th>Dividend</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = b</td>
<td>ax</td>
<td>al</td>
<td>ah</td>
</tr>
<tr>
<td>s = w</td>
<td>dx:ax</td>
<td>ax</td>
<td>ax</td>
</tr>
<tr>
<td>s = l</td>
<td>edx:eax</td>
<td>eax</td>
<td>edx</td>
</tr>
<tr>
<td>s = r</td>
<td>rdx:rax</td>
<td>rax</td>
<td>rdx</td>
</tr>
</tbody>
</table>

Use **idiv** for signed division

- however, unlike **imul**, only a 1-operand version of **idiv**
- no 2 operand and no 3 operand versions
Data Section

Use the `.data` section for storing data in memory

- similar to defining “variables” in your program
- allows reading from and writing to memory
- a different set of mnemonics for data sizes (types) ...

Example:

```
.global _start
.text

_start:
    # write mystr
    movq $1, %rax # system call 1 is write
    movq $1, %rdi # file handle 1 is stdout
    movq $mystr, %rsi # address of string to output
    movq $14, %rdx # number of bytes
    syscall

    # exit(0)
    movq $60, %rax # system call 60 is exit
    xorq %rdi, %rdi # we want return call 0
    syscall

.data

mystr: .ascii "Hello, World!\n" # or .string to add \n
```
Data type specifiers in a .data section:

- `.byte` 1 byte integer
- `.ascii, .string` 1 byte
- `.word` 2 byte integer
- `.long` 4 byte integer
- `.quad` 8 byte integer

Example storing a string length:

```assembly
data

mystr: .ascii "Hello, World!\n" # or .string to add \0
mylen: .long 14
```

A string is just an “array” of bytes (ascii)

- the label (e.g., `mystr`) is the address of the first byte
- the bytes are stored sequentially from the first address

Calculating string lengths with the .equ (directive)

```assembly
data

mystr: .ascii "Hello, World!\n" # or .string to add \0
.equ mylen, (. - mystr)
```

- `.equ` sets `mylen` to the expression `. - mystr`
- the . is the “current location counter”
- so `. - mystr` is the current address minus the `mystr` address
Q: How do we make a letter uppercase? What about lowercase?
Manipulating the first character of a string

- note that we can easily manipulate the first string character ...

```
.global _start
.text

_start:
    movb mystr, %al
    addb $32, %al  # make "H" lowercase
    movb %al, mystr

    # write mystr
    movq $1, %rax   # system call 1 is write
    movq $1, %rdi   # file handle 1 is stdout
    movq $mystr, %rsi  # address of string to output
    movq $mylen, %rdx  # number of bytes
    syscall

    # exit(0)
    movq $60, %rax   # system call 60 is exit
    xorq %rdi, %rdi  # we want return call 0
    syscall

.data
mystr:  .ascii "Hello, World!\n"
.equ mylen, (. - mystr)
```
Accessing subsequent characters ...  

- we use something like a “pointer”

```assembly
.global _start
.text

_start:
    movb mystr, %al  # make "H" lowercase
    addb $32, %al
    movb %al, mystr

    movb $mystr, %rax  # treats mystr as address literal
    addq $1, %rax  # add one to address
    movb (%rax), %bl  # uses %rax as pointer (follow address)
    subb $32, %bl  # make "e" uppercase
    movb %bl, (%rax)  # store result (again as pointer)

    # write mystr
    movq $1, %rax  # system call 1 is write
    movq $1, %rdi  # file handle 1 is stdout
    movq $mystr, %rsi  # address of string to output
    movq $mylen, %rdx  # number of bytes
    syscall

    # exit(0)
    movq $60, %rax  # system call 60 is exit
    xorq %rdi, %rdi  # we want return call 0
    syscall

.data
mystr: .ascii "Hello, World!\n"
.equ mylen, (. - mystr)
```