## Today
- Quiz 4
- More basic assembly instructions
- Data section

## Assignments
- HW4 due
- HW5 out (soon)
Exercise: Write a GAS program to simulate the following C code snippet

- Assume \( x \) stored in \( r8d \), \( y \) in \( r9d \), \( u \) in \( r10d \), \( v \) in \( r11d \), and \( w \) in \( eax \)

```c
int x = 132;
int y = 219;
int u = y - x;
int v = -(x - y);
int w = (++u) + (++v);  // w == 176
```

One solution:

```assembly
.global _start
.text
_start:
    movl $132, %r8d  # x = 132
    movl $219, %r9d  # y = 219
    movl %r9d, %r10d  # u = y
    subl %r8d, %r10d  # u = u - x
    movl %r8d, %r11d  # v = x
    subl %r9d, %r11d  # v = v - y
    negl %r11d  # v = -v
    movl %r10d, %eax  # w = u
    incl %eax  # w = ++w
    movl %r11d, %edx  # edx = v
    incl %edx  # edx = ++edx
    addl %edx, %eax  # w = w + edx
    # exit
    movq $60, %rax
    xor %rdi, %rdi
    syscall
```
Instructions we've looked at so far:

- mov, add, sub, neg, inc, dec

Today we'll look at:

- xchg, shl, shr, mul, and div
- the .data section
- basic branching
The `xchg` instruction

Basic syntax:

```
xchgs operand1, operand2
```

- `s` is the size of data being moved
- `operand` is either a memory loc or register
- both operands can’t be memory locations

• example:

```
# swap register contents with three moves
movl %eax, %ebx  # ebx = eax
movl %edx, %eax  # eax = edx
movl %ebx, %edx  # edx = ebx

# swap register contents with one instruction
xchgl %eax, %edx
```
The shl / shr instructions

Basic syntax:

\[
\text{shls } \text{amt, dst} \quad \ldots \text{ dst } = \text{ dst } \ll \text{ amt} \\
\text{shrs } \text{amt, dst} \quad \ldots \text{ dst } = \text{ dst } \gg \text{ amt}
\]

- \( s \) is the size of data being shifted
- \( \text{amt} \) is the number of bits to shift
- \( \text{dst} \) is either a memory loc or register

• example:

\[
\begin{align*}
\text{movb } & \quad $42, \%al \quad \# \text{ al } = 42 \quad (0010 \ 1010) \\
\text{shlb } & \quad $2, \%al \quad \# \text{ al } = 168 \quad (1010 \ 1000) \\
\text{shrb } & \quad $3, \%al \quad \# \text{ al } = 21 \quad (00010101)
\end{align*}
\]

Q: What happens with this shift (move outside of a byte)?

\[
\begin{align*}
\text{movb } & \quad $16, \%al \quad \# \text{ al } = 16 \quad (0001 \ 0000) \\
\text{shlb } & \quad $4, \%al \quad \# \text{ al } = 0 \ldots \text{ instead of: } (0001 \ 0000 \ 0000)
\end{align*}
\]
The `mul` instruction ... unsigned integer multiplication

Basic syntax:

```
muls amt                     ... dx:ax = ax * amt (for s = w)
```

- `s` is the multiplicand size
- `ax` is the multiplicand (or `eax` for 32-bit, etc)
- `amt` is the multiplier
- result stored in two registers `dx:ax` where `dx` high and `ax` low order bits
- thus, result can be twice the size of original

- example:

```
movw $8096, %ax       # 0x 1F A0
movw $64, %bx        # 0x 40
mulw %bx             # 0x 7 E8 00, dx = 0x7, ax = 0xE800
                     # 51844, dx = 7, ax = 59392
                     # note: CF flag is set!
```

Unsigned integer operands for unsigned multiplication:

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Multiplicand</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = b</code></td>
<td><code>al</code></td>
<td><code>ax</code></td>
</tr>
<tr>
<td><code>s = w</code></td>
<td><code>ax</code></td>
<td><code>dx:ax</code></td>
</tr>
<tr>
<td><code>s = l</code></td>
<td><code>eax</code></td>
<td><code>edx:eax</code></td>
</tr>
<tr>
<td><code>s = r</code></td>
<td><code>rax</code></td>
<td><code>rdx:rax</code></td>
</tr>
</tbody>
</table>
The `imul` instruction ... similar, 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 \texttt{div} instruction ... unsigned integer division

Basic syntax:

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

- example:

\begin{verbatim}
movw $0, %dx
movw $32, %ax # dx:ax = 32
movw $3, %cx # cx = 3
divw %cx # ax = 32/3, dx = 32 \% 3
\end{verbatim}

Unsigned integer operands for unsigned division:

\begin{table}[h]
\centering
\begin{tabular}{llll}
\hline
Divisor & Dividend & Quotient & Remainder \\
\hline
\(s = b\) & ax & al & ah \\
\(s = w\) & dx:ax & ax & ax \\
\(s = 1\) & edx:eax & eax & edx \\
\(s = r\) & rdx:rax & rax & rdx \\
\hline
\end{tabular}
\end{table}

Use \texttt{idiv} for signed division

- however, unlike \texttt{imul}, only a 1-operand version of \texttt{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:

```assembly
.global _start
.text
_start:
    # write(1, mystr, strlen)
    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 \0
```
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:

```
.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 .equ (directive)

```
.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
Manipulating the first character of a string

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

```assembly
.global _start
.text

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

    # write(1, mystr, strlen)
    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 need user “pointers” (manipulate addresses)

```
.global _start
.text

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

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

# write(1, mystr, strlen)
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)
```