CSC231 - Assembly

Week #4

Dominique Thiébaut
dthiebaut@smith.edu
• Review Hexdump

• Pentium Data Registers

• 32-bit, 16-bit and 8-bit quantities (registers & memory locations)
Review
• These are the numbers that get loaded into RAM when a program runs.
- The code segment always starts first, at offset 0000080
• b8 04 00 00 00 are "op codes" or "operation codes"

• These particular bytes represent "mov eax, 4"
• `b8 04 00 00 00` are "op codes" or "operation codes"

• These particular bytes represent "mov eax, 4"
• The data segment follows the code segment
• The data segment follows the code segment
• Strings can easily be recognized
The Pentium Data Registers
Pentium Registers

eax
ebx
ecx
edx
Pentium Registers

32 bits = 4 bytes

eax
ebx
cdx
edx
mov eax, 4

section .data
Hello db "Hello there!", 10, 10
HelloLen equ $-Hello

section .text
_global _start

_start:

;;; print message
mov eax, 4 ; write
mov ebx, 1 ; stdout
mov ecx, Hello ; stdout
mov edx, HelloLen ;
int 0x80

;;; exit
mov ebx, 0
mov eax, 1
int 0x80
Pentium Registers

**mov eax, 4**
Pentium Registers

mov ebx, eax

eax

ebx

cx

dx
Pentium Registers

x dd 5

mov ecx, dword[x]

eax
ebx
ecx
edx

00
00
00
05
x dd 5
add ecx, 1
mov dword[x]

Pentium Registers

eax
ebx
cx
edx

x
00
00
00
05
Code this in assembly:

```c
int x = 3, y;
y = 10;
z = x*2 + y;
```
The Pentium data registers can also be used to operate on 16-bit quantities (words), as well as 8-bit quantities (bytes)
Pentium Registers

eax  |  ax
ebx  |  bx
ecx  |  cx
derx  |  dx
Why 16-bit quantities?

- **byte**: The `byte` data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The `byte` data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of `int` where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.

- **short**: The `short` data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with `byte`, the same guidelines apply: you can use a `short` to save memory in large arrays, in situations where the memory savings actually matters.

- **int**: By default, the `int` data type is a 32-bit signed two's complement integer, which has a minimum value of $-2^{31}$ and a maximum value of $2^{31}-1$. In Java SE 8 and later, you can use the `int` data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of $2^{32}-1$. Use the `Integer` class to use `int` data type as an unsigned integer. See the section The Number Classes for more information. Static methods like `compareUnsigned`, `divideUnsigned` etc have been added to the `Integer` class to support the arithmetic operations for unsigned integers.

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
a       dw     9
mov  ax, 1
mov  bx, word[a]
mov  cx, bx
mov  dx, ax

Pentium Registers

eax       | ax
ebx       | bx
cx        | cx
dx        | dx
<table>
<thead>
<tr>
<th>eax</th>
<th>ah</th>
<th>al</th>
<th>ax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>bh</td>
<td>bl</td>
<td>bx</td>
</tr>
<tr>
<td>ecx</td>
<td>ch</td>
<td>cl</td>
<td>cx</td>
</tr>
<tr>
<td>edx</td>
<td>dh</td>
<td>dl</td>
<td>dx</td>
</tr>
</tbody>
</table>
Think of **ah** and **al** as boxes inside a bigger one called **ax**, and **ax** as half of a bigger box still, called **eax**.
Declaring Variables
• **db**: define **byte** storage

• **dw**: define **word** storage

• **dd**: define **double-word** storage
## Examples: db

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
<td>db</td>
<td>&quot;Hello&quot;, 10</td>
</tr>
<tr>
<td>a</td>
<td>db</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>db</td>
<td>'H'</td>
</tr>
<tr>
<td>c</td>
<td>db</td>
<td>255</td>
</tr>
<tr>
<td>d</td>
<td>db</td>
<td>0x80</td>
</tr>
</tbody>
</table>
### Examples: `dw`

<table>
<thead>
<tr>
<th>x</th>
<th>dw</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>dw</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>dw</td>
<td>255</td>
</tr>
<tr>
<td>t</td>
<td>dw</td>
<td>0x1234</td>
</tr>
</tbody>
</table>

Examples: `dd`

<table>
<thead>
<tr>
<th>Alpha</th>
<th><code>dd</code></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beta</td>
<td><code>dd</code></td>
<td>255</td>
</tr>
<tr>
<td>Gamma</td>
<td><code>dd</code></td>
<td>0x12345678</td>
</tr>
</tbody>
</table>
Summary of important concepts just seen

- Numbers
- Op Codes
- Machine Language
- Hexadecimal
- Executable Files
More Examples of The *mov* Instruction

`mov dest, source`
section .data
lf db 10
ch db 0
a dw 0x1234
b dw 0
x dd 0
y dd 0x12345678

section .text
; put lf in al

<table>
<thead>
<tr>
<th>eax</th>
<th>ah</th>
<th>al</th>
<th>ax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>bh</td>
<td>bl</td>
<td>bx</td>
</tr>
<tr>
<td>ecx</td>
<td>ch</td>
<td>cl</td>
<td>cx</td>
</tr>
<tr>
<td>edx</td>
<td>dh</td>
<td>dl</td>
<td>dx</td>
</tr>
</tbody>
</table>
section .data
lf db 10
ch db 0
a dw 0x1234
b dw 0
x dd 0
y dd 0x12345678

section .text
; put al in ch

Exercises
section .data
lf  db  10
ch db  0
a  dw  0x1234
b  dw  0
x  dd  0
y  dd  0x12345678

section .text

; put a in bx

; put bx in b

; put bx in ax

; put 0 in cx
section .data
lf       db      10
ch       db      0
a        dw      0x1234
b        dw      0
x        dd      0
y        dd      0x12345678

section .text
; put x in eax

; put y in ecx

; put ecx in edx

; put ex into y

Exercises
section .data
lf       db      10
ch       db      0
a        dw      0x1234
b        dw      0
x        dd      0
y        dd      0x12345678

section .text
; put 0 in ah

; put 3 in cx

; put 5 in edx

; put 0x12345678 into eax
We understand mov!