Addressing example explained

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Operand</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%eax</td>
<td>0x100</td>
<td>Value is in the register</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
<td>0x104</td>
<td>0xAB</td>
<td>Value is at the address</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
<td>$0x108</td>
<td>0x108</td>
<td>Value is the value ($ says “I’m an immediate, i.e. constant, value”)</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
<td>(%eax)</td>
<td>0xFF</td>
<td>Value is at the address stored in the register → GTV@(reg)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4(%eax)</td>
<td>0XAB</td>
<td>GTV@(4+ reg)</td>
</tr>
<tr>
<td>Register</td>
<td>Value</td>
<td>9(%eax,%edx)</td>
<td>0X11</td>
<td>GTV@(9 + reg + reg)</td>
</tr>
<tr>
<td>%eax</td>
<td>0x100</td>
<td>260(%ecx,%edx)</td>
<td>0X13</td>
<td>Same as above; be careful, in decimal</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1</td>
<td>0xFC(%ecx,4)</td>
<td>0XFF</td>
<td>GTV@(0xFC + 0 + reg*4)</td>
</tr>
<tr>
<td>%edx</td>
<td>0x3</td>
<td>(%eax,%edx,4)</td>
<td>0X11</td>
<td>GTV@(reg + reg*4)</td>
</tr>
</tbody>
</table>

In red are memory types of operands which is why you get the value at the address; because you are accessing memory (see the book section 3.4.1 and figure 3.3 for more)
FYI: last two, the 3rd value in () is the scaling factor which must be 1, 2, 4 or 8
NOTE: Do not put ‘$’ in front of constants when they are addressing indexes, only when they are literals.
An *addressing mode* is a mechanism for specifying an address.

- **Immediate**
- **Register**
- **Memory**
  - **Absolute**
    - specify the address of the data
  - **Indirect**
    - use register to calculate address
  - **Base + displacement**
    - use register plus absolute address to calculate address
  - **Indexed**
    - Indexed
      - Add contents of an index register
    - Scaled index
      - Add contents of an index register scaled by a constant
Data movement instructions

- Move, push and pop
- MOVE example
- Operands
  - source, dest
- Fill-in
  - S = sign extend
  - Z = zero extend
- b, w, l = byte, word, long
  - 8, 16, 32 bits respectively
- Instructions (a sample set)
  - movb, movw, movl = S → D
  - movsbw, movsbl, movswl = SignExtend(S) → D
  - movzbw, movzbl, movzwl = ZeroExtend(S) → D

Given %dh = 0xCD and %eax = 0x98765432
What is in %eax after each instruction?
1. movb %dh, %al
2. movsbl %dh, %eax
3. movzbl %dh, %eax
Push and Pop
Stack = LIFO
pushl S
 - R[esp] – 4 → R[esp]... decrement stack ptr
 - S → M[R[esp]]... store to memory
 - Order matters!

popl D
 - M[R[ESP]] → D... reading from memory
 - R[esp] + 4 → R[esp]... increment stack ptr
 - Order matters!

By convention, we draw stacks upside down
 - “top” of the stack is shown at the bottom

Stack “grows” toward lower addresses (push)
 - Top element of the stack has the lowest address of all stack elements
The stack

Initially

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x123</td>
</tr>
<tr>
<td>%edx</td>
<td>0</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

pushl %eax

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x123</td>
</tr>
<tr>
<td>%edx</td>
<td>0</td>
</tr>
<tr>
<td>%esp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

popl %edx

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x123</td>
</tr>
<tr>
<td>%edx</td>
<td>0x123</td>
</tr>
<tr>
<td>%esp</td>
<td>0x108</td>
</tr>
</tbody>
</table>
Determine the appropriate instruction suffix based on the operands

- `mov %eax, (%esp)`
- `mov (%eax), %dx`
- `mov $0xFF, %b1`
- `mov (%esp, %edx,4), %dh`
- `push $0xFF`
- `mov %dx, (%eax)`
- `pop %edi`
Practice problem 3.3

Each of the following lines of code generates an error message. Explain what is wrong with each line.

- `movb $0xF, (%bl)` – cannot use `%bl` as addr reg
- `movl %ax, (%esp)` – mismatch long and %ax
- `movw (%eax), 4(%esp)` – can’t mem to mem
- `movb %ah, %sh` – no `%sh` reg
- `movl %eax, $0x123` – can’t have immed as dest
- `movl %eax, %dx` – dest op incorrect size
- `movb %si, 8(%ebp)` –
Swap example

REMINDER: we use pointers so can pass address since can’t pass values back outside of the function

```c
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
} //codeswap.c
```

**Register | Value**
--- | ---
%edx | xp
%ecx | yp
%ebx | t0
%eax | t1

*swap:*
- `pushl %ebp`
- `movl %esp, %ebp`
- `pushl %ebx`
- `movl 8(%ebp), %edx`
- `movl 12(%ebp), %eax`
- `movl (%edx), %ecx`
- `movl (%eax), %ebx`
- `movl %ebx, (%edx)`
- `movl %ecx, (%eax)`
- `popl %ebx`
- `popl %ebp`
- `ret`

**Setup/prologue**

**Body**
- `edx=xp`
- `ecx=yp`
- `ebx=*xp (t0)`
- `eax=*yp (t1)`
- `*xp = t1`
- `*yp=t0`

**Finish/epilogue**

### Understanding Swap

1. Move 0x124 to %edx
2. Move 0x120 to %ecx
3. Move 123 to %ebx
4. Move 456 to %eax
5. Move 456 to M[0x124]
6. Move 123 to M[0x120]

```
pushl %ebp
movl %esp, %ebp
pushl %ebx

popl %ebx
popl %ebp
ret
```