x86 general-purpose registers

(most significant)                  (least)
[........ ........ ........ ........] eax 32 bits
[........ ........] ax 16 bits
[........] ah 8 bits
[........] al 8 bits

[........ ........ ........ ........] ebx
[........ ........] bx
[........] bh
[........] bl

[........ ........ ........ ........] ecx
[........ ........] cx
[........] ch
[........] cl

[........ ........ ........ ........] edx
[........ ........] dx
[........] dh
[........] dl

[........ ........ ........ ........] esi
[........ ........ ........ ........] edi

Referred to as %eax, %ebx, %ecx, %edx, %esi, %edi, etc.

INSTRUCTION: mov SOURCE, DESTINATION

definition: moves "SOURCE" into "DESTINATION"

commonly has trailing character that indicates size of move, e.g.,
 movb - move a byte
 movl - move "long" or 4 bytes (that’s an L after mov, not a one)
 movq - quad or 8 bytes

our focus: movl (mostly)

Initial (limited) usage
- source=number ("immediate")  destination=register
  e.g., mov $10, %eax
- source=register               destination=register
  e.g., mov %eax, %ebx

Later, we will add different types of operands for mov
INSTRUCTION: **addl SOURCE, DESTINATION**

definition: adds SOURCE and DESTINATION, puts result into DESTINATION
i.e., DESTINATION = DESTINATION + SOURCE

limited usage (for now):
- source=number ("immediate")  destination=register
- source=register               destination=register

INSTRUCTION: **subl SOURCE, DESTINATION**

definition: DESTINATION = DESTINATION - SOURCE

limited usage (for now):
- source=number ("immediate")  destination=register
- source=register               destination=register

INSTRUCTION: **imull SOURCE, DESTINATION**

definition: DESTINATION = DESTINATION * SOURCE

alternate:
  imull AUX, SOURCE, DESTINATION
definition: DESTINATION = AUX * SOURCE

limited usage (for now):
- source=number ("immediate")  destination=register
- source=register               destination=register
- (aux=immediate)

INSTRUCTION: **idivl DIVISOR**

definition: contents of %edx:%eax (64 bit number) divided by DIVISOR
  quotient  -> %eax
  remainder -> %edx

limited usage (for now):
- divisor=register

Notes: A bit weird in its usage of VERY SPECIFIC registers!
**Problem #1**
Write assembly to:
- move value 1 into %eax
- add 10 to it and put result into %eax

**Problem #2**
Expression: 3 + 6 * 2
Use one register (%eax), and 3 instructions to compute this piece-by-piece

**Problem #3**
```
movl $0, %edx
movl $7, %eax
movl $3, %ebx
idivl %ebx
movl %eax, %ecx
movl $0, %edx
movl $9, %eax
movl $2, %ebx
idivl %ebx
movl %edx, %eax
addl %ecx, %eax
```
Write simple C expression that is equivalent to these instructions
Many x86 instructions can refer to memory addresses; these addresses take on many different forms.

**ABSOLUTE/DIRECT addressing**

definition: just use a number as an address

movl 1000, %eax
gets contents (4 bytes) of memory at address 1000, puts into %eax

NOTE: DIFFERENT than movl $1000, %eax
(which just moves the VALUE 1000 into %eax)

**INDIRECT addressing**

definition: address is in register

movl (%eax), %ebx
treat contents of %eax as address, get contents from that address, put into %ebx

**BASE + DISPLACEMENT addressing**

definition: address in register PLUS displacement value (an offset)

movl 8(%eax), %ebx
address = 8 + contents of eax
get contents from that address, put into %ebx

**INDEXED addressing**

definition: use one register as base, other as index

movl 4(%eax, %ecx), %ebx
address = 4 + contents[eax] + contents[ecx]
get contents from that address, put into %ebx

**SCALED INDEXED addressing (most general form)**

definition: use one register as base, other as index, scale index by constant (e.g., 1, 2, 4, 8)

movl 4(%eax, %ecx, 8), %ebx
address = 4 + contents[eax] + 8*contents[ecx]
get contents from that address, put into %ebx
Problem #4 (from CSAPP 3.1)

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

Registers

%eax 0x100
%ecx 0x1
%edx 0x3

Value of:
%eax
0x104
$0x108
(%eax)
4(%eax)
9(%eax, %edx)
260(%ecx, %edx)
0xFC(%ecx, 4)
(%eax, %edx, 4)
New register to help with stack: esp (extended stack pointer)

Referred to as %esp

```
[........ ........ ........ ........] eax 32 bits
[........ ........] ax 16 bits
[........]  ah 8 bits
[........]  al 8 bits

[........ ........ ........ ........] ebx
[........ ........] bx
[........]  bh
[........]  bl

[........ ........ ........ ........] ecx
[........ ........] cx
[........]  ch
[........]  cl

[........ ........ ........ ........] edx
[........ ........] dx
[........]  dh
[........]  dl

[........ ........ ........ ........] esi
[........ ........ ........ ........] edi

[........ ........ ........ ........] esp 32 bits
[........ ........ ........ ........] eip 32 bits
```

Points to "top of stack" when program is running
Changes often (room for local variables, function call/return, etc.)

Can use normal instructions to interact with it, e.g., addl, subl
Can also use special instructions (we’ll see this later)

---

**Problem #5**

Use instructions to:
- Increase size of stack by 4 bytes
- Store an integer value 10 into the top of the stack
- Retrieve that value and put it into %ecx
- Add 5 to it
- Put final value into %eax
Condition codes: new bits in hidden %eflags register.

Some instructions set those bits based on comparisons:
- cmp, test
Other instructions change control flow (%eip) based on results:
- jmp family

**INSTRUCTION:** `cmpl B, A`

computes A-B (but doesn’t put result anywhere)

condition codes (incomplete):
- zero flag : ZF=1 if (A-B) == 0 otherwise ZF=0
- signed flag : SF=1 if (A-B) < 0 otherwise SF=0

**INSTRUCTION:** `jmp` TARGET always changes %eip to TARGET

**INSTRUCTION:** `je` TARGET %eip=TARGET if ZF==1

**INSTRUCTION:** `jne` TARGET %eip=TARGET if ZF== ______

**INSTRUCTION:** `jg` TARGET %eip=TARGET if __________

**INSTRUCTION:** `jge` TARGET %eip=TARGET if __________

**INSTRUCTION:** `jl` TARGET %eip=TARGET if __________

**INSTRUCTION:** `jle` TARGET %eip=TARGET if __________
Problem #6
Assume value of a is in %eax, and value of b is in %ebx
Write x86 assembly code for:
   if (a > b) {
      a++;
   }

Problem #7
Assume value of a is in %eax, and value of b is in %ebx
Write x86 assembly code for:
   if (a > b) {
      a++;
   } else {
      b = a;
   }

Problem #8
Assume value of a is in %eax, and value of b is in %ebx
Write x86 assembly code for:
   while (b > 0) {
      a++;
      b--;
   }