Assembly Reference

This is a reference for **Intel syntax** x86 assembly.

General-Purpose Registers

The following are general-purpose 32-bit registers that you can use in your calculations: `eax`, `ebx`, `ecx`, `edx`, `edi`, `esi`.

The low 16 bits of each of these registers is available under the following names: `ax`, `bx`, `cx`, `dx`, `di`, `si`.

The low 8 bits of each of these registers is available under the following names: `al`, `bl`, `cl`, `dl`, `dil`, `sil`.

Arithmetic

- `add op1, op2`
  - Adds two the two operands, storing the result in the first operand.
  - e.g., `add eax, 1` is equivalent to `eax = eax + 1`.
- `sub op1, op2`
  - Subtracts the two operands, storing the result in the first operand.
  - e.g., `sub eax, ebx` is equivalent to `eax = eax - ebx`.
- `imul op1, op2`
  - Multiplies the two operands, storing the result in the first operand.
  - e.g., `imul ebx, -1` is equivalent to `ebx = ebx  * -1`.
- `lea op1, [op2 + op3 * op4 + op5]`
  - All-in-one math instruction. Computes `op2 + op3 * op4 + op5`, storing the result in `op1`.
  - **This instruction does not access memory**. Don't let the brackets fool you!
  - There are significant limitations on the valid operands to this instruction. Play with it, and see what assembles.
  - e.g., `lea eax, [ebx + esi * 8 - 1]` is equivalent to `eax = ebx + esi * 8 - 1`.
- `inc op`
  - Adds 1 to the operand.
  - e.g., `inc eax` is equivalent to `eax++`.
- `dec op`
  - Subtracts 1 from the operand.
  - e.g., `dec eax` is equivalent to `eax--`.

Bitwise Operations

- `shl op1, op2`
  - Shifts the first operand left by the number of bits specified in the second operand.
  - e.g., `shl eax, 5` is equivalent to `eax = eax << 5`.
- `shr op1, op2`
  - Shifts the first operand right by the number of bits specified in the second operand.
  - e.g., `shr edi, ebx` is equivalent to `edi = edi >> ebx`.
- `xor op1, op2`
  - XORs the two operands, storing the result in the first operand.
  - e.g., `xor ecx, ecx` is equivalent to `ecx = ecx ^ ecx`.
- `or op1, op2`
  - ORs the two operands, storing the result in the first operand.
  - e.g., `or eax, 0x0000FFFF` is equivalent to `eax = eax | 0x0000FFFF`.
- `and op1, op2`
  - ANDs the two operands, storing the result in the first operand.
  - e.g., `and edx, ecx` is equivalent to `edx = edx & ecx`.
- `not op`
  - Flips all the bits in the operand.
  - e.g., `not ebx` is equivalent to `ebx = ebx ^ 0xFFFFFFFF`.

Stack Operations

- `push op`
  - Pushes `op` onto the stack.
- `pop op`
  - Pops `op` from the stack

Subroutine Operations

- `call label`
  - Calls the subroutine at `label`.
- `ret`
  - Returns from a subroutine.

Reading/Writing

- `mov op1, op2`
  - Copies the value of the second operand into the first operand. The first operand must be either a register or memory location.
  - e.g., `mov esi, 5` is equivalent to `esi = 5`.
  - e.g., `mov BYTE PTR [esi], 5` is equivalent to `*(uint8_t *)esi = 5`.
- `movsx op1, op2`
  - Copies the value of the second operand into the first operand, sign-extending. The second operand must
  - e.g., `movsx ebx, al` is equivalent to `ebx = (int8_t)al < 0 ? 0xFFFFFF00 | al : 0x00000000 | al`
- `movzx op1, op2`
  - Copies the value of the second operand into the first operand, zero-extending.
  - e.g., `movzx ebx, al` is equivalent to `ebx = 0x00000000 | al`

Labels and Unconditional Jumps

A label is a name attached to a location in your assembly code.

- `jmp label`
  - Takes a label as its operand, and changes sets the instruction pointer to that label's value.
  - This should remind you of `goto` from C.

For example, the following is an infinite loop that increments `ebx` forever:

```
my_label_name:
    add ebx, 1
    jmp my_label_name
```

Comparison

- `cmp op1, op2`
  - Compares the two operands via subtraction so that conditional jumps can be executed.
- `test op1, op2`
  - Compares the two operands via bitwise AND so that conditional jumps can be executed.
  - You'll most commonly see `test` used with `op1` and `op2` being the same. This is roughly equivalent to `cmp op1, 0`, but is slightly faster.

Conditional Control Flow

- `jg label`
  - Jumps to `label` if the first operand of the preceding `cmp` instruction was greater than the second operand (signed).
- `jge label`
  - Jumps to `label` if the first operand of the preceding `cmp` instruction was greater than or equal to the second operand (signed).
- `jl label`
  - Jumps to `label` if the first operand of the preceding `cmp` instruction was less than the second operand (signed).
- `jle label`
  - Jumps to `label` if the first operand of the preceding `cmp` instruction was less than or equal to the second operand (signed).
- `je label`
  - Jumps to `label` if the first operand of the preceding `cmp` instruction was equal to the second operand.
- `jne label`
  - Jumps to `label` if the first operand of the preceding `cmp` instruction was not equal to the second operand.
- `ja label`
  - Jumps to `label` if the first operand of the preceding `cmp` instruction was greater than the second operand (unsigned).
- `jae label`

    - Jumps to `label` if the first operand of the preceding `cmp` instruction was greater than or equal to the second operand (unsigned).
- `jb label`
    - Jumps to `label` if the first operand of the preceding `cmp` instruction was less than the second operand (unsigned).
- `jbe label`
    - Jumps to `label` if the first operand of the preceding `cmp` instruction was less than or equal to the second operand (unsigned).

For example, the following will jump to `some_label` if `ebx` is 1, and do nothing otherwise:

```
cmp ebx, 1
je some_label
```

Miscellaneous

- `nop`
    - Short for "no operation." Does nothing.