1 Instructions

The following instructions are written using Unix syntax, where the destination comes last, and the \( \text{l} \) suffix on the instruction specifies that we are dealing with 32-bit quantities. Except where specified, sources can be immediate values prefixed by a dollar sign (e.g., \$32, \$-18), address labels (e.g., `Point__count`), registers (e.g., `%eax` or `%ebp`), contents of memory whose address is in a register (e.g., `(eax)`), or the contents of memory at a constant offset from an address in a register (e.g., `-8(ebp)`). Destinations cannot be constants or labels, but can be any of the other possibilities.

### Instructions

- **nop**
  - No-op; does nothing.

- **movl src, dest**
  - Move contents of source into destination. Special cases of **movl** include loading from and storing to memory, and (if src is an immediate) initializing a register or memory location. If you want to load a single byte rather than 32-bits, it's **movb**.

- **pushl src**
  - Push the given value on the stack (modifying `%esp`).

- **popl dest**
  - Push the given value on the stack (modifying `%esp`).

- **call *src**
  - Jump to the given address and push the return value on the stack (modifying `%esp`). The asterisk is optional when jumping to a label.

- **ret**
  - Pop return value off the stack (modifying `%esp`) and return.

- **jmp *src**
  - Jump to the given address (which can be a label constant). The asterisk is optional when jumping to a label.

- **j?? src**
  - Conditionally jump to the given address (often a label constant), where ?? is \( \text{e} \), \( \text{ne} \), \( \text{l} \), \( \text{g} \), \( \text{le} \), \( \text{ge} \), \( \text{b} \) (below, i.e., unsigned less-than), \( \text{a} \) (above, i.e., unsigned greater-than), \( \text{be} \), or \( \text{ae} \), based on the current condition flags.

- **cmpl src\(_2\), src\(_1\)**
  - Compare the two operands and set the condition flags. **WARNING:** in AT&T syntax (used by the Gnu Assembler), the comparison looks backwards! To jump when \( x \) is greater than \( y \), you would say **cmpl y, x** followed by **jg**.

- **???l src, dest**
  - Apply binary operator to source and destination (NB: destination is used a source too!) and put the result in the destination, where **??** is **add**, **sub**, **and**, or **xor**.

- **???l dest**
  - Apply unary operator, where **??** is **not** or **neg**.

- **imull src, dest**
  - Signed multiplication of source and the destination. Destination must be a register and holds only the low 32 bits of the potentially 64-bit result.

- **idivl src**
  - Divides the 64-bit number `%edx%eax` by the source, which cannot be an immediate. Quotient goes into `%eax` and remainder into `%edx`. (For 32-bit arithmetic, you call this right before **idiv**.)

- **cdq**
  - Sign-extends the 32-bit number `%eax` into the 64-bit number `%edx%eax`, overwriting `%edx`. (For 32-bit arithmetic, you call this right before **idiv**.)

- **???l immed, dest**
  - Shift the bits in the destination the number of positions given by the immediate operand, where **??** is **shl**, **shr** (unsigned), or **sar** (signed).

- **???l %c1, dest**
  - Shift the bits in the destination the number of positions given by the value in register `%c1` (and only this register!), where `%c1` is the low 8 bits of the `%ecx` register, and **??** is one of the above shift instructions.
2 Assembly Language Files

- Labels can be specified by writing them followed by a colon. Labels must start with a letter or an underscore. Case is distinguished.
- Line comments begin with #
- Generated code is much easier to read if every instruction is on its own line, prefixed by a tab character.
- Assembler directives are instructions to the assembler; they begin with periods (the same way that preprocessor directives begin with # in C or C++).
- Code belongs in the text segment, and initialized data belongs in the data segment. At any time you can switch between these with the assembler .text or .data (note the leading periods). If your data is read-only, you may prefer to put it in a special read-only data segment, which can be specified by .section .rodata.
- The directive .long n tells the assembler to put the 32-bit constant n into the next location. (Normally one would put this in the data segment, immediately following a label.) n can actually be any value that can be determined at compile-time, including a label (in which case that address is stored). If you just want a single byte, the directive is .byte b.
- The directive .ascii "..." puts the given string into the following bytes.
- You can tell the assembler to insert padding to ensure the next line falls on a multiple of $2^n$ bytes with the directive .align n.
- The directive .globl l tells the assembler that the label l should be part of the external interface of the generated object code, so that other code can call it. Otherwise, all labels are treated as being local to the assembly file. (Hint: your Start.main function will be called from outside).
- Historically, there are two standard ways of writing assembly code for the x86: Intel’s original syntax, and AT&T syntax. The Gnu assembler that we are using uses AT&T syntax. By far the biggest difference is whether the source comes before the destination or after (does mov a, b mean a ← b [Intel] or a → b [AT&T]). There are other minor syntactic differences as well. (Intel uses square brackets rather than parentheses, and doesn’t need $ or % prefixes or the l suffix.). For example, here are two ways to write the same store-to-memory instruction:
  AT&T/Gnu  movl $5, 4(%eax)
  Intel      mov [eax + 4], 5
- When you see assembly language that you didn’t write (e.g., the output of a disassembler, or documentation on a web page), be sure to figure out which syntax it’s using!
- This swapped-argument convention is also why operands to cmp look backwards in AT&T/Gnu syntax.

3 Other sources of information:

- The Gnu Assembler manual (which discusses directives and a few syntactic issues, but not x86 instructions specifically) http://sourceware.org/binutils/docs/as/index.html
- You can run C or C++ files through g++ -S -march=i386 -m32 and look at the resulting .s file. (On WILKES.CS, you don’t need the last two flags.)
- The nice table of instructions at http://www.jegerlehner.ch/intel/IntelCodeTable.pdf, except that it uses Intel’s (backwards, compared to AT&T/Gnu) order of operands.