S4 - x86 Assembly

Date: 2013-05-04
tags: hwswint

Moving Data

- `movx <source> <dest>`
- x is one of {b, w, l}
- movl <source> <dest> - Move 4-byte "long word"
- movw <source> <dest> - Move 2-byte "word"
- movb <source> <dest> - Move 1-byte "byte"
- movq <source> <dest> - Move 8-byte "quad word" (only in x86-64)

<source> and <dest> are operands. There are 3 types of operands:

- **Immediate: Constant integer data**
    - E.g. `$x400`, `$-533`
    - Like C constant, but prefixed with $
    - Encoded with 1,2 or 4 bytes
- **Register: One of 8 integer registers**
    - E.g. `%eax`, `%edx`
    - But `%esp` and `%ebp` are reserved for special use
    - Others have special uses for particular instructions
- **Memory: 4 consecutive bytes of memory at address given by register**
    - Simplest example: (%eax) (this means store the data in the memory address that is stored in %eax register)
    - Various other "address modes"

movl Operand Combinations

Source	Dest	Source, Dest	C Analog
Imm	Reg	`movl $0x4, %eax`	`var a = 0x4`
Imm	Mem	`movl $-147, (%eax)`	`*p_a = -147`
Reg	Reg	`movl %eax, %edx`	`var_d = var_a`
Reg	Mem	`movl %eax, (%edx)`	`*p_d = var_a`
Mem	Reg	`movl (%eax), %edx`	`var_d = *p_a`

Cannot do memory-memory transfer with a single instruction.

Memory Addressing Modes

Indirect: (R) refers to Mem[Reg[R]] where register R specifies a memory address

E.g.: `movl (%ecx), %eax`

Displacement: D(R) refers to Mem[Reg[R]+D] where R specifies a memory address (e.g. the start of some memory region). Constant displacement D specifies the offset from that address.

E.g.: `movl 8(%ebp), %edx` psuedo-equivalent to %edx = Mem[%ebp + 8]

Watch `5 - 1 - Moving Data (1738).mp4` for an assembly program example.

Watch `5 - 3 - Memory Addressing Modes (1422).mp4`.

Conditionals and Control Flow

- **A condition branch is sufficient to implement most control flow structs offered in high level languages like**
    - `if (condition) then {...} else {...}`
    - `while (condition) {...}`
    - `do {...} while (condition)`
    - `for (initialization; condition; iterative) {...}`
- **Unconditional branches implement some related control flow constructs**
    - break, continue
- In x86, we'll refer to branches as "jumps" (either conditional or unconditional)

Jumping: `jX` is used to jump to different part of the code depending on the condition codes such as `jmp` for unconditional, `je` for equal/zero, etc.

When a branch instruction is executed, value of instruction pointer `%eip` is changed to one of the various condition codes:

- `CF` Carry Flag (for unsigned)
- `ZF` Zero Flag
- `SF` Sign Flag (for signed)
- `OF` Overflow Flag (for signed)

These condition codes can be set implicitly as a side-effect of arithmetic operations such as `addl <source> <destination>` <--> `t = a+b`.

- `CF` set if carry out from MSB (unsigned overflow)
- `ZF` set if `t == 0`
- `SF` set if `t < 0` (as signed)
- `OF` set if two's complement (signed) overflow

Also, these condition codes can be set explicitly using "compare" (`cmpl/cmpq`) instruction such as `cmpl b, a` <--> `a - b`. Here, there result is not directly stored anywhere, rather the condition code is determined by the result and that is stored in instruction pointer.

- `CF` set if carry out from MSB (unsigned overflow)
- `ZF` set if `a == b` (essentially same as `a - b == 0`)
- `SF` set if `(a-b) < 0` (as signed) (or `a < b`)
- `OF` set if two's complement (signed) overflow

Condition codes can be set explicitly using "test" (`testl/testq`) instruction such as `testl a, b` <--> `a & b` (bitwise &) without setting destination.

- Sets condition codes based on value of src1 & src2

- Useful to have one of the operands be a mask
- `ZF` set if a&b `== 0`
- `SF` set if a&b `< 0`

Reading Condition Codes

`setX` instruction can be used to read condition code from condition code register and store it into a general purpose register.