CSE 351 Lecture 9 – x86 Programming II

Address Computation Instruction (`lea`)

Because sometimes we don't want to dereference a memory operand and want to use the address instead, we have one special instruction in x86-64 known as *load effective address* (`lea`). Its source operand *must* be a memory operand and its destination operand *must* be a register operand:

```
lea D(Rb,Ri,S), R  # stores Reg[Rb]+Reg[Ri]*S+D in Reg[R]
```

This is the ONLY instruction that does NOT dereference its memory operand – instead just using the computed address as the result. As the name suggests, its result does not have to actually be used as an address (it's an "effective address"), so `lea` can be used to perform arithmetic computations that fit the address calculation format. For example, take the instruction:

Example: `lea (%rdi,%rsi,4), %rax  # x in %rdi, y in %rsi, z in %rax`

- If we have `int* x` and `long y`, then this is equivalent to `int* z = &x[y];`
- If we have `long x` and `long y`, then this is equivalent to `long z = x + 4*y;`

Condition Codes

Condition codes are status bits that are part of the CPU state that indicate information about the most recently executed assembly instructions. They can be thought of as multiple single-bit registers, though they are actually part of a larger EFLAGS register in x86-64. The four condition codes that we will focus on are the *Carry Flag* (CF), *Zero Flag* (ZF), *Sign Flag* (SF), and *Overflow Flag* (OF).

These flags are set *implicitly* (*i.e.* as a side effect) by arithmetic and logical operations and indicate whether or not the result had unsigned overflow (CF), was zero (ZF), was negative (SF), or had signed overflow (OF). So, while the instruction's main objective is to store the result in the destination operand, the values of the condition codes are being automatically updated based on the result.

Example: If `0x80` is stored in `%al`, then `addb %al, %al` would update the value in `%al` to `0x00`, but *also* set `CF = 1`, `ZF = 1`, `SF = 0`, and `OF = 1`.

These flags can also be set *explicitly* by two special instructions: compare (`cmp`) and test (`test`). The purpose of these instructions is just to update the condition codes and their results are never stored. `cmp` produces a result equivalent to the `sub` instruction and `test` produces a result equivalent to the `and` instruction, with the condition codes being set as they would for those instructions.

The reason that we care about the condition codes is that their values are used to determine the outcome of two families of instructions: `jump` and `set`. It is through the use of these families of instructions that we are able to *implement all control flow* – unconditional and conditional jumps allow us to construct more abstract constructs such as if-else, looping, and switch statements.

Jump and Set

The table below show both the `jump` and `set` family of instructions (note that there is no unconditional `set`). We will concern ourselves with the "Description" column rather than the "Condition" column – it's good to know that the effects of these instructions depend on the current value of the condition codes, but you don't need to know the exact logical expressions.

Jump Instr	Set Instr	Condition	Description
`jmp` *target*	—	`1`	Unconditional
`je` *target*	`sete` *dst*	`ZF`	Equal to `0`
`jne` *target*	`setne` *dst*	`~ZF`	Not Equal to `0`
`js` *target*	`sets` *dst*	`SF`	Negative
`jns` *target*	`setns` *dst*	`~SF`	Nonnegative
`jg` *target*	`setg` *dst*	`~(SF^OF)&~ZF`	Greater Than `0` (Signed)
`jge` *target*	`setge` *dst*	`~(SF^OF)`	Greater Than or Equal To `0` (Signed)
`jl` *target*	`setl` *dst*	`(SF^OF)`	Less Than `0` (Signed)
`jle` *target*	`setle` *dst*	`(SF^OF)|ZF`	Less Than or Equal To `0` (Signed)
`ja` *target*	`seta` *dst*	`~CF&~ZF`	Above `0` (unsigned ">")
`jb` *target*	`setb` *dst*	`CF`	Below `0` (unsigned "<")

The difference between these families of instructions is that the `jump` family will jump our program to the specified *target* (*i.e.* change which instruction we execute next) if the condition is met, while the set family will set the value of the 1-byte *dst* register to the value of the condition (*i.e.* 0x00 or 0x01).

The condition can be thought of more intuitively as whether or not the description is true of the result of the last instruction that changed the condition codes (either implicitly via an arithmetic/logical instruction or explicitly via a compare/test instruction).

Extension Instructions (`movz` and `movs`)

Extension instructions are similar to a regular `mov` instruction, except that the source operand is smaller/shorter than the destination operand. `movz` and `movs` will perform the two types of extension we talked about with integers in C: *zero extension* and *sign extension*, respectively.

Unlike a normal `mov` instruction that takes one width specifier/instruction suffix, the extension instructions require two: the first for the source width and the second for the destination width.

Note: In x86-64, any instruction that generates a 32-bit value for a register (*e.g.* uses `%eax` as its destination operand) also sets the higher-order 32-bits of the register to all zeros. This is to maintain backwards compatibility with older IA32 code. See CSPP p. 184 for a good example.

Example: If `0x80` is currently stored in `%al`:

`movzbw %al, %bx` → `%bx` to **`0x0080`**, `movsbw %al, %bx` → `%bx` to **`0xFF80`**,

`movsbl %al, %ebx` → `%ebx` to **`0xFFFFFF80`** and `%rbx` to **`0x00000000FFFFFF80`**.