Compilers Course
Lecture 10: Target Machines

A typical target machine will look like this:

- A 32-bit [64-bit] RISC processor
- Integers are 32 bits [64 bits] wide
- A memory with 2^{32} [2^{64}] addressable bytes
- 2-, 4-, and 8-byte values (integers, floats) have a so-called "alignment restriction": they can only be loaded from or stored to addresses that are a whole multiple of the size of the value
- There are N (usually 32) general-purpose registers for integer and pointer values
- Likewise there are N floating-point registers
- Instructions operate on registers and constants:
    - reg := reg op (reg or const) [for +, -, <<, >>, ^, &, etc]
    - reg := op (reg or const) [unary minus, bitwise negation]
    - reg := const
    - reg := MEM[reg + (reg or const)] (loading values)
    - MEM[reg + (reg or const)] := reg (storing values)
    - flags := cmp reg (reg or const) (compare and set flags)
    - if cond(flags) goto label (jump on condition)
    - goto label
    - goto reg (indirect jump)
    - call label (save return address, then jump)
    - call reg (indirect call)
    - ret (indirect jump to return address)

This is typical for machines like MIPS, SPARC, PowerPC, ARM, etc.

The x86 is different:

- Two-address instructions: reg op= reg/const (+=, *=, -=, etc)
- Has complex memory operands: base reg + offset constant + (offset reg * 2/4/8)
- Allows memory destination operands: MEM[...] op= reg/const
- Allows memory source operands: reg op= MEM[...]
- The return address is not in a register but on the stack
- Misaligned loads/stores work but are slower than aligned load/stores
- Few general-purpose registers (8 in 32-bit mode, 16 in 64-bit mode)

What remains now for the compiler?

1. Assign variables to registers or memory addresses
2. Translate statements and expressions to sequences of instructions
3. Implement procedures, parameter passing, and recursion