Addressing mode in MIPS

Different formats of addressing registers or memory locations are called **addressing modes**.

- *Immediate addressing.* where operand is a constant in the instruction. e.g. `addi, lui, slti, andi, ori, sll, srl`
- *Register addressing.* when the operand is in a register. Simple, addresses location inside the processor. e.g. `add, sub, and, or, nor, jr`
- *Base addressing.* where operand is at a location = (16-bit constant in instruction) + (memory location stored in a register). Used for addressing elements of an array. e.g. `lw, sw, lh, sh, lb, sb`
- *PC-relative addressing.* when the address of the operand = PC + (16-bit constant shifted by 2). e.g.: branching instructions, `beq, bne`
- *Pseudo-direct addressing.* used for jump instruction. address = (26 bits shifted left = 28 bits) concatenated w/ upper 4 bits of PC. e.g. `j, jal`

Translating and Starting a program

Q. What happens when you compile a C program? When you run one?
There are 4 steps for transforming a C source code into a running program in memory: compiling, aseembling, linking, loading (accomplished by **systems programs**). Of course in an IDE, these steps are hidden from the user.
The steps taken are:

- *Preprocessing.* processing included header files, condition compilation (`ifdef`s), and macros
- **Compiler.** Produces an assembly language program, a symbolic form of machine (binary) language. Much more lines than the source code. Low-level code (OS, assemblers) were written in AL.
- **Assembler.** Translates the assembly program into *object file*: machine code + (global) data + information for placing instructions in memory properly.
  - **header.** size and position of sections in `.o` file.
  - **text.** contains machine code
  - **static data.** Data that will available for the lifetime of program
    - `.bss` uninitialized global data
    - `.data` initialized global data
    - `.rodata` read-only global data. string literals and constants.
  - **relocation information.** instructions and data that depend on absolute addresses when the program runs. For e.g. `j Label1`. Linker uses this info to adjust section contents. For e.g. the linker tracks the address of a procedure so other procedures may call it.

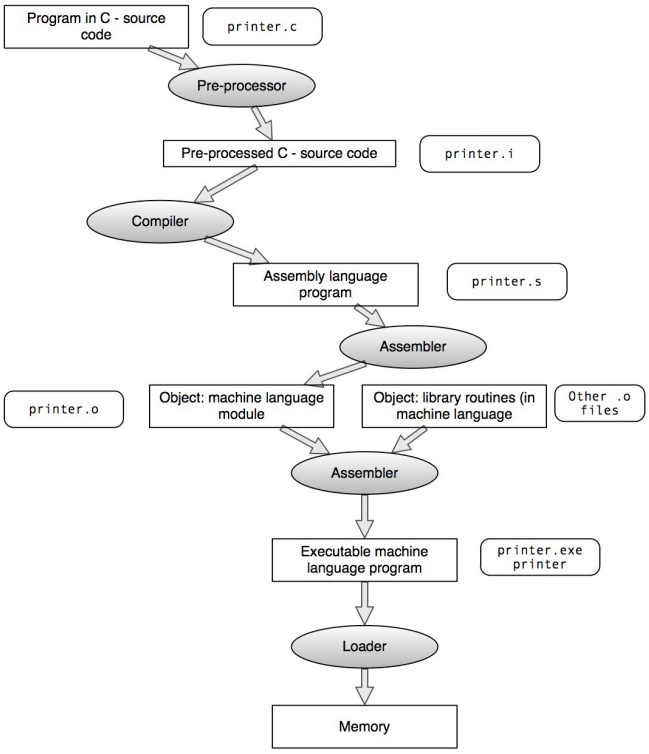

Figure 1: Steps for translating a C program

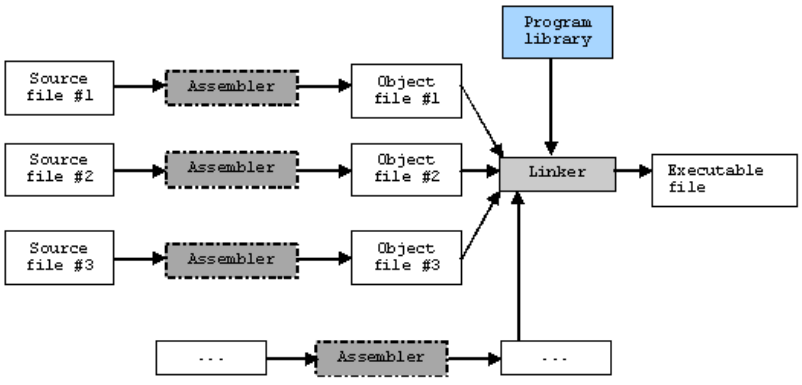

Figure 2: Linking the objects files

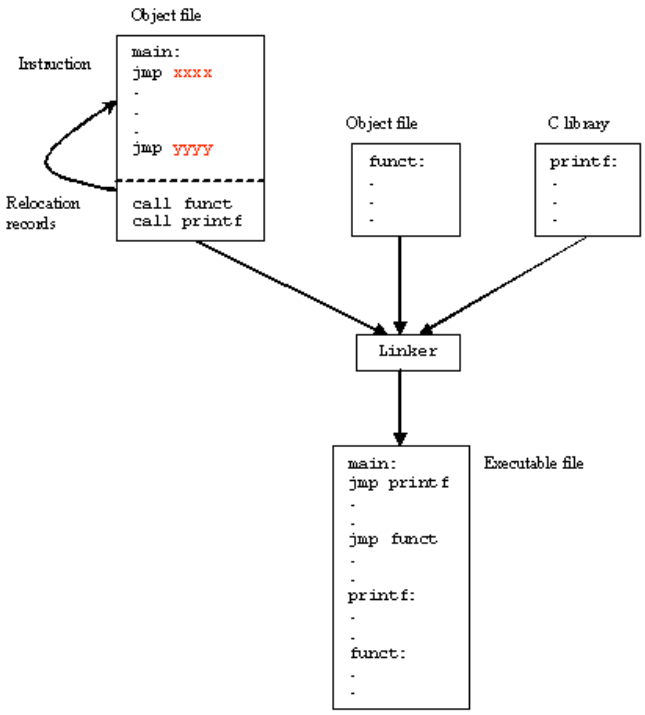

Figure 3: Use of relocation records

- **symbol table.** tracks labels. set of label symbols and their addresses. Since assembler needs to remove traces of all labels.
- **debugging information.** description of how code was compiled

Assembler also provides **pseudoinstructions** to make things easy for assembly code writer. For e.g. `move $t0, $t1` is a p-instruction that is translated as `add $t0, $zero, $t1`. Similarly, `blt` is translated into `slt` and `bne`, and so on. Assembler reserves register `$at` for its own purposes like these.

- **Linker.** Stitches all independent assembled, machine language programs into one.
  - Changing one line of code would require recompilation.
  - obvious for personal code. Very *wasteful for libraries*, since they never change.
  - Better:
    - Compile & assemble code separately so recompilation is contained
  - Linker also allows us to develop libraries in isolation
  - Three steps in linking
    - place code and data in memory *symbolically*
    - determine addresses of data and instruction labels
    - patch internal and external references
  - Uses *relocation info* and *symbol table* in each .o module to resolve undefined labels
  - produces an **executable file** w/ same format as .o file but with no unresolved references.
  - Statically linked vs. dynamically linked

    * with static approach: can be using old linked library; size issues

- **Loader.** Loads the executable in memory to start it
  - reads size of `text` and `data` segments
  - creates address space for them and copies them to the created space
  - copies arguments for `main` to the stack
  - initializes registers (sets to 0) sets SP to 1st free location on stack
  - calls a start-up routine that copies args to arg registers and calls the main routine.