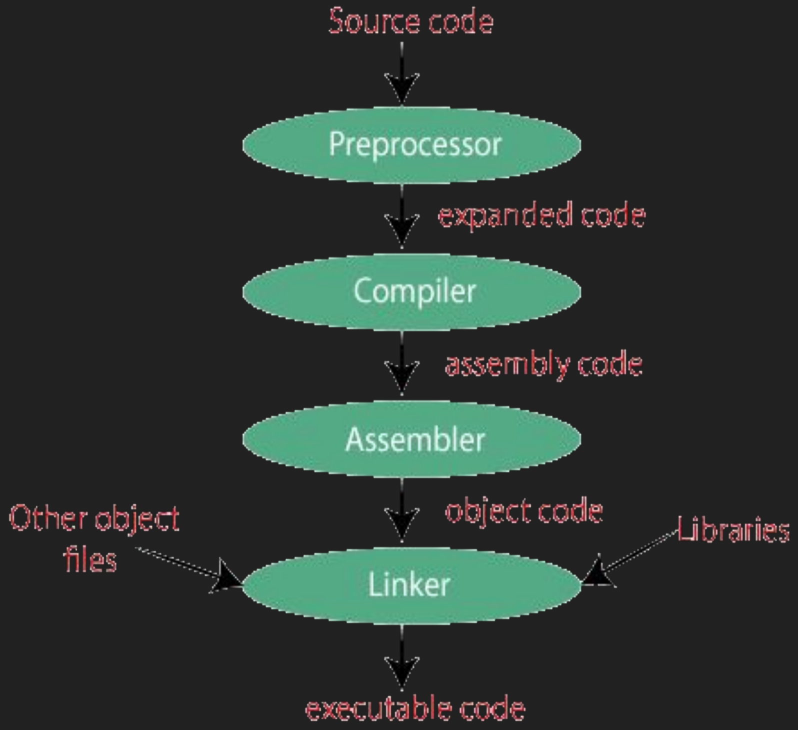
chibicc: A small C compiler

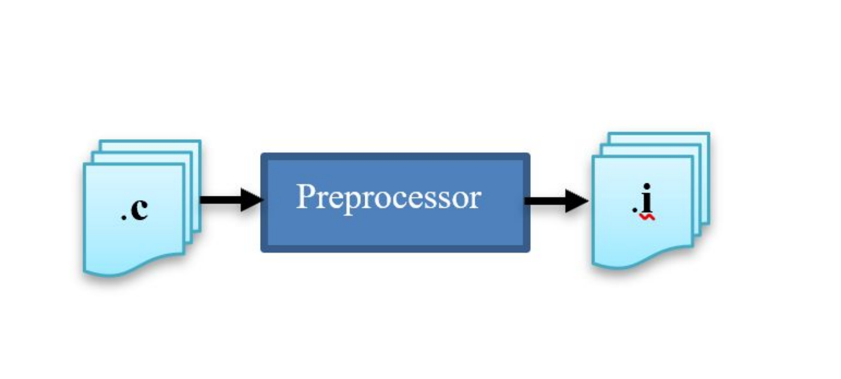
Jack Sangdahl, CSCI3155

Introduction

- A small C compiler named chibicc
- C compiler has multiple steps to produce something your computer can run
- Compilers you may already be familiar with: GCC, Clang/LLVM
- Next: how a compiler works



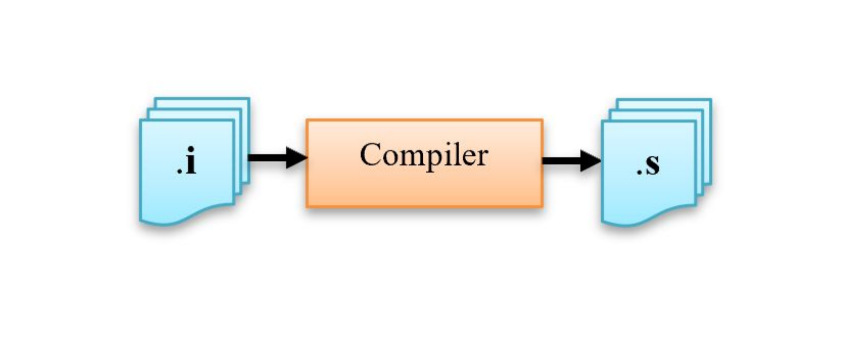
Compiler Internals – Preprocessor



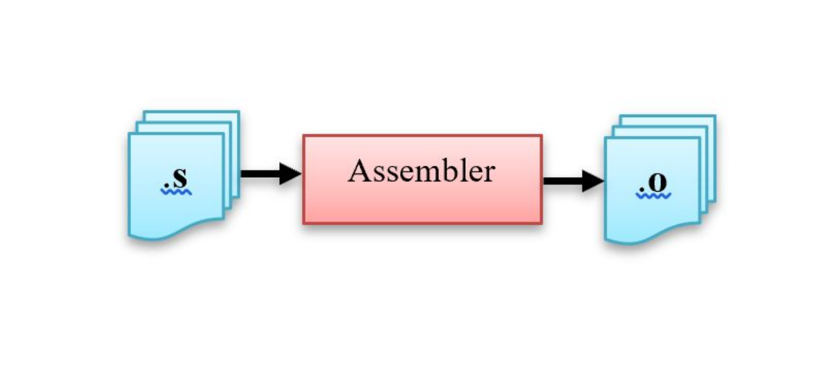
- Expands macros in the form of direct text replacement
- Replaces header file includes directly with the content of the file
- Resolves conditional directives like `#ifdef/#else`, and omits code that is not to be compiled

Compiler Internals – Compiler

- Converts preprocessor output to architecture specific Assembly code
- Output is closer to something readable by a computer, rather than a human
- Includes mnemonic symbols like `MOV & ADD` that directly represent CPU instructions



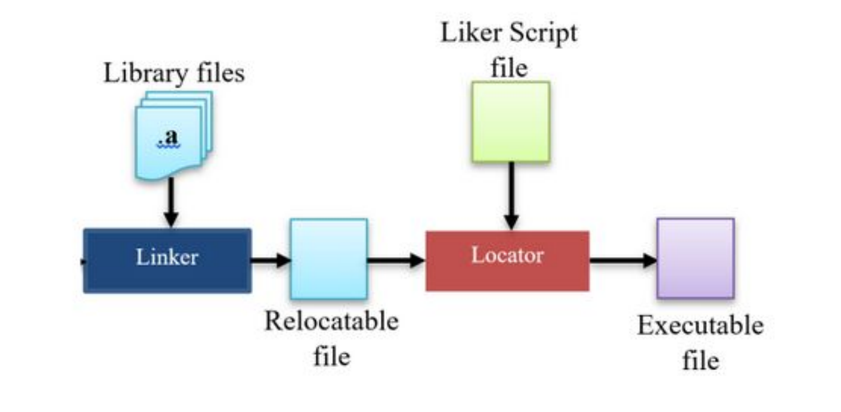
Compiler Internals – Assembler



- Assembly instruction consists of an opcode (eg. 0b0110)
- Followed by an operand, like a memory address
- At this stage, original C codes’ functions have been turned into machine code understandable by a CPU

Compiler Internals – Linker

- Merges all object code from previous step into a single executable file
- Resolves function call references in object files to actual symbols known by the OS
- Heavily architecture dependant and complex process
- Fun fact: this process requires significant amounts of recursion



Relating this to chibicc

- In common compiler infrastructures, the stages are delegated to individual programs (`cpp, as, ld`)
- Tools you may be familiar with (GCC/Clang) bundle all these steps into one program

```
$ cc foo.c

$ ./a.out
```

- chibicc provides a preprocessor and compiler. Assembling and linking is handled by aforementioned utilities
- Supports all mandatory features of C11, as well features like floating point numbers.
- Does not support GCC inline assembly or optimisation.

chibicc Internals

chibicc consists of the following stages:

1. **Tokeniser:** takes a string as input and breaks it into a list of tokens and returns this
2. **Preprocessor:** takes a list of tokens as an input, and outputs a new list of macro-expanded tokens
3. **Parser:** recursively descends and constructs abstract syntax trees from the preprocessor output.

   Also adds a type to each AST node
4. **Code Generator:** outputs an assembly text for given AST nodes

Observations & Comparisons

- Something interesting: chibicc does not free memory as needed, which may be a foreign concept
- Compilers are generally short lived and meant to run as fast as possible, which freeing memory impedes, so it’s simply done once at the end of execution
- chibicc of course does not produce binaries that run as fast as those made by larger compilers like GCC or Clang/LLVM
- chibicc is still able to compile large code bases like git, sqlite, and more