Compilation Pipeline

- Compiler, e.g., `lcc`

  translates from high-level language to assembly language
  consumes `.c` files, produces `.s` files
  some compilers produce object code directly

- Assembler, e.g., `as`

  translates from assembly language to machine language or object code
  consumes `.s` files, produces `.o` files

- Archiver, e.g., `ar`

  collects objects files into a single library
  consumes `.o` files, produces a `.a` file

- Linker/loader, e.g., `ld`

  links together object files and libraries into a single executable file or object file
  consumes `.o` files, produces a `.o` file or an `a.out` file

- Execution

  loads executable file into memory, starts the program

Assembly Languages

- Assembly language is a ***symbolic*** representation of ***virtual machine*** instructions
- Assemblers ***translate*** assembly language into ***object code***
- Object code contains the machine language instructions

  object files contain information needed to link, load, and execute the program

- Assembly language statements

  imperative statements specify instructions; “pure” assemblers map 1 imperative statement to 1 machine instruction

  some assemblers provide ***synthetic instructions***, which are mapped to several machine instructions depending on context, e.g., the SPARC assembler

  declarative statements specify “assembly-time” services, e.g., reserve space, define symbols, specify “segments” and scope (local vs. global), initialize data

  declarative statements do ***not*** yield machine instructions; they add “information” to the object file that is used by the linker

Assembly Languages, cont’d

- Most important function of an assembler is ***symbol manipulation***

  e.g., create labels and determine their addresses

- “forward-reference” problems

```
loop:   cmp i,n                       .seg    "text"
        bge done; nop                         set count,%l0
        ...                                   ...
        inc i                         .seg    "data"
        ba loop; nop                  count:  .long 0
done:
```

“value” of **done** is unknown when **bge** is assembled

address of **count** is unknown when **set** is assembled

- Most assemblers have ***two passes***

  pass 1: symbol definition

  pass 2: instruction assembly

  “pass” usually means reading the file, although it may also store/read a temporary file

- Other considerations, such as branch displacements, also may require two passes

Assembly Languages, cont’d

- Pass 1 constructs a symbol table with entries with name, type, value, attributes, etc., e.g., mapping of labels to values
- Pass 2 uses the symbol table to assemble and output instructions
- Opcodes may be a part of the symbol table or be a separate table; details depend on opcode structure and assembly language syntax
- Both passes maintain ***location counters*** that are used to determine the values of labels; a location counter is incremented by instruction lengths or data sizes
- High-level assembler structure

```
<assembler> ≡
        <initialize symbol table>
        pass1(symbol table)
        pass2(symbol table)
```

Assembler: Pass 1

- `pass1` builds the symbol table

```
void pass1(symbol table) {
    unsigned lc = 0;

    while (not EOF) {
        read a line
        save line in the temp file for pass 2
        if (line contains a label)
            enter(symbol table, label, lc)
        if (line contains a directive) {
            if (pass 1 directive)
                process directive
        } else
            lc += length(instruction)
    }
}
```

e.g., use `Table_get` and `Table_put`

might change `lc`

might involve inspecting instruction, operands, etc.

Assembler: Pass 2

- pass 2 reads the symbols built in pass 1

```
void pass2(symbol table) {
    unsigned lc = 0;

    while (not EOF) {
        read a line from the temp file
        if (line contains a directive) {
            if (pass 2 directive)
                process directive
        } else {
            assemble and output instruction using definitions in symbol table
            lc += length(instruction);
        }
    }
}
```

might change `lc`
emit output

may change some symbol
table entries, e.g., use `Table_get`

Assembler Features

- ***Multiple location counters***: programmer/compiler divides program into several ***logical segments*** using assembler directives, and each segment has its own location counter

```
.seg "text"
    A
.seg "data"
    B
.seg "text"
    C
.seg "data"
    D
```

assembler may concatenate segments on output

```
.seg "text"
    A
    C
.seg "data"
    B
    D
```

multiple location counters affects ***both*** passes; may appear in object files

- Multiple location counters may be simply logical segments to facilitate program organization or may be motivated by machine architecture

text segments are typically loaded into ***read-only*** memory and ***shared*** by other processes

data are loaded into ***read/write*** memory, ***one copy*** per process

Assembler Features, cont’d

- ***Macros***

  parameterized abbreviations for often-repeated instruction sequences

  conditional assembly

  no macros in UNIX assemblers; use the C preprocessor or **`m4`**

- ***One-pass assemblers***

  assemble instructions in first pass

  build a “fix-up table” for those instructions associated with undefined symbols

  as symbols are defined, fix the instructions given in the table and remove them from the table

  good for ***in-memory*** assemblers