Assembler

CS 217

Compilation Pipeline

- Compiler (**`gcc`**): **`.c`** → **`.s`**
  translates high-level language to assembly language
- Assembler (**`as`**): **`.s`** → **`.o`**
  translates assembly language to machine language
- Archiver (**`ar`**): **`.o`** → **`.a`**
  collects object files into a single library
- Linker (**`ld`**): **`.o`** + **`.a`** → **`a.out`**
  builds an executable file from a collection of object files
- Execution (**`execlp`**)
  loads an executable file into memory and starts it

Assembly Language

- A symbolic representation of machine instructions
- Assemblers translate assembly language into object code
- Object code contains everything needed to link, load, and execute the program

Assembly Language (cont)

- Assembly language statements…
  - imperative statements specify instructions; typically map 1 imperative statement to 1 machine instruction
  - some assemblers provide synthetic instructions that are mapped to one or more machine instructions
  - declarative statements specify *assembly time* actions; e.g., reserve space, define symbols, identify segments, and initialize data (they do not yield machine instructions but they may add information to the object file that is used by the linker)

Assembler

- Most important function: symbol manipulation
  - create labels and remember their addresses
- Forward reference problem

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


Assembler (cont)

- Most assemblers have two passes
  - Pass 1: symbol definition
  - Pass 2: instruction assembly

  where "pass" usually means reading the file, although it may store/read a temporary file

Pass 1

- State
  - **`lc`** (location counter); initially **`0`**
  - **`symtab`** (symbol table); initially empty
- For each line of input
  - if line contains a lable **`l`**
    - enter **`<l,lc>`** into **`symtab`**
  - if line contains a directive
    - adjust **`lc`** according to directive
  - else
    - **`lc += length_of_instruction`**

Pass 2

- State
  - reset **`lc = 0`**
- For each line of input
  - if line contains a directive
    - process directive (may change **`lc`**)
  - else
    - assemble and output instruction using **`symtab`**
    - **`lc += length_of_instruction`**

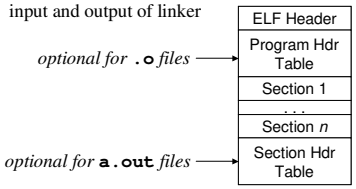
Directives

- Delineate segments
  - **`.section`**
  - may need multiple location counters (one per segment)
- Allocate/initialize data and bss segments
  - **`.word .half .byte`**
  - **`.ascii .asciz`**
  - **`.align .skip`**
- Make symbols in text externally visible
  - **`.global`**

ELF

- Format of `.o` and `a.out` files

  ELF: Executable and Linking Format
  output by the assembler
  input and output of linker



ELF (cont)

- ELF Header

```
typedef struct {
   unsigned char e_ident[EI_NIDENT];
   Elf32_Half    e_type;
   Elf32_Half    e_machine;
   Elf32_Word    e_version;
   Elf32_Addr    e_entry;
   Elf32_Off     e_phoff;
   Elf32_Off     e_shoff;
   ...
} Elf32_Ehdr;
```

ET_REL
ET_EXEC
ET_CORE

ELF (cont)

- Section Header Table: array of…

```
typedef struct {
   Elf32_Word   sh_name;
   Elf32_Word   sh_type;
   Elf32_Word   sh_flags;
   Elf32_Addr   sh_addr;
   Elf32_Off    sh_offset;
   Elf32_Word   sh_size;
   Elf32_Word   sh_link;
   ...
} Elf32_Shdr;
```

.text
.data
.bss

SHT_SYMTAB
SHT_RELA
SHT_PROGBITS
SHT_NOBITS

ELF (cont)

- Symbol Table Entry

```
typedef struct {
   Elf32_Word      st_name;
   Elf32_Addr      st_value;
   Elf32_Word      st_size;
   unsigned char   st_info;   --> STB_LOCAL
   unsigned char   st_other;      STB_GLOBAL
   Elf32_Half      st_shndx;
} Elf32_Sym;
```

symbol table section in `.o` file contains an array of such entries

ELF (cont)

- Relocation Entries

```
typedef struct {
   Elf32_Addr    r_offset;
   Elf32_Word    r_info;
   Elf32_Sword   r_addend;
} Elf32_Rela;
```

where `r_info` gives both the symbol table index and the type of relocation "edits" to apply