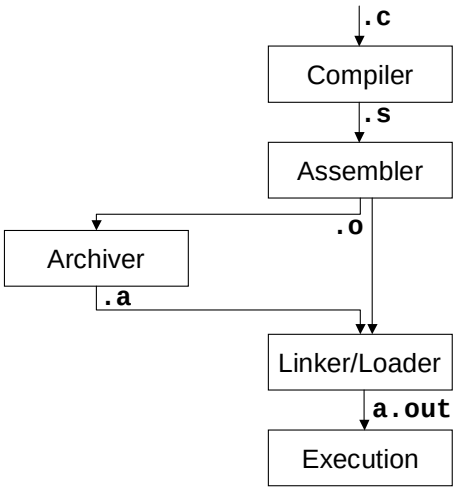
Assembler

CS 217

Compilation Pipeline

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

Compilation Pipeline



Assembler

- Purpose
  - Translates assembly language into machine language
  - Store result in object file (.o)
- Assembly language
  - A symbolic representation of machine instructions
- Object file
  - Contains everything needed to link, load, and execute the program

Assembly Language

- Assembly language statements…
  - imperative statements specify instructions; typically map 1 imperative statement to 1 machine instruction
  - synthetic instructions 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)

Main Task

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

```
loop: cmp i,n
      bge done
      nop
      ...
      inc i
done:
```

```
      .section ".text"
      set count, %l0
      ...
      .section ".data"
count: .word 0
```

Two-Pass Assemblers

- 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

```
loop: cmp i,n
      bge done
      nop
      ...
      inc i
done:
```

```
        .section “.text”
        set count, %l0
        ...
        .section “.data”
count:  .word 0
```

Pass 1

- State
  - loc (location counter); initially 0
  - symtab (symbol table); initially empty
- For each line of input ...

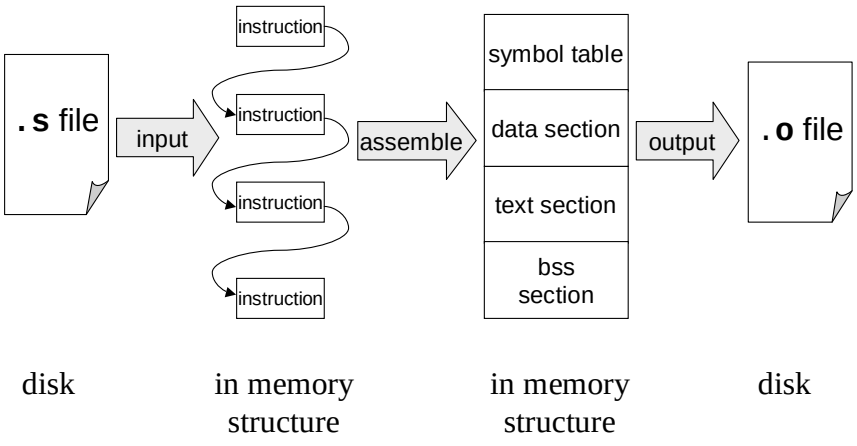
```
/* Update symbol table */
if line contains a label
   enter <label,loc> into symtab

/* Update location counter */
if line contains a directive
   adjust loc according to directive
else
   loc += length_of_instruction
```

Pass 2

- State
  - lc (location counter); reset to 0
  - symtab (symbol table); filled from previous pass
- For each line of input

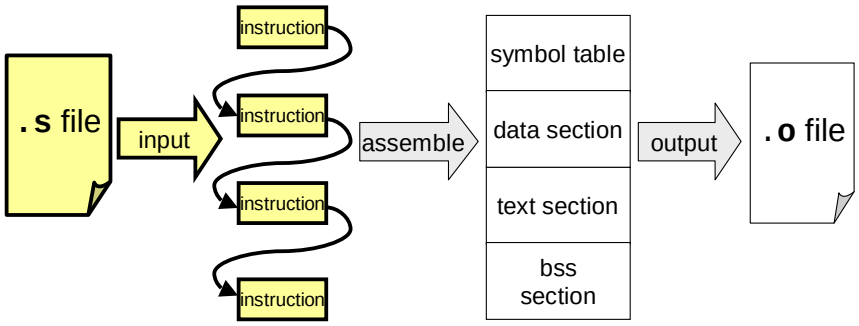
```
/* Output machine language code */
if line contains a directive
   process/output directive
else
   assemble/output instruction using symtab

/* Update location counter */
if line contains a directive
   adjust loc according to directive
else
   loc += length_of_instruction
```

Implementing an Assembler



Input Functions

- Read assembly language and produce list of instructions



These functions are provided

Input Functions

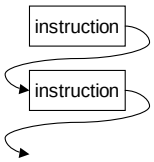
- Lexical analyzer
  - Group a stream of characters into tokens

```
add %g1 , 10 , %g2
```

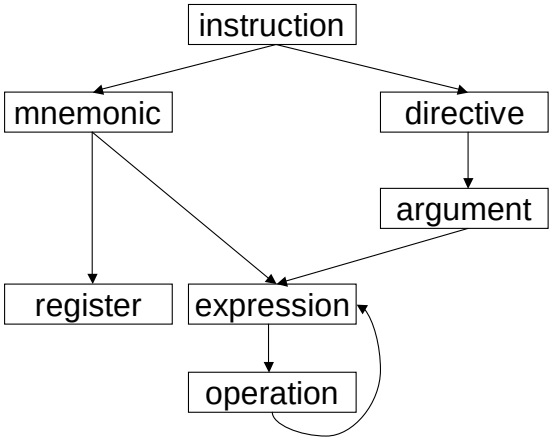
- Syntactic analyzer
  - Check the syntax of the program

```
<MNEMONIC><REG><COMMA><REG><COMMA><REG>
```

- Instruction list producer
  - Produce an in-memory list of instruction data structures



Input Data Structures

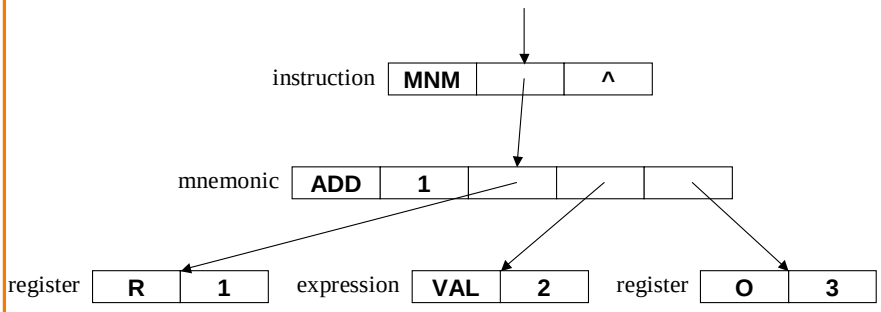


Input Data Structures (cont)

- Three types of assembly instructions
  - label (symbol definition)
  - mnemonic (real or synthetic instruction)
  - directive (pseudo operation)

```
struct instruction {
   int instr_type;  ──────────► LBL, MNM, DIR
   union {
      char *lbl;
      struct mnemonic *mnm;
      struct directive *dir;
   } u;
   struct instruction *next;
};
```

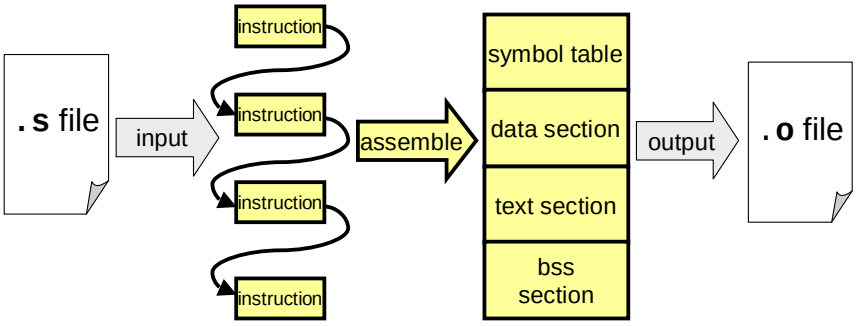
Example: `add %r1, 2, %o3`



Your Task in Assignment 5

- Implement two pass assembler to produce…

```
Table_T symbol_table;
struct section *data;
struct section *text;
struct section *bss;
```



Output Data Structures

- For symbol table, produce Table ADT, where each *value* is given by…

```
typedef struct {
   Elf32_Word    st_name;   = 0
   Elf32_Addr    st_value;  = offset in object code
   Elf32_Word    st_size;   = 0
   unsigned char st_info;   = see next slide
   unsigned char st_other;  = unique seq num
   Elf32_Half    st_shndx;  = DATA_NDX,
} Elf32_Sym;                  TEXT_NDX,
                              BSS_NDX, or
                              UNDEF_NDX
```

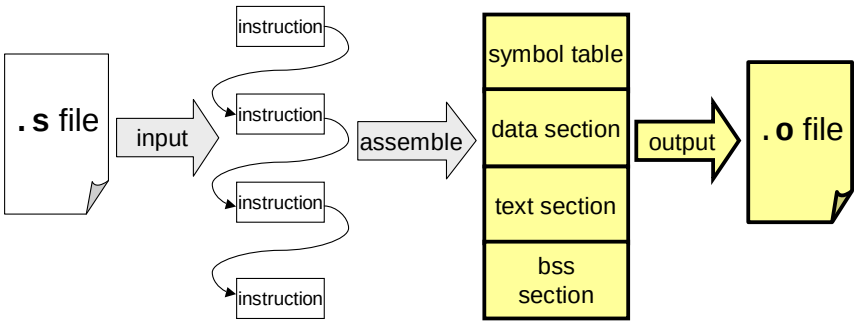
Output Data Structures (cont)

- For each section, produce…

```
struct section {
   unsigned int      obj_size;
   unsigned char     *obj_code;
   struct relocation *rel_list;
};
```

Output Functions

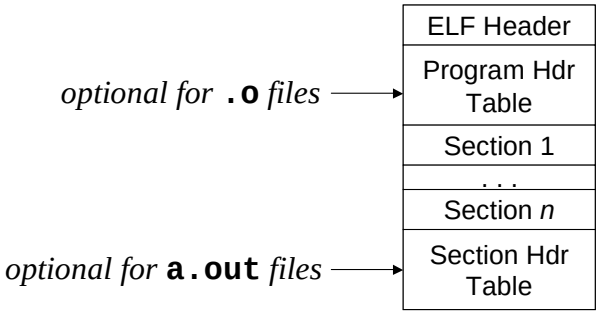
- Machine language output
  - Write symbol table and sections into object file (ELF file format)



This function is provided

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

Summary

- Assember
  - Read assembly language
  - Two-pass execution (resolve symbols)
  - Write machine language