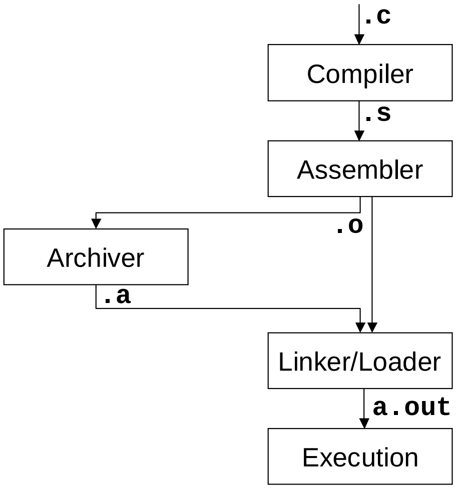
Assembler

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



.c

Compiler

.s

Assembler

.o

Archiver

.a

Linker/Loader

a.out

Execution

Assembler

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

Translating to machine code

- Assembly language: addcc %r3, %r7, %r2
  - addcc %r3, 1000, %r2
- Machine language:

10	rd	op3	rs1	0	unused(0)	rs2
31	29	24	18	13	12	4 0

10	rd	op3	rs1	1	simm13
31	29	24	18	13	12 0

10	00010	010000	00011	0	00000000	00111
31	29	24	18	13	12	4 0

10	00010	010000	00011	1	0001111101000
31	29	24	18	13	12 0

Assembly Language

- Assembly language statements…
  - 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)
  - imperative statements specify instructions; typically map 1 imperative statement to 1 machine instruction
  - synthetic instructions are mapped to one or more machine instructions

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
```

Dealing with forward references

- Most assemblers have two passes
  - Pass 1: symbol definition
  - Pass 2: instruction assembly
- Or, alternatively,
  - Pass 1: instruction assembly
  - Pass 2: patch the cross-reference

Dealing with forward references

- Most assemblers have two passes
  - Pass 1: symbol definition
  - Pass 2: instruction assembly
- Or, alternatively,
  - Pass 1: instruction assembly
  - Pass 2: patch the cross-reference

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

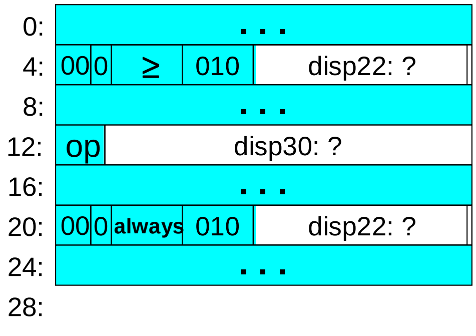
- State
  - lc (location counter); reset to 0
  - symtab (symbol table); filled from previous pass
- For each line of input

```
/* 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
```

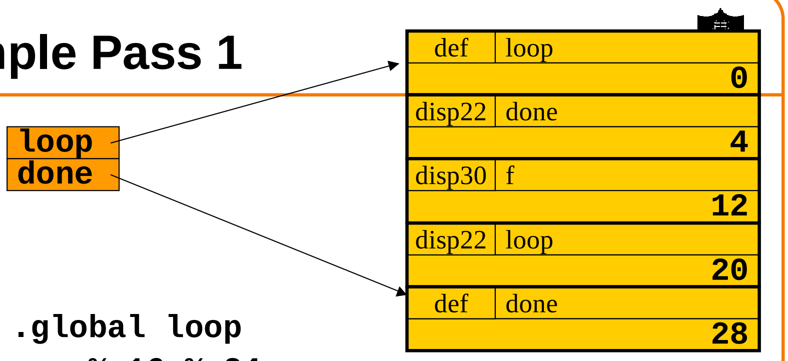
Example assembly

```
      .global loop
loop: cmp %r16,%r24
      bge done
      nop
      call f
      nop
      ba loop
      inc %r16
done:
```



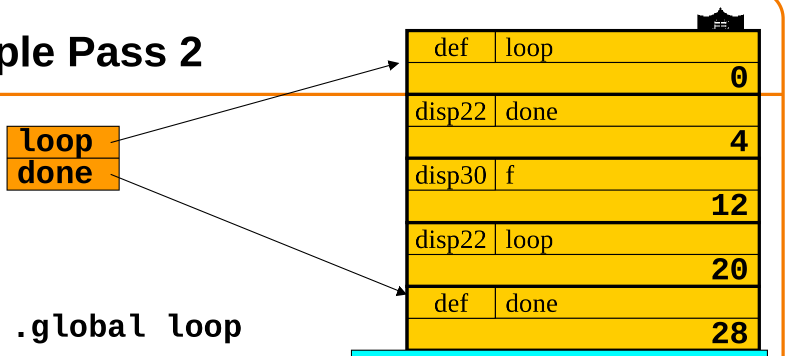
0: . . .
4: 00 | 0 | ≥ | 010 | disp22: ?
8: . . .
12: op | disp30: ?
16: . . .
20: 00 | 0 | always | 010 | disp22: ?
24: . . .
28:

Example Pass 1



```
        .global loop
loop:   cmp %r16,%r24
        bge done
        nop
        call f
        nop
        ba loop
        inc %r16
done:
```

Example Pass 2

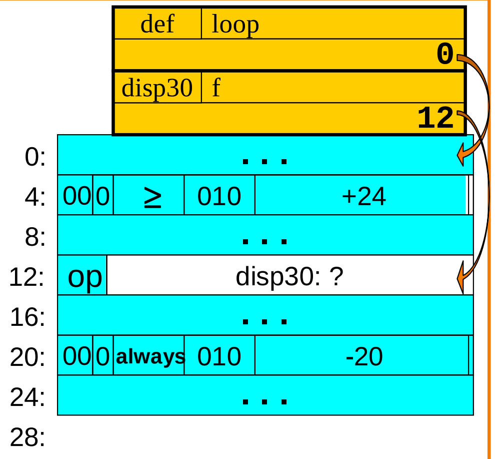


```
        .global loop
loop:   cmp %r16,%r24
        bge done
        nop
        call f
        nop
        ba loop
        inc %r16
done:
```

0:	. . .				
4:	00	0	≥	010	+24
8:	. . .				
12:	op	disp30: ?			
16:	. . .				
20:	00	0	always	010	-20
24:	. . .				
28:					

Relocation records

```
       .global loop
loop:  cmp %r16,%r24
       bge done
       nop
       call f
       nop
       ba loop
       inc %r16
done:
```



def | loop
0

disp30 | f
12

0: . . .

4: 00 | 0 | ≥ | 010 | +24

8: . . .

12: op | disp30: ?

16: . . .

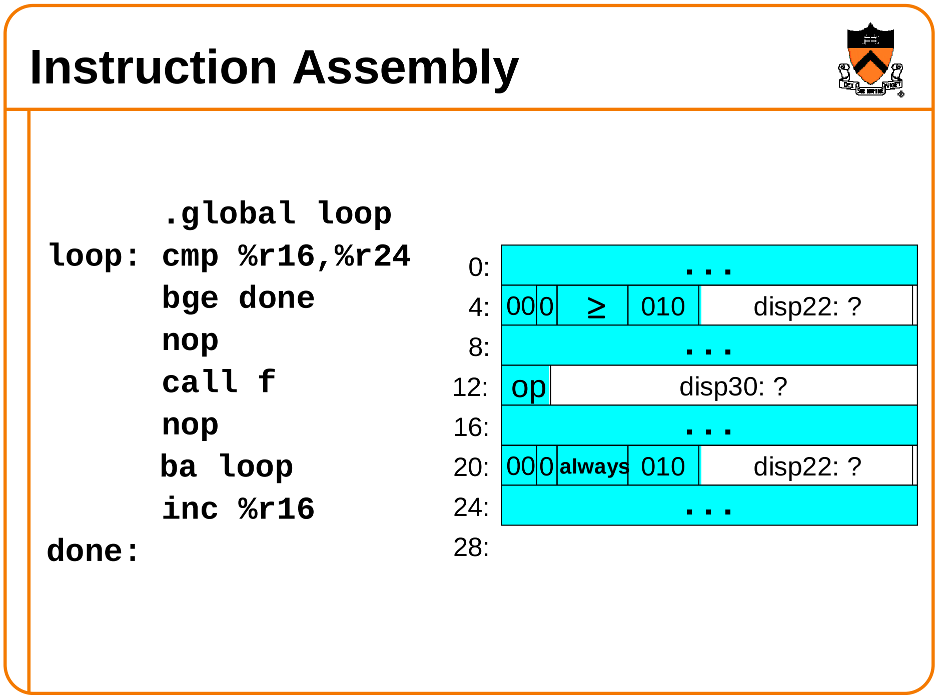
20: 00 | 0 | always | 010 | -20

24: . . .

28:

Dealing with forward references

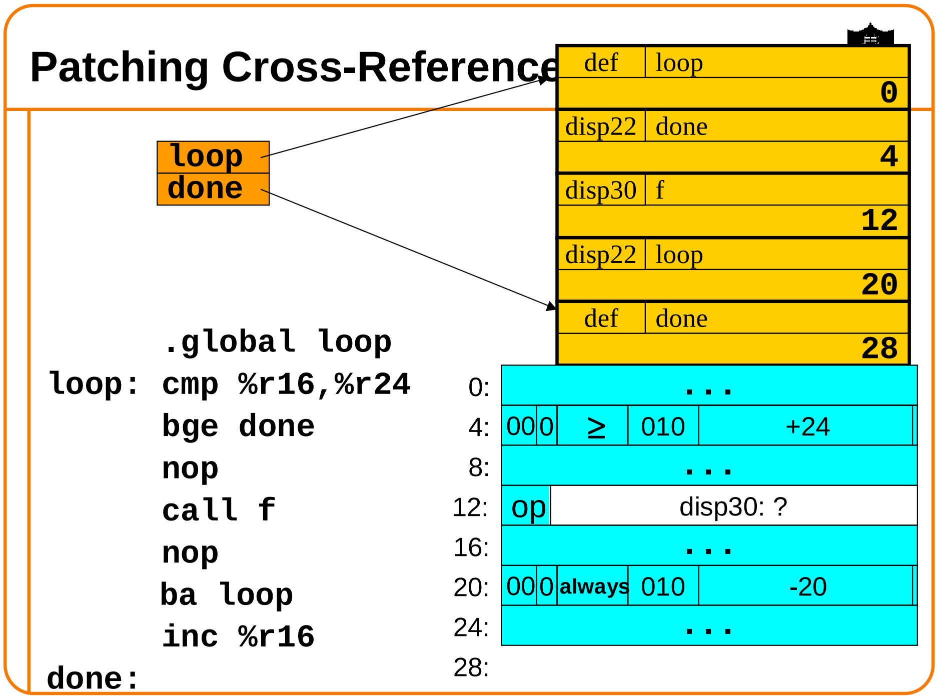
- Most assemblers have two passes
  - Pass 1: symbol definition
  - Pass 2: instruction assembly
- Or, alternatively,
  - Pass 1: instruction assembly
  - Pass 2: patch the cross-reference



Instruction Assembly

```
        .global loop
loop:   cmp %r16,%r24
        bge done
        nop
        call f
        nop
        ba loop
        inc %r16
done:
```

0:	. . .				
4:	00	0	≥	010	disp22: ?
8:	. . .				
12:	op	disp30: ?			
16:	. . .				
20:	00	0	always	010	disp22: ?
24:	. . .				
28:					



Patching Cross-References

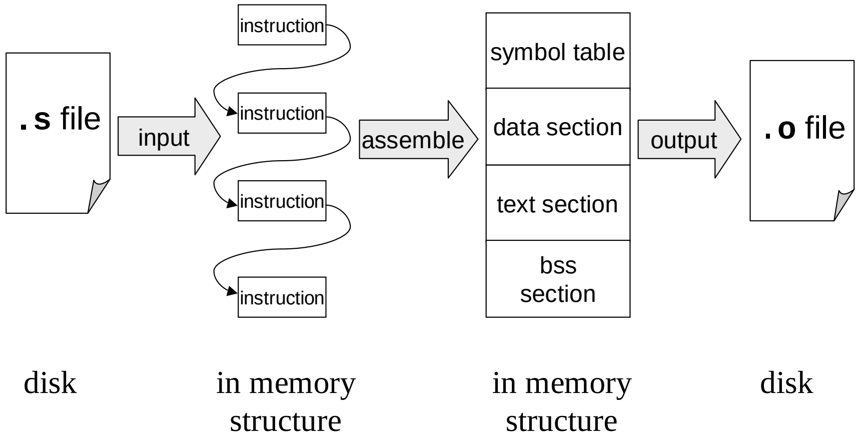
loop
done

def	loop
	0
disp22	done
	4
disp30	f
	12
disp22	loop
	20
def	done
	28

```
        .global loop
loop:   cmp %r16,%r24
        bge done
        nop
        call f
        nop
        ba loop
        inc %r16
done:
```

0:	. . .				
4:	00	0	≥	010	+24
8:	. . .				
12:	op	disp30: ?			
16:	. . .				
20:	00	0	always	010	-20
24:	. . .				
28:					

Implementing an Assembler



.s file

input

instruction

instruction

instruction

instruction

assemble

symbol table

data section

text section

bss section

output

.o file

disk

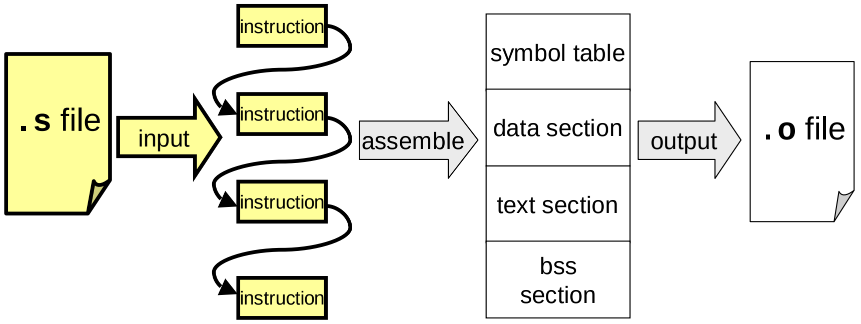
in memory structure

in memory structure

disk

Input Functions

- Read assembly language and produce list of instructions



.s file

input

instruction

instruction

instruction

instruction

assemble

symbol table

data section

text section

bss section

output

.o file

These functions are provided

Input Functions

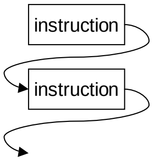
- Lexical analyzer
  - Group a stream of characters into tokens

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

- Syntactic analyzer
  - Check the syntax of the program

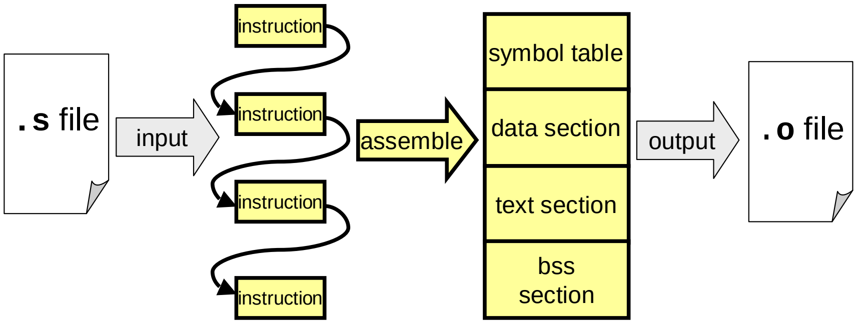
```
<MNEMONIC><REG><COMMA><REG><COMMA><REG>
```

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



Your Task in Assignment 5

- Implement two pass assembler
  - Process list of instructions to produce object file output structures



Packing fields using C

- Assembly language: addcc %r3, %r7, %r2
  -
- Format of arithmetic instructions:

`10`	`rd`	`op3`	`rs1`	`0`	`unused(0)`	`rs2`

31 29 24 18 13 12 4 0

rd = 2; op3 = 16; rs1 = 3; rs2 = 7;

w = (2<<29) | (rd<<24) | (op3<<18) | (0<<13) | (0<<4) | (rs2<<0) ;

`10`	`00010`	`010000`	`00011`	`0`	`000000000`	`00111`

31 29 24 18 13 12 4 0

In C language, you can also use the “bit field” feature.

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
```

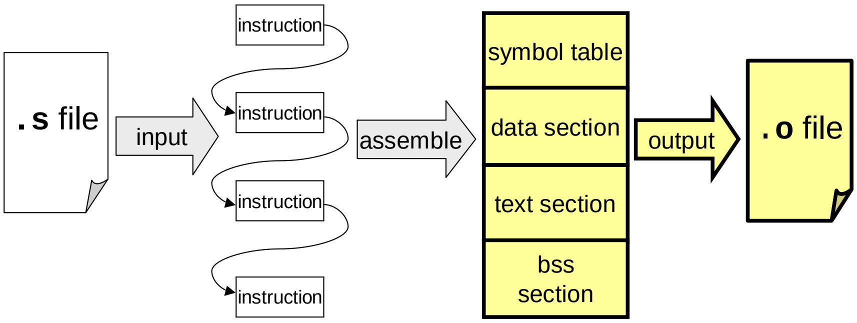
Output Data Structures (cont)

- For each section, produce…

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

Output Functions

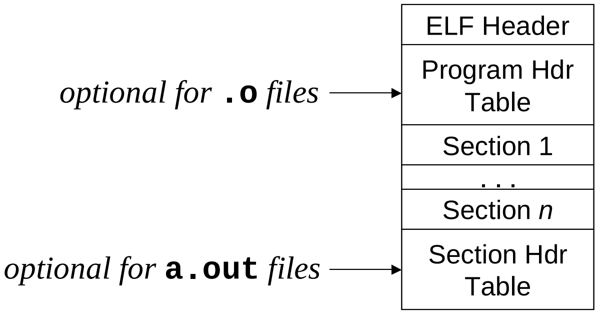
- Machine language output
  - Write symbol table and sections into object file (ELF file format)



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 Header

Program Hdr Table

Section 1

. . .

Section *n*

Section Hdr Table

optional for **`.o`** *files*

optional for **`a.out`** *files*

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

Data Section

```
           .section “.data”
           .ascii “COS”
           .align 2
           .byte 1
           .skip 7
var2:      .asciz “IS”
           .ascii “GR”
           .word 8

           .section “.text”
           add %r8, %r9, %r10
           add %r10, 22, %r10
           call addthree
           bg label1
           sethi %hi(var2), %r8
           or %r8, %lo(var2), %r8
           call printf
label1:    ld [%r5], %r7
           ld [%r5 + %r6], %r8
           ld [%r5 + 3], %r9
           ret

addthree:  add %r8, %r9, %r8
           retl
           add %r8, %r10, %r8
```

0	0x43	C .ascii "COS"
1	0x4f	O
2	0x53	S
3	0x00	.align 2
4	0x01	.byte 1
5	0x00	.skip 7
6	0x00	
7	0x00	
8	0x00	
9	0x00	
10	0x00	
11	0x00	
12	0x49	I .asciz "IS"
13	0x53	S
14	0x00	\0
15	0x47	G
16	0x52	R
17	0x00	.word 8
18	0x00	
19	0x00	
20	0x08	