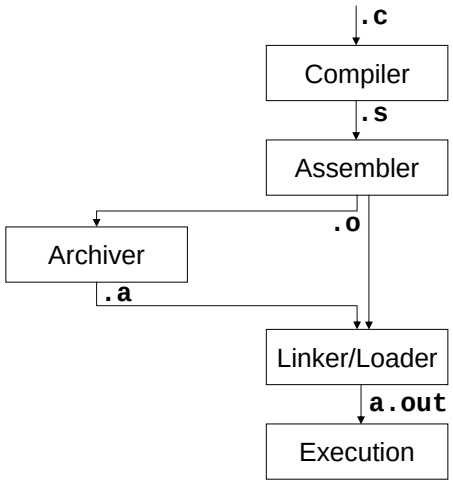
Assembler and Linker

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

Compilation Pipeline



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

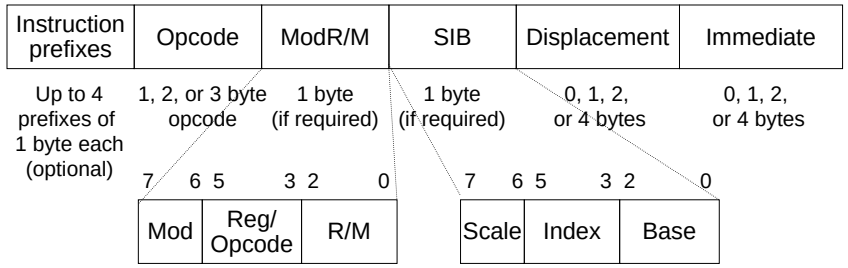
Translating to Machine Code

- Assembly language:

```
leal (%eax,%eax,4), %eax
```

- Machine code:
  - Byte 1: 8D (opcode LEA) — 1000 1101
  - Byte 2: 04 (Dest %eax, with SIB) — 0000 0100
  - Byte 3: 80 (base=%eax, index=%eax, scale=4) — 1000 0000

General IA32 Instruction Format



- Prefixes: lock, rep, repne/repnz, repe/repz, segment overwrite, operand-size overwrite, address-size overwrite
- Opcode
- ModR/M and SIB: most memory operands need these
- Displacement and immediate: depending on opcode, ModR/M and SIB
- IA32's byte order is little endian (left-hand side: smaller addresses)

Assembly Language Statements

- Imperative statements specify instructions
  - Typically map 1 imperative statement to 1 machine instruction
- Synthetic instructions
  - They are mapped to one or more machine instructions
- Declarative statements specify assembly time actions
  - Reserve space (.comm, .lcomm, …)
  - Define symbols (.globl Foo, …)
  - Identify segments (.text, .rodata, …)
  - 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: Symbol Manipulation

```
.text
    ...
    movl count, %eax
    ...
    .data
count:
    .word 0
    ...
```

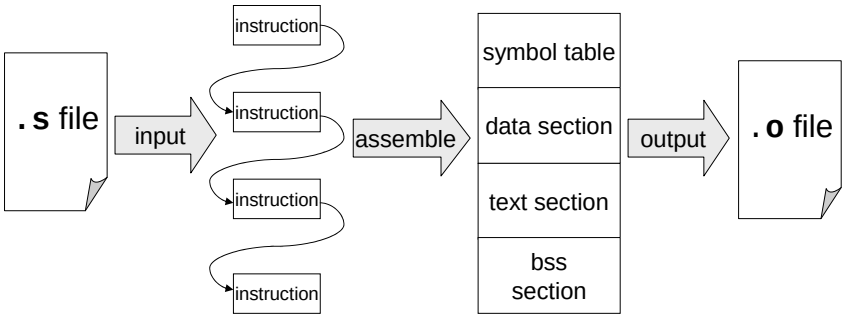
```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```

Create labels and remember their addresses
Deal with the "forward reference problem"

Dealing with Forward References

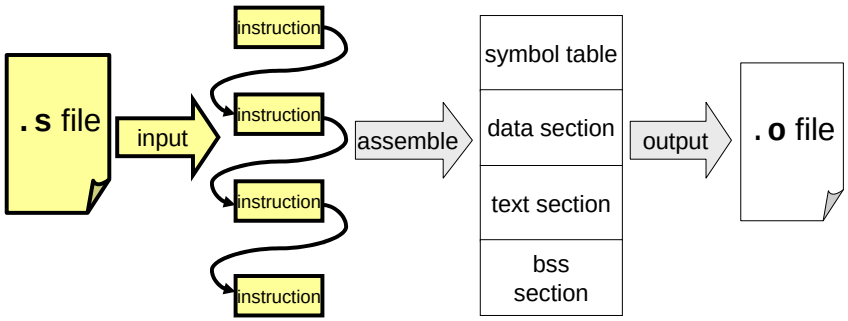
- Most assemblers have two passes
  - Pass 1: symbol definition
  - Pass 2: instruction assembly
- Or, alternatively,
  - Pass 1: instruction assembly
  - Pass 2: patch the cross-reference

Implementing an Assembler



Input Functions

- Read assembly language and produce list of instructions



Input Functions

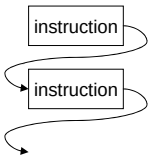
- Lexical analyzer
  - Group a stream of characters into tokens

```
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



Instruction Assembly

```
...
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```

D 0	3 9	[0]
disp?	7 D	[2]
	5 2	[4]
disp?	E 8	[5]
disp?	E 9	[10]
		[15]

Symbol Table

	type	label	address
loop	def	loop	0
done	disp_s	done	2
	disp_l	foo	5
	disp_l	loop	10
	def	done	15

```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```

D 0	3 9	[0]
disp_s	7 D	[2]
	5 2	[4]
disp_l	E 8	[5]
disp_l	E 9	[10]
		[15]

Symbol Table

	type	label	address
loop	def	loop	0
done	disp_s	done	2
	disp_l	foo	5
	disp_l	loop	10
	def	done	15

```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```

D 0	3 9	[0]
disp_s	7 D	[2]
	5 2	[4]
disp_l	E 8	[5]
disp_l	E 9	[10]
		[15]

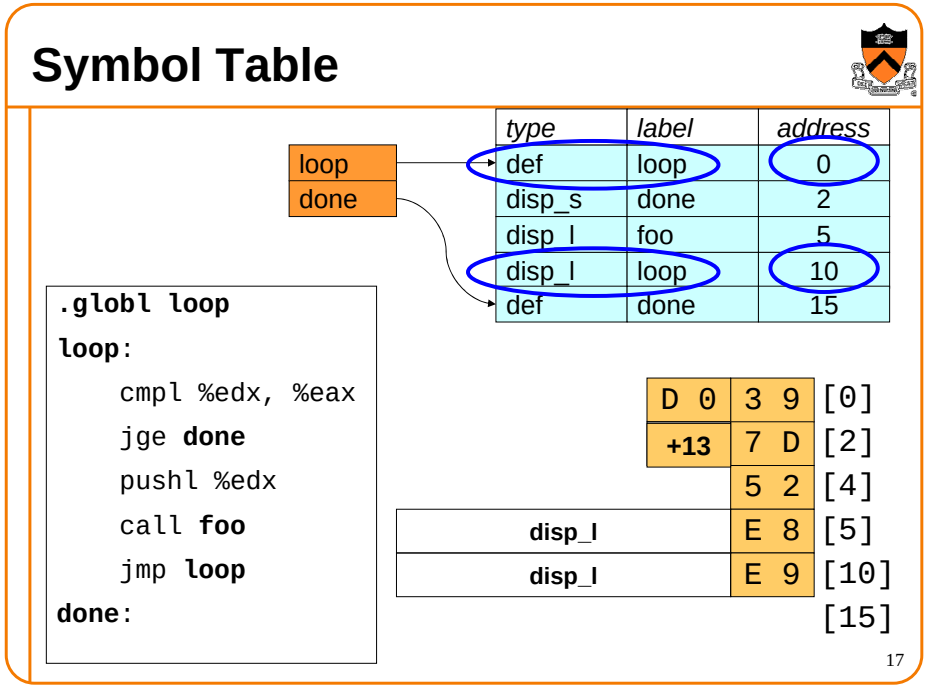
Symbol Table

	type	label	address
loop	def	loop	0
done	disp_s	done	2
	disp_l	foo	5
	disp_l	loop	10
	def	done	15

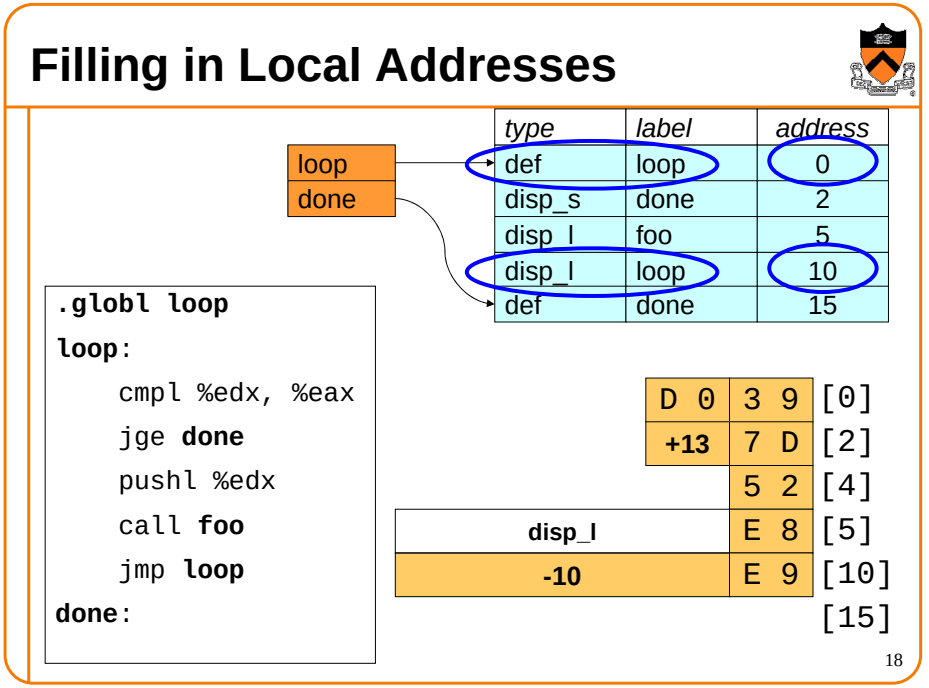
```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```

D 0	3 9	[0]
+13	7 D	[2]
	5 2	[4]
disp_l	E 8	[5]
disp_l	E 9	[10]
		[15]

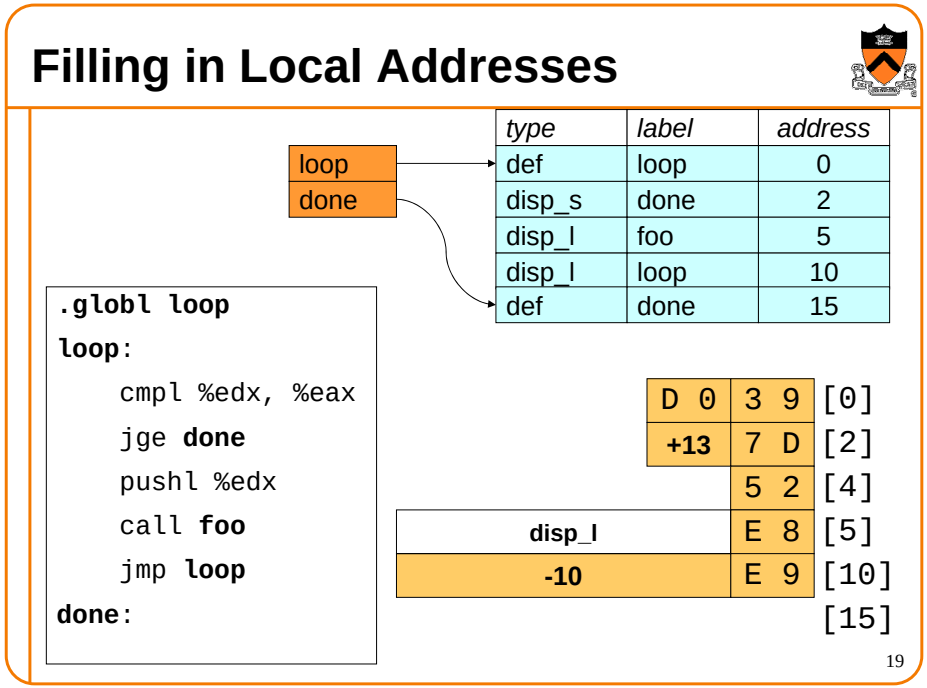
Symbol Table



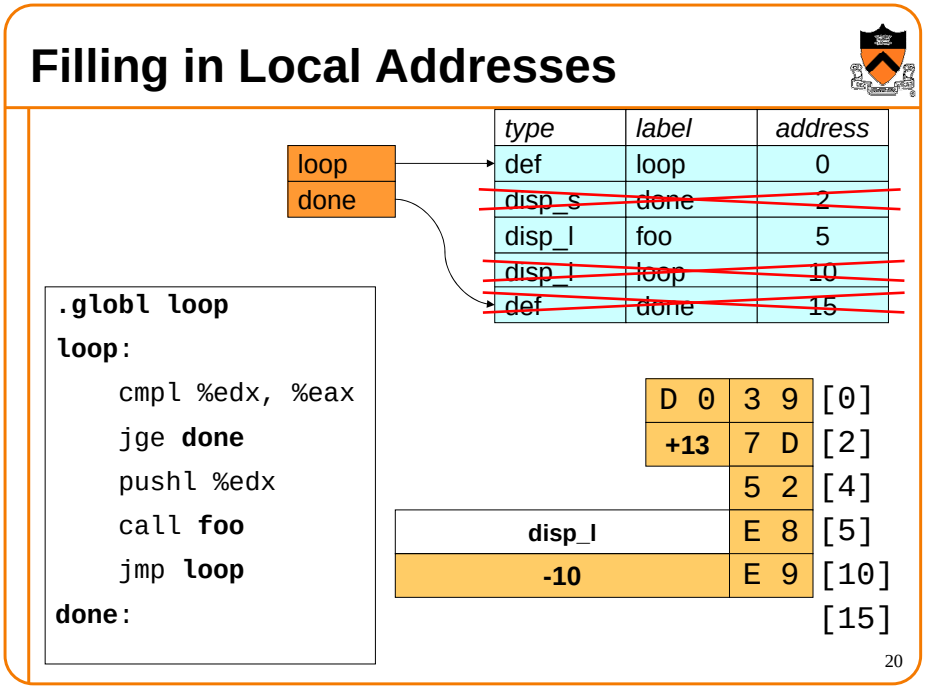
Filling in Local Addresses



Filling in Local Addresses



Filling in Local Addresses



Relocation Records

```
...
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```

def	loop	0
disp_l	foo	5

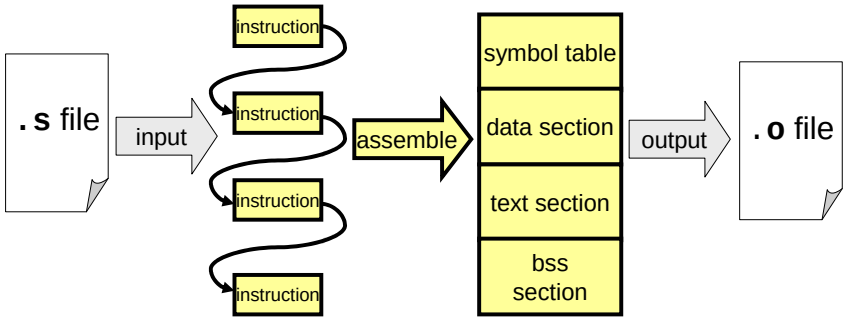
	D 0	3 9 [0]
	+13	7 D [2]
		5 2 [4]
disp_l		E 8 [5]
-10		E 9 [10]
		[15]

Assembler Directives

- Delineate segments
  - .section
- Allocate/initialize data and bss segments
  - .word .half .byte
  - .ascii .asciz
  - .align .skip
- Make symbols in text externally visible
  - .global

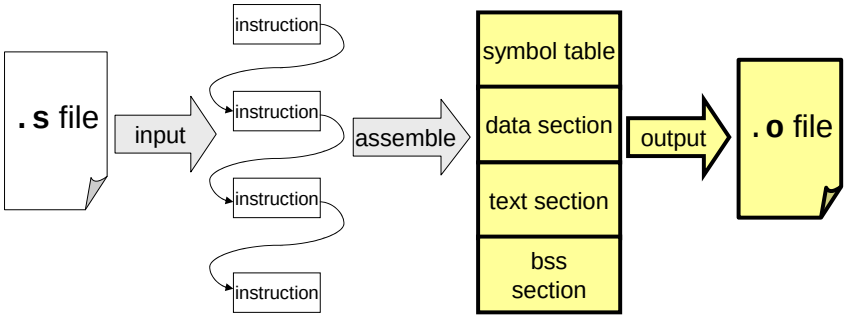
Assemble into Sections

- Process instructions and directives to produce object file output structures



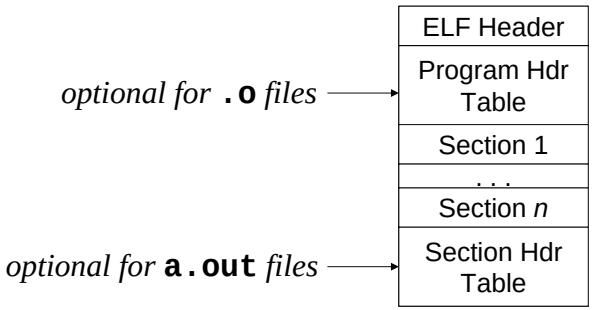
Output Functions

- Machine language output
  - Write symbol table and sections into object file



ELF: Executable and Linking Format

- Format of `.o` and `a.out` files
  - Output by the assembler
  - Input and output of linker



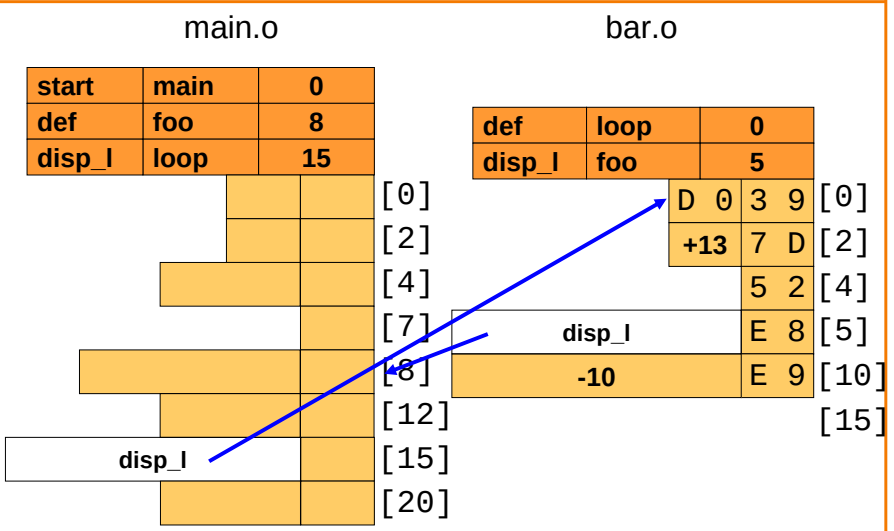
Invoking the Linker

- `ld bar.o main.o –l libc.a –o a.out`

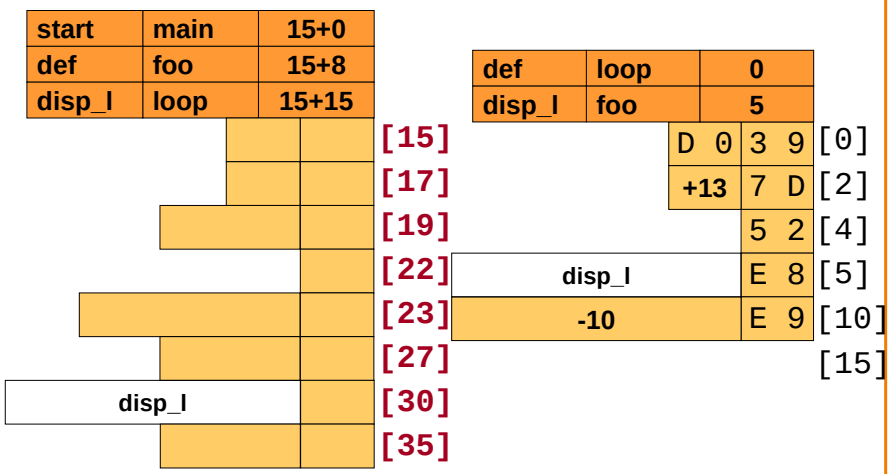
compiled program modules	library (contains more .o files)	output (also in ".o" format, but no undefined symbols)

- Invoked automatically by gcc,
- but you can call it directly if you like.

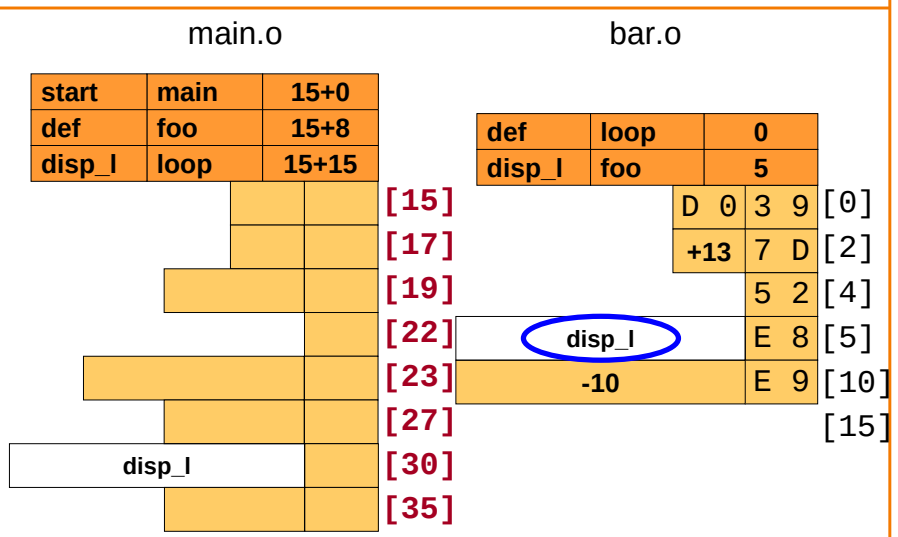
Multiple Object Files



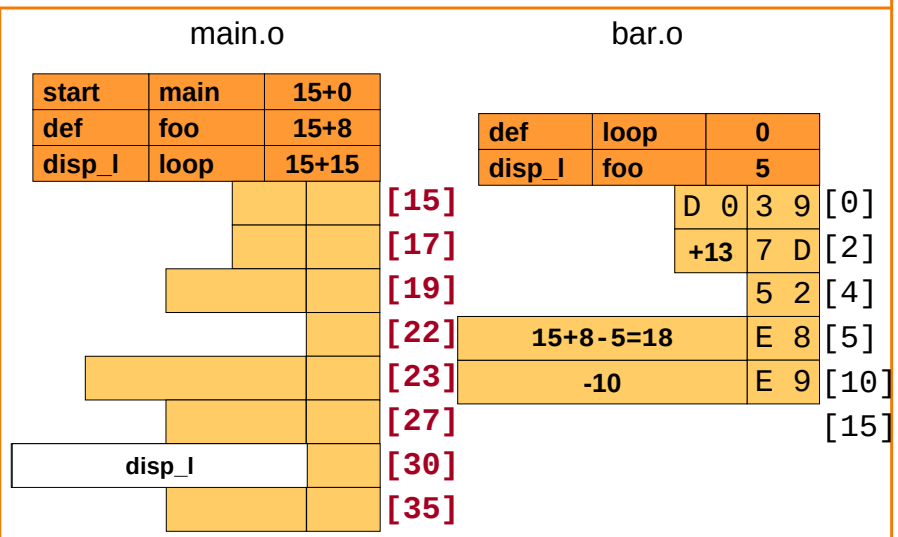
Step 1: Pick An Order



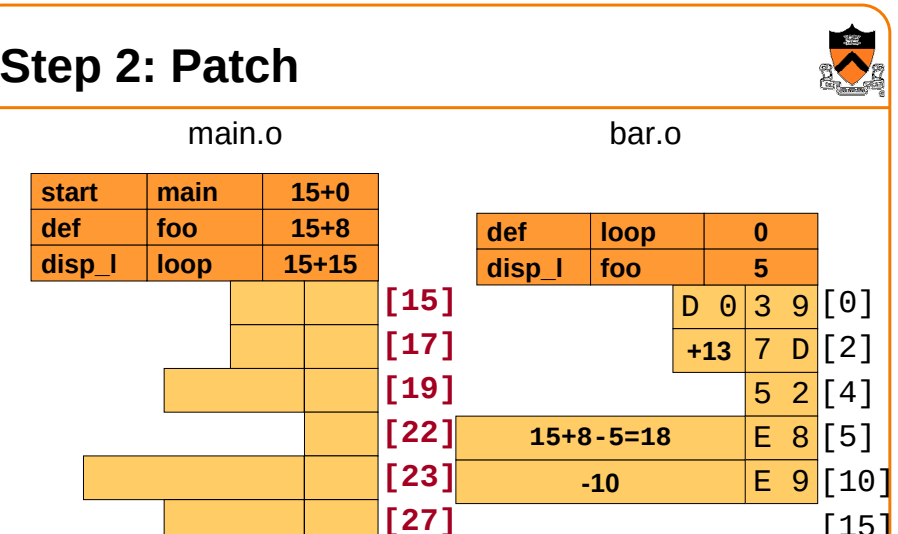
Step 1: Pick An Order



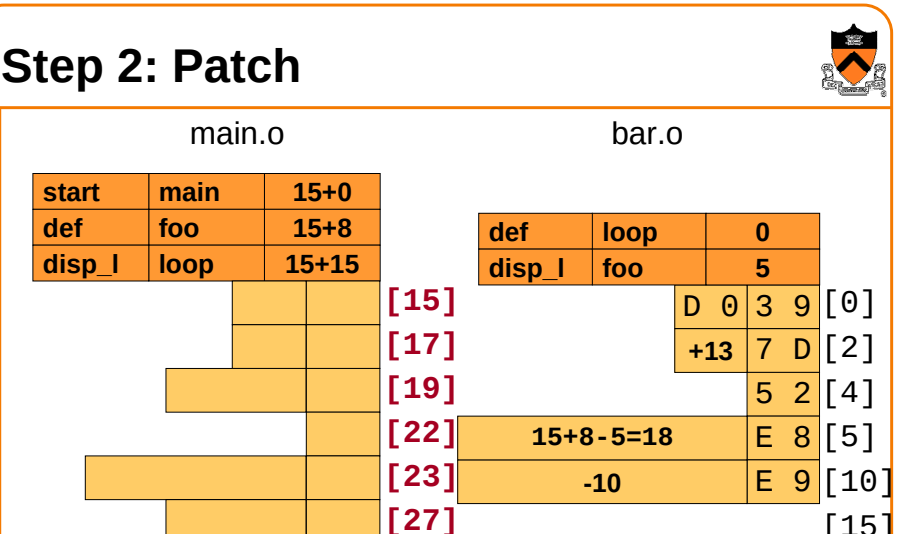
Step 2: Patch



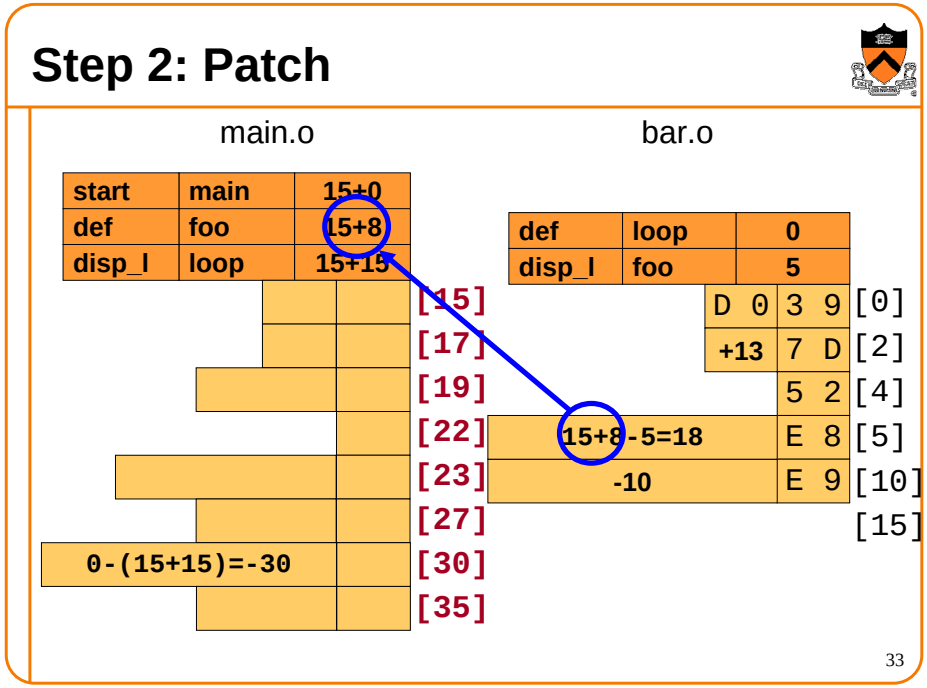
Step 2: Patch



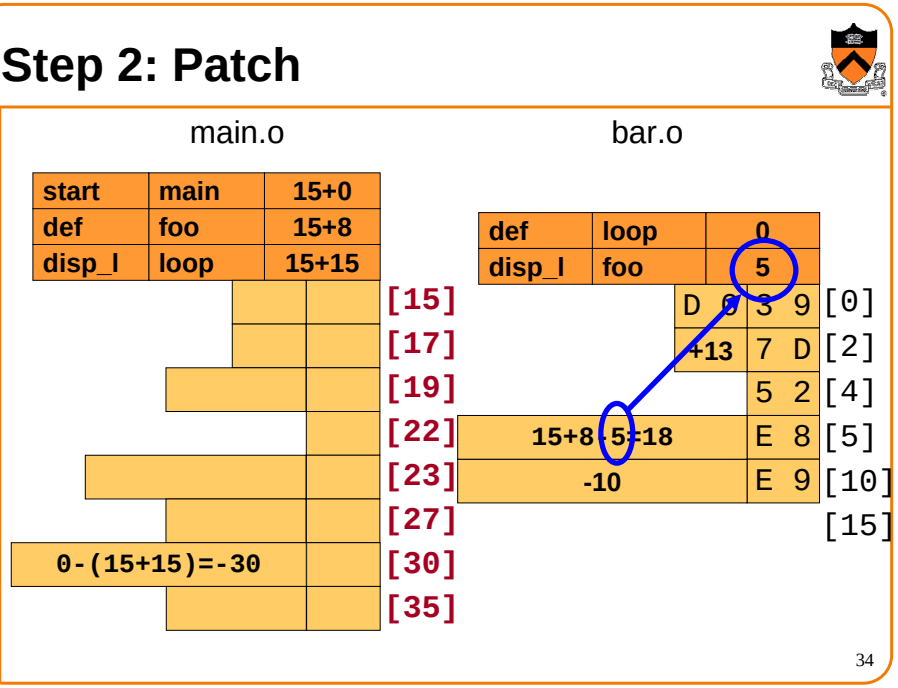
Step 2: Patch



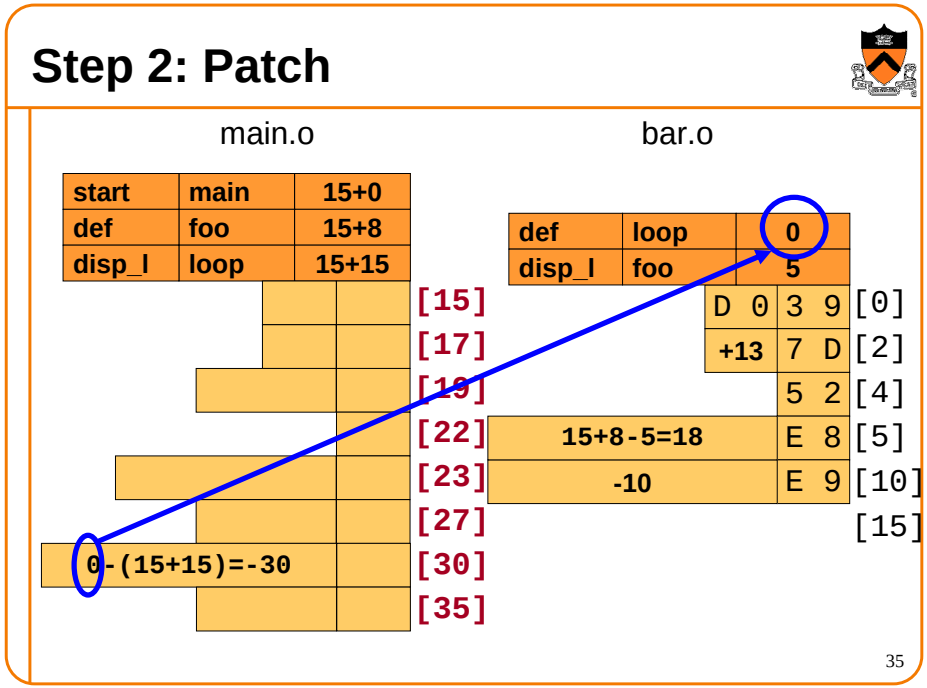
Step 2: Patch



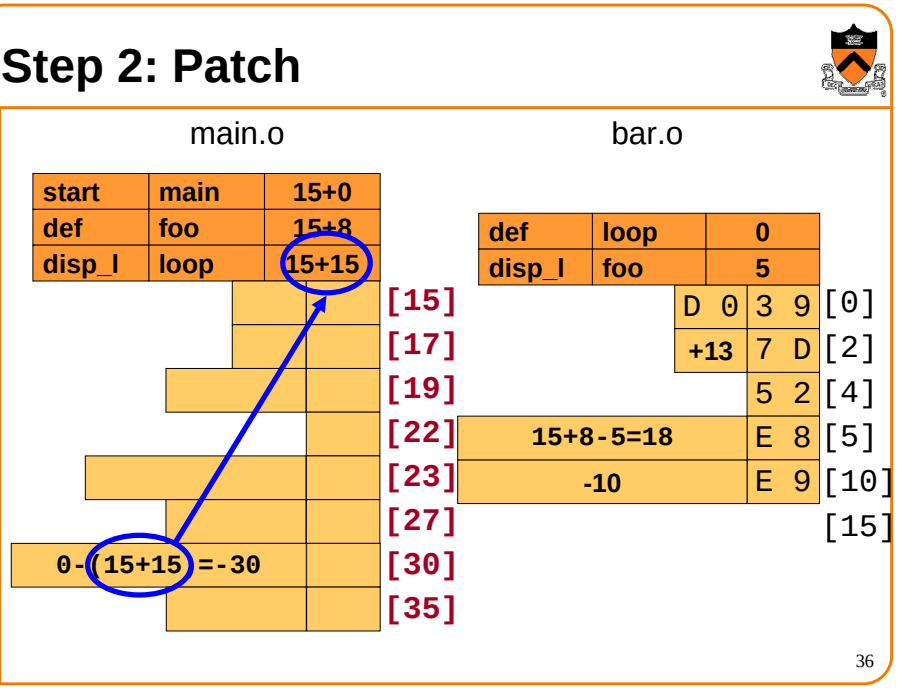
Step 2: Patch



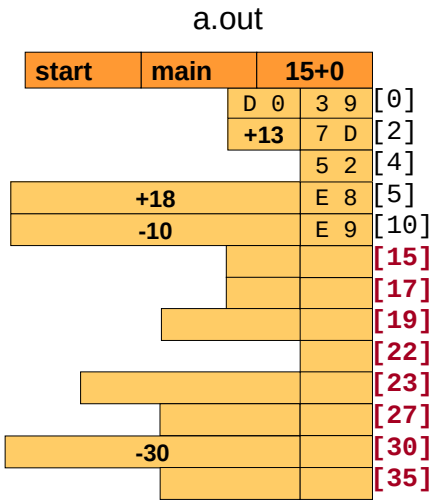
Step 2: Patch



Step 2: Patch



Step 3: Concatenate



Summary

- Assember
  - Read assembly language
  - Two-pass execution (resolve symbols)
  - Produce object file
- Linker
  - Relocation records
  - Order object codes
  - Patch and resolve displacements
  - Produce executable