Assemblers and Linkers

CS 217

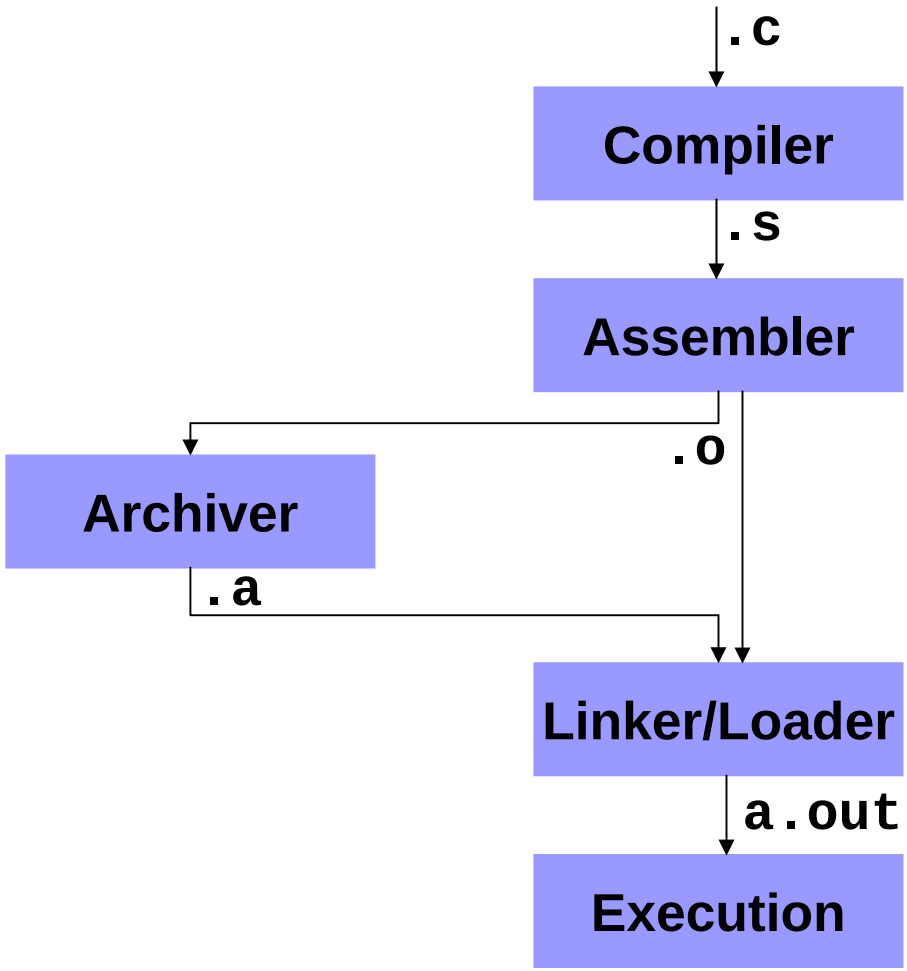
Goals of This Lecture

- Compilation process
  - Compile, assemble, archive, link, execute
- Assembling
  - Representing instructions
    - Prefix, opcode, addressing modes, operands
  - Translating labels into memory addresses
    - Symbol table, and filling in local addresses
  - Connecting symbolic references with definitions
    - Relocation records
  - Specifying the regions of memory
    - Generating sections (data, BSS, text, etc.)
- Linking
  - Concatenating object files
  - Patching references

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

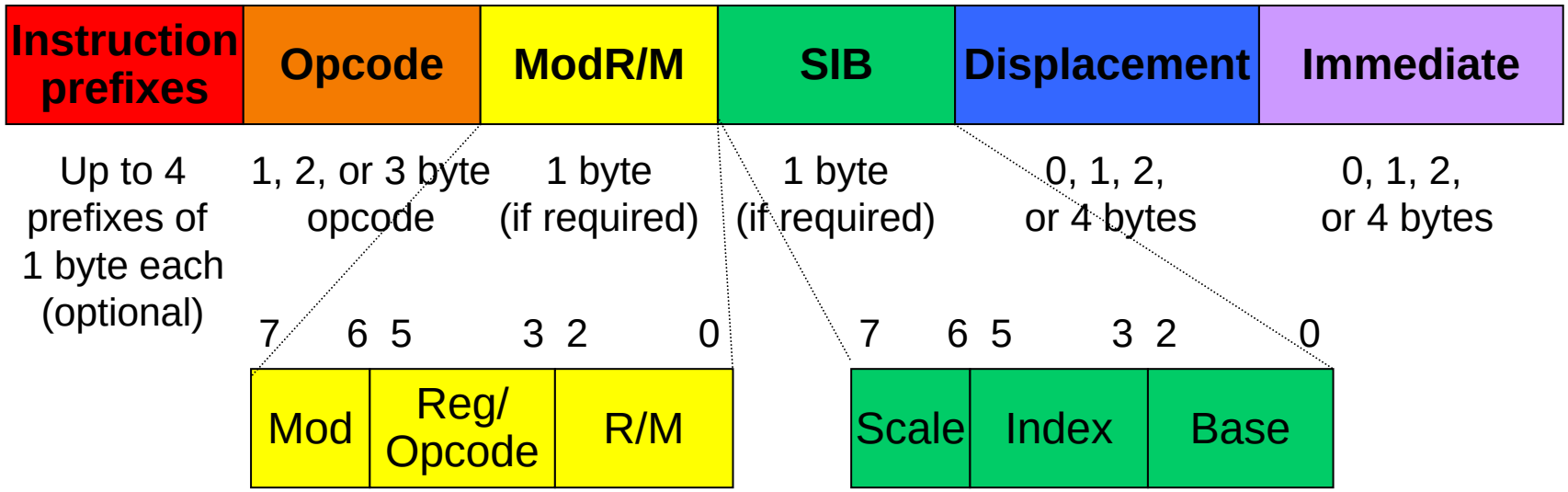
Compilation Pipeline



Assembler

- Purpose
  - Translates assembly language into machine language
    - Translate instruction mnemonics into op-codes
    - Translate symbolic names for memory locations
  - 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

General IA32 Instruction Format



- Prefixes: we won't worry about these for now
- Opcode
- ModR/M and SIB (scale-index-base): for memory operands
- Displacement and immediate: depending on opcode, ModR/M and SIB
- Note: byte order is little-endian (low-order byte of word at lower addresses)

Example: Push on to Stack

- Assembly language:

```
pushl %edx
```

- Machine code:
  - IA32 has a separate opcode for push for each register operand
    - 50: pushl %eax
    - 51: pushl %ecx
    - 52: pushl %edx → 0101 0010
    - …
  - Results in a one-byte instruction
- Observe
  - Sometimes one assembly language instruction can map to a *group* of different opcodes

Example: Load Effective Address

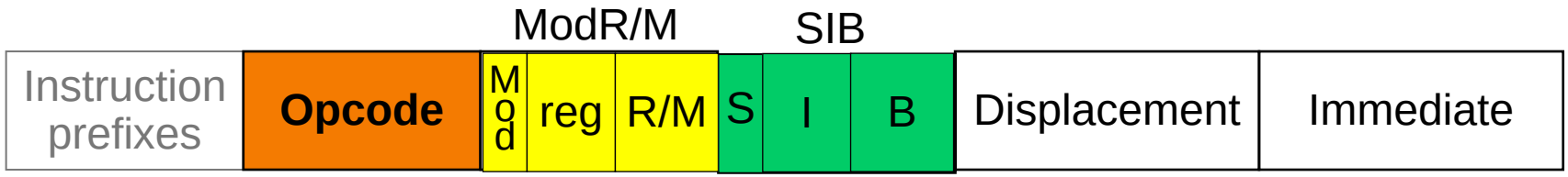
- Assembly language:

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

- Machine code:
  - Byte 1: 8D (opcode for “load effective address”) — 1000 1101
  - Byte 2: 04 (dest %eax, with scale-index-base) — 0000 0100
  - Byte 3: 80 (scale=4, index=%eax, base=%eax) — 1000 0000

Load the address %eax + 4 * %eax into register %eax

Example: Movl (Opcode 44)



movl %ecx, %ebx

4 4 2 3 3

1000 1000 11 011 001

ebx ecx

mov r/m32,r32

mode %_, %_

movl 12(%ecx), %ebx

4 4 2 3 3 8

1000 1000 01 011 001 00001100

ebx (ecx) 12

mode disp8(%_), %_

Mod=11 table

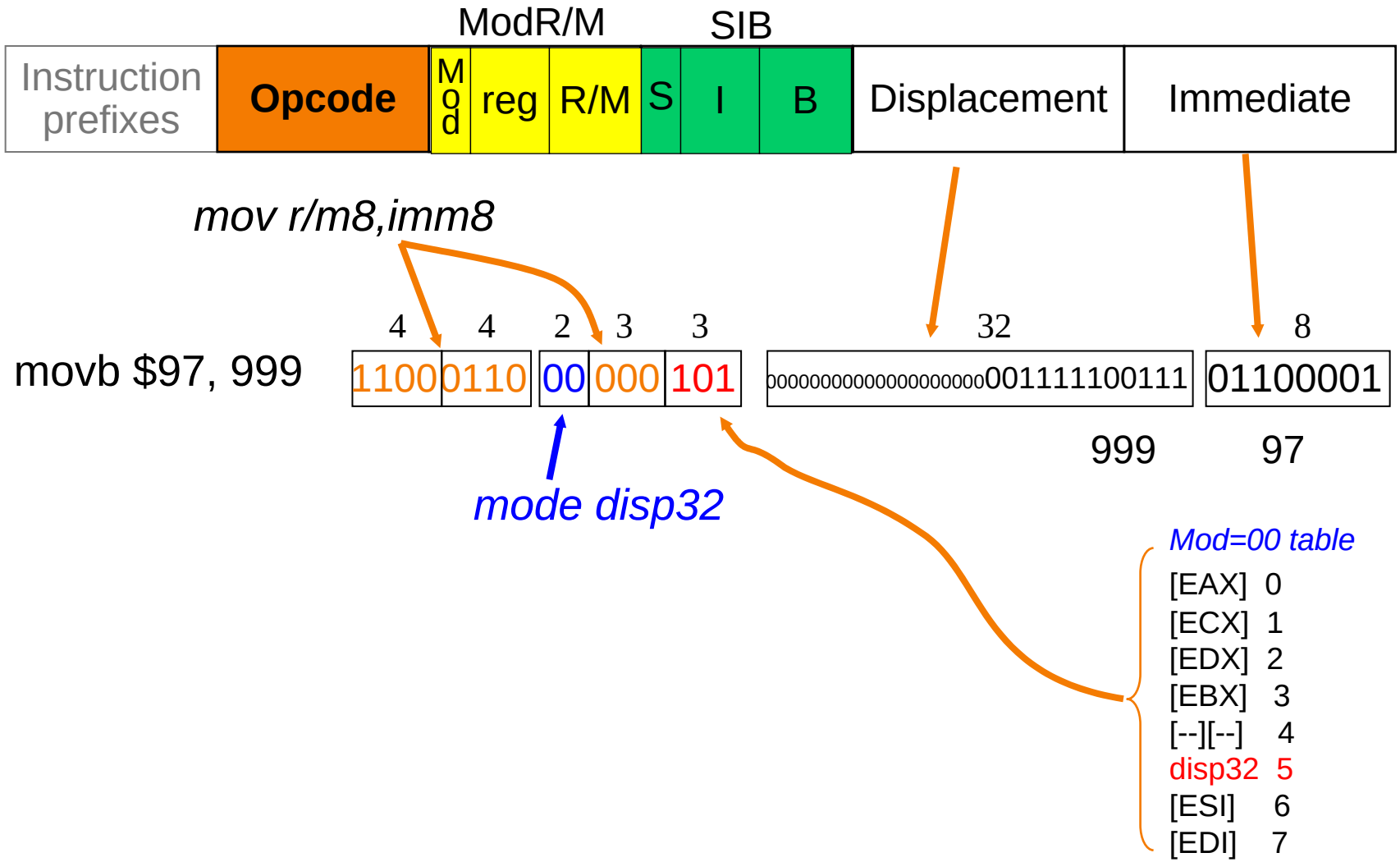
EAX 0
ECX 1
EDX 2
EBX 3
ESP 4
EBP 5
ESI 6
EDI 7

Mod=01 table

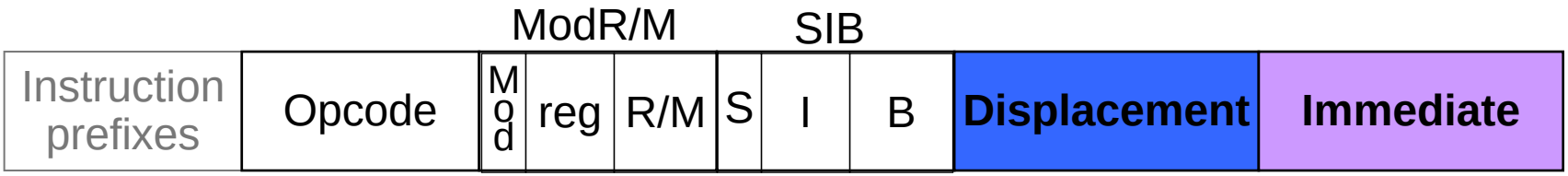
[EAX]+disp8 0
[ECX]+disp8 1
[EDX]+disp8 2
[EBX]+disp8 3
[--][--]+disp8 4
[EBP]+disp8 5
[ESI]+disp8 6
[EDI]+disp8 7

Reference: *IA-32 Intel Architecture Software Developer's Manual, volume 2*, page 2-1, page 2-6, and page 3-441

Example: Mov Immediate to Memory



Encoding as Byte String



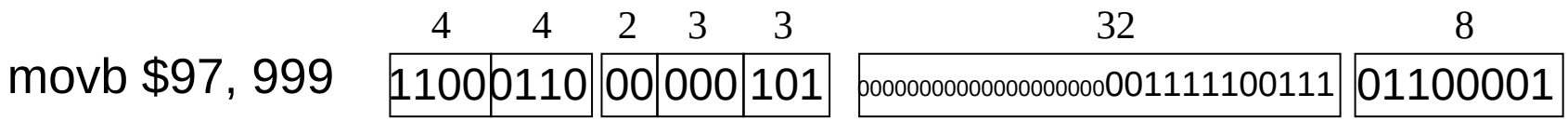
movb $97, 999

4	4	2	3	3	32	8
1100	0110	00	000	101	00000000000000000001111100111	01100001

C6 05 E7 03 00 00 61

little-endian

Assembly Language



C6 05 E7 03 00 00 61

```
.globl grade
       .data
grade:
       .byte  67
       .text
         :
         .
       movb   $'a', grade
         :
         :
```

located at address 999

Symbol Manipulation

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

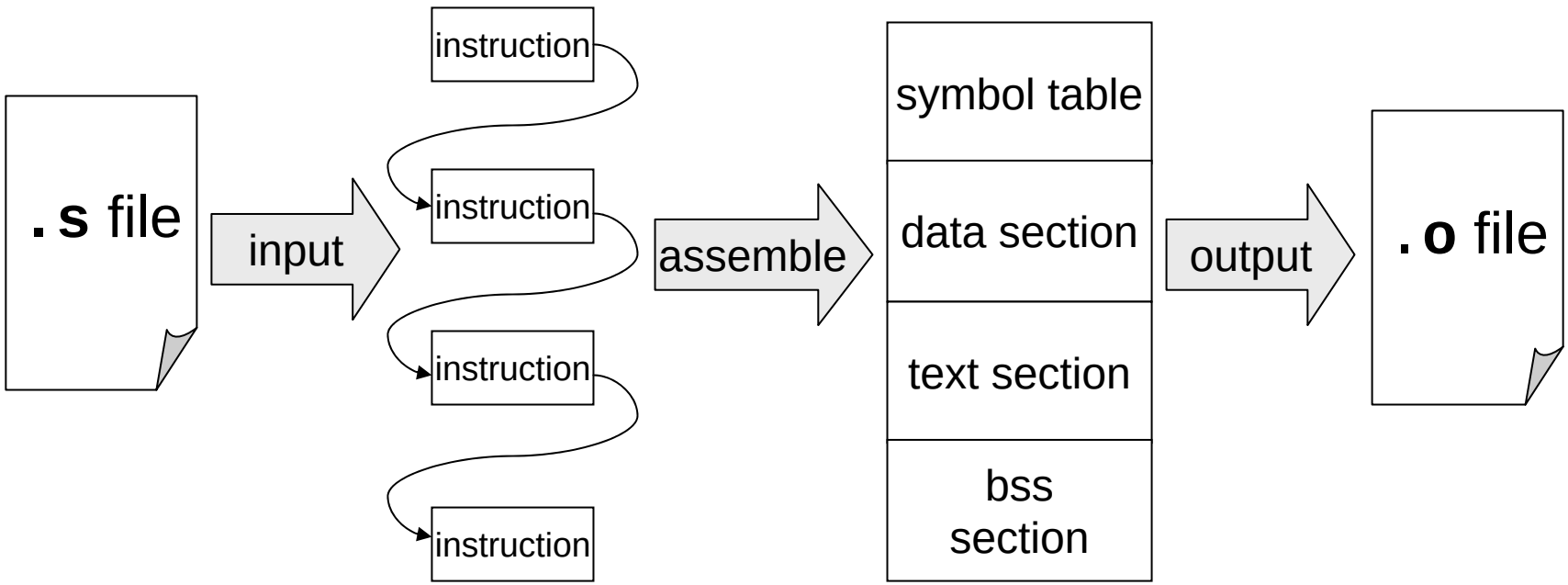
```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```

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

Dealing with Forward References

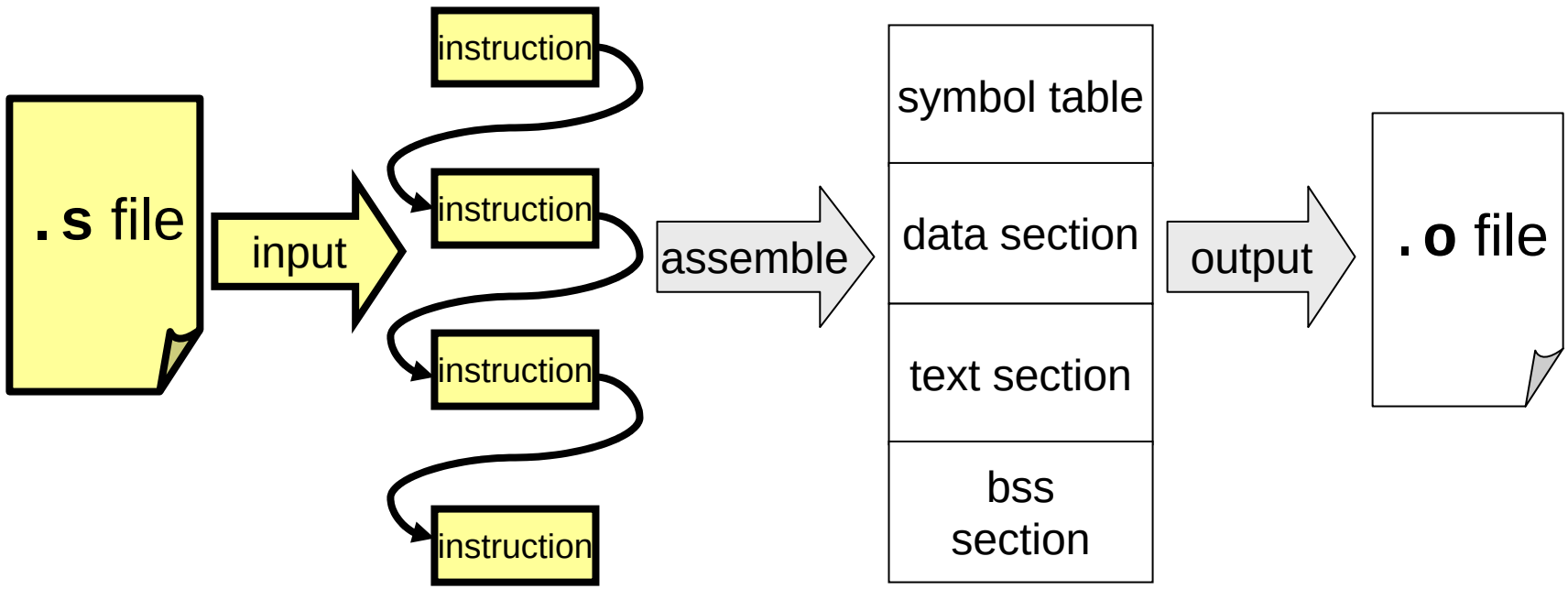
- Most assemblers have two passes
  - Pass 1: symbol definition
  - Pass 2: instruction assembly
- Or, alternatively,
  - Pass 1: instruction assembly
  - Pass 2: patch the cross-reference

Implementing an Assembler



Input Functions

- Read assembly language and produce list of instructions



Input Functions

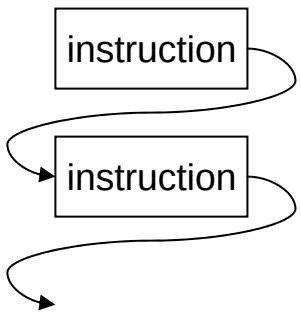
- Lexical analyzer
  - Group a stream of characters into tokens

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

- Syntactic analyzer
  - Check the syntax of the program

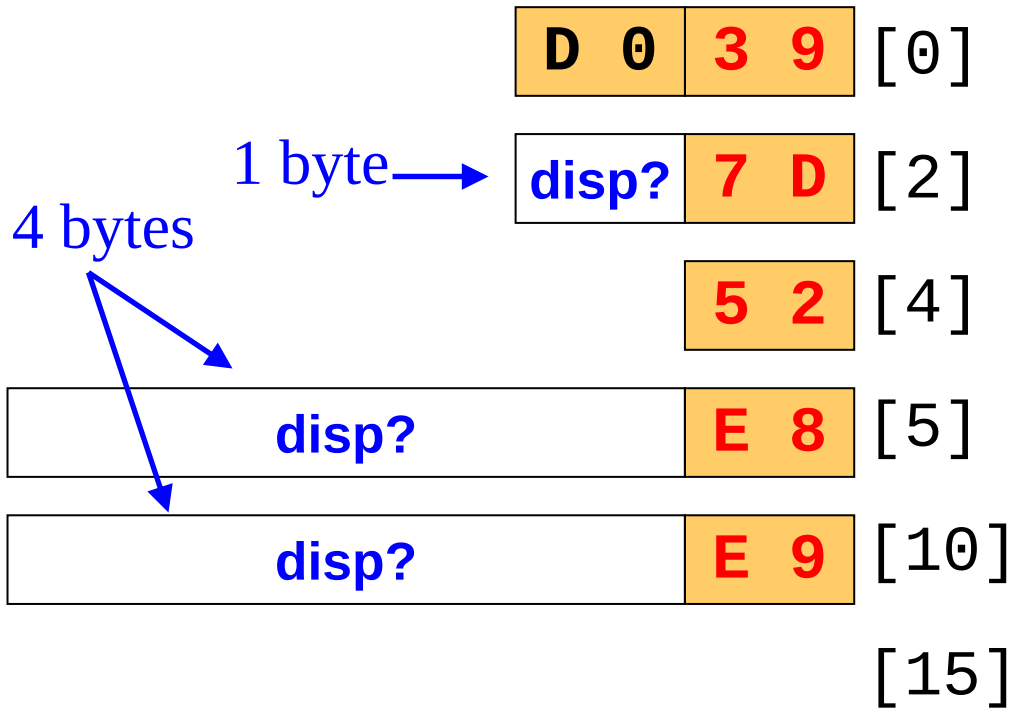
```
<MNEMONIC><REG><COMMA><REG><COMMA><REG>
```

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



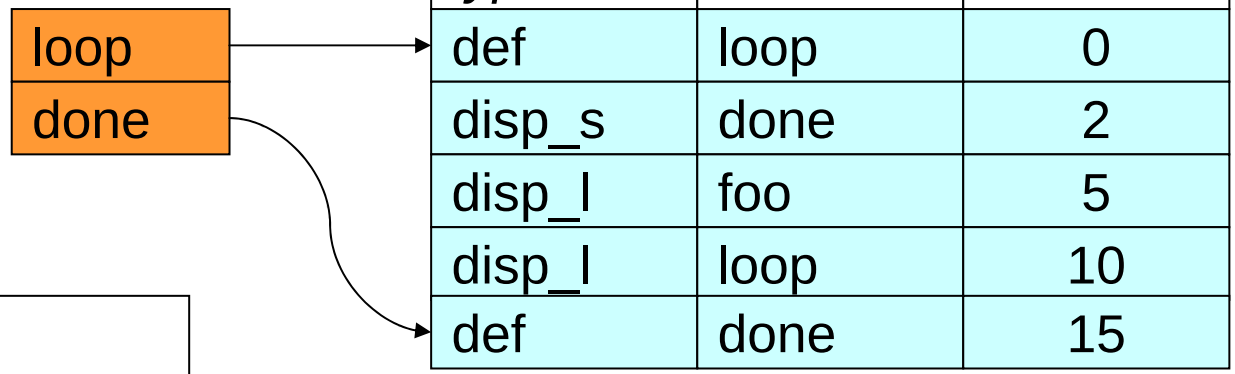
Instruction Assembly

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

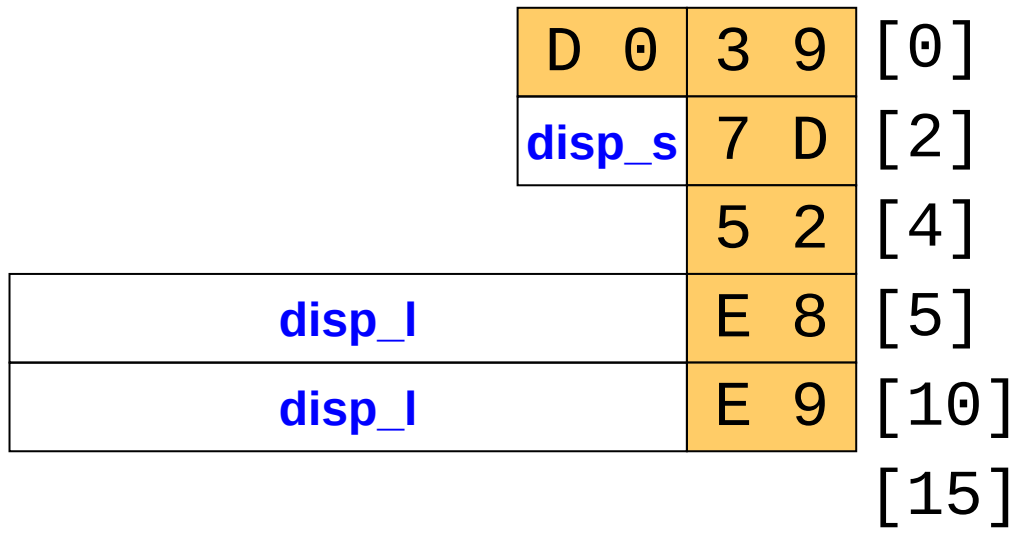


How to compute the address displacements?

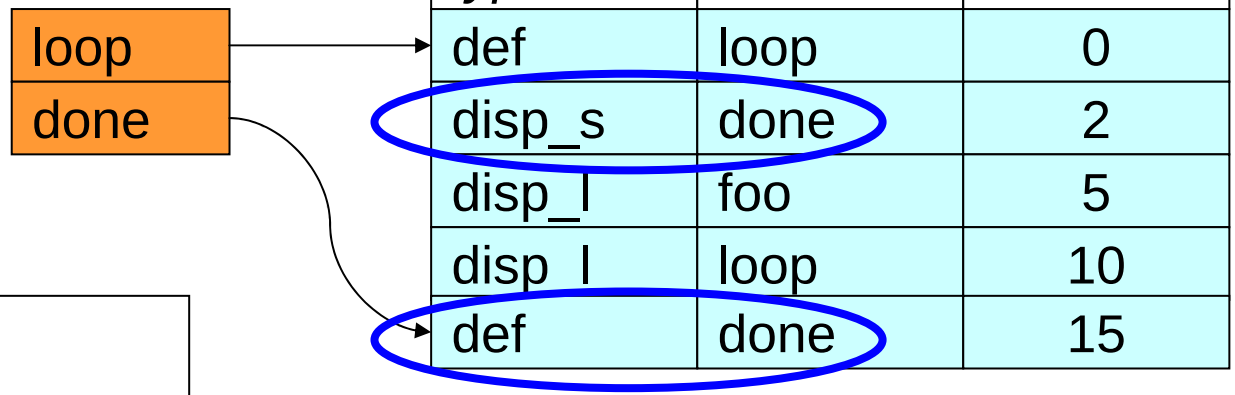
Symbol Table



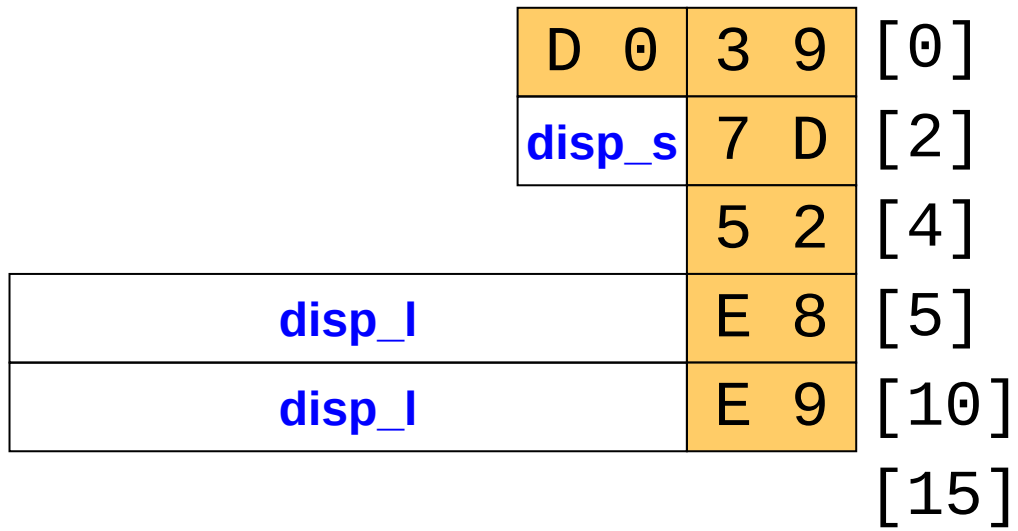

```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```



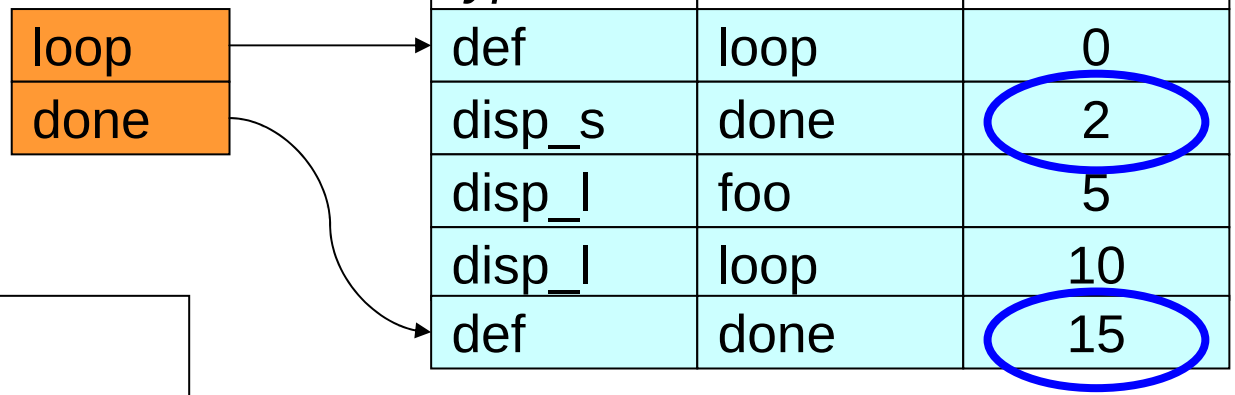
Symbol Table



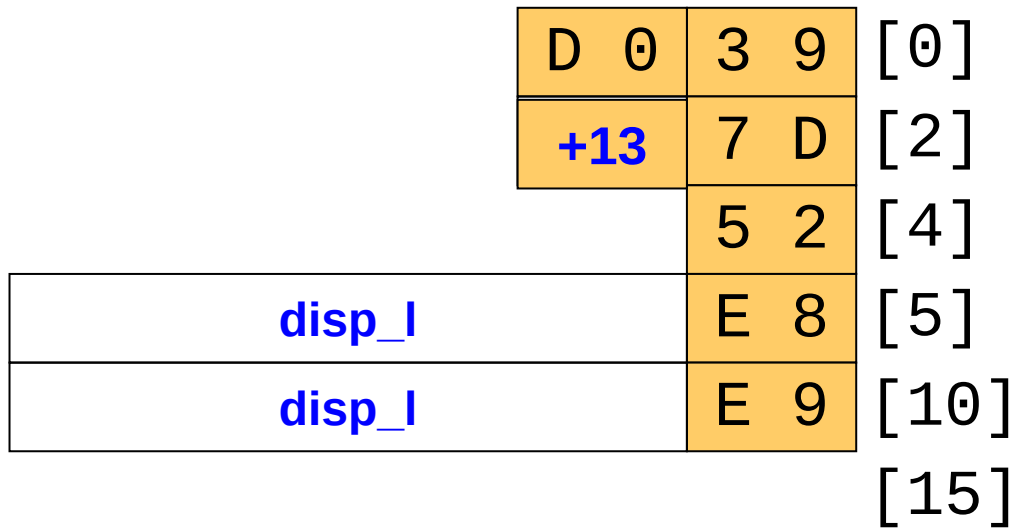

```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```



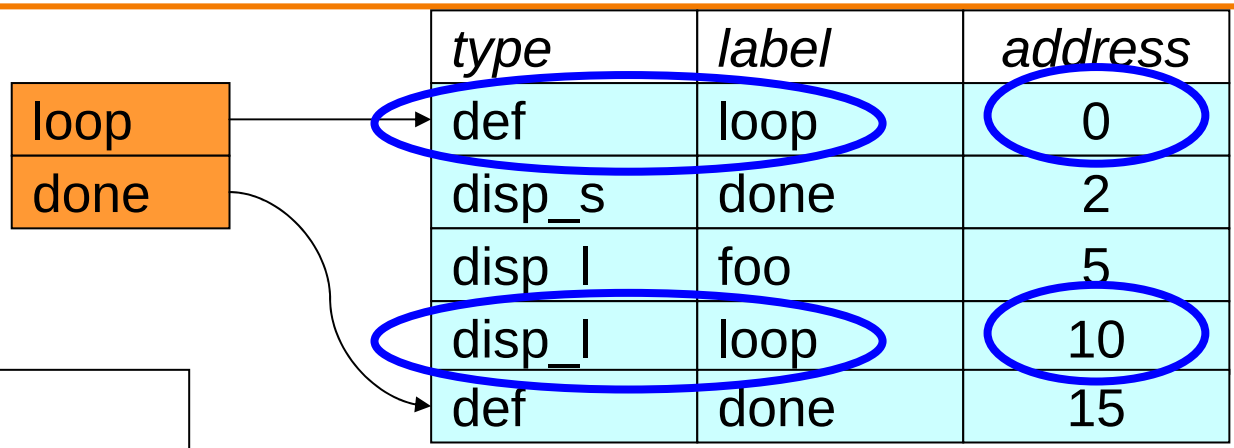
Symbol Table



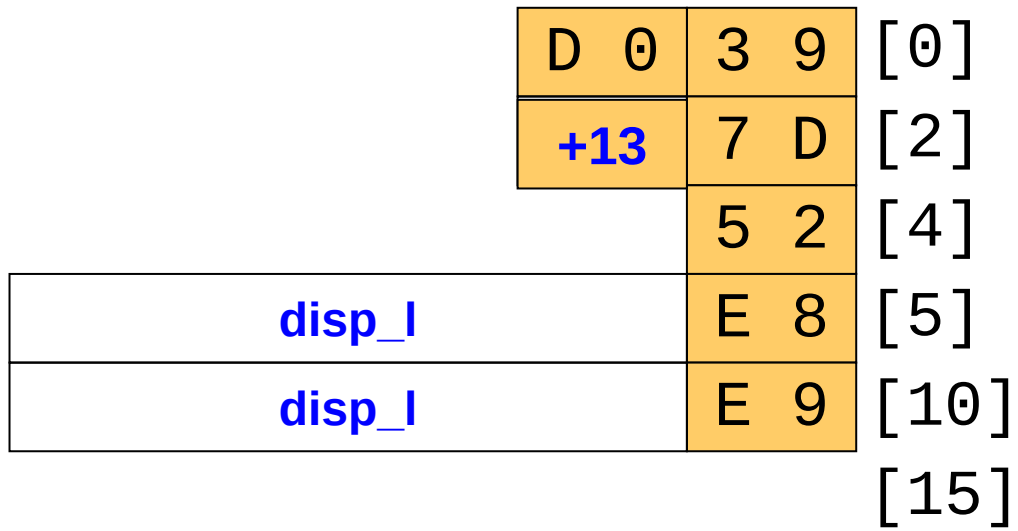

```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```



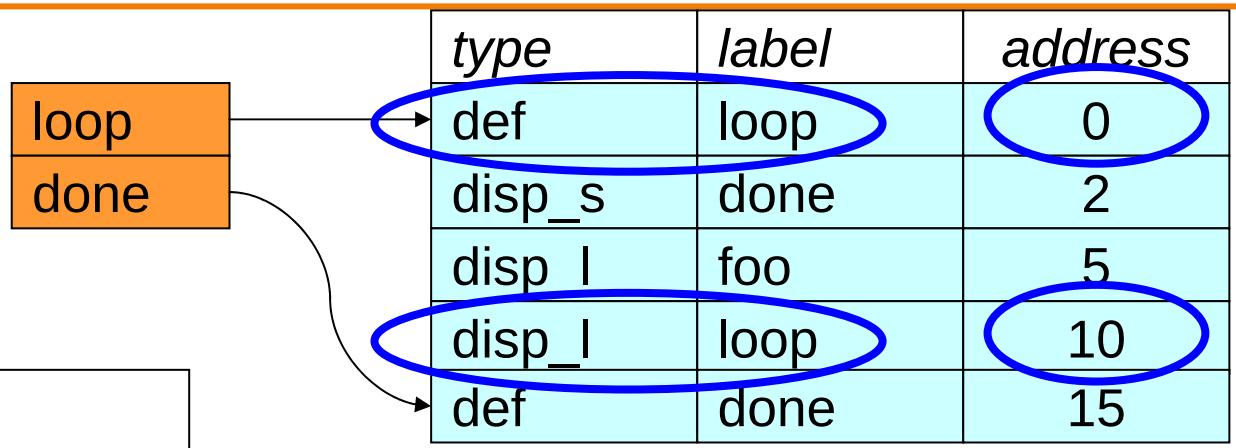
Symbol Table



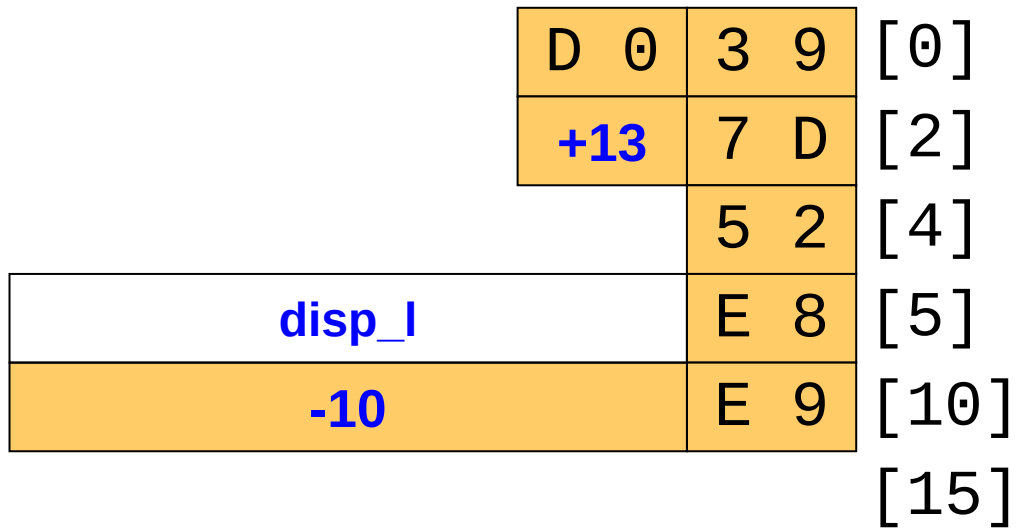

```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```



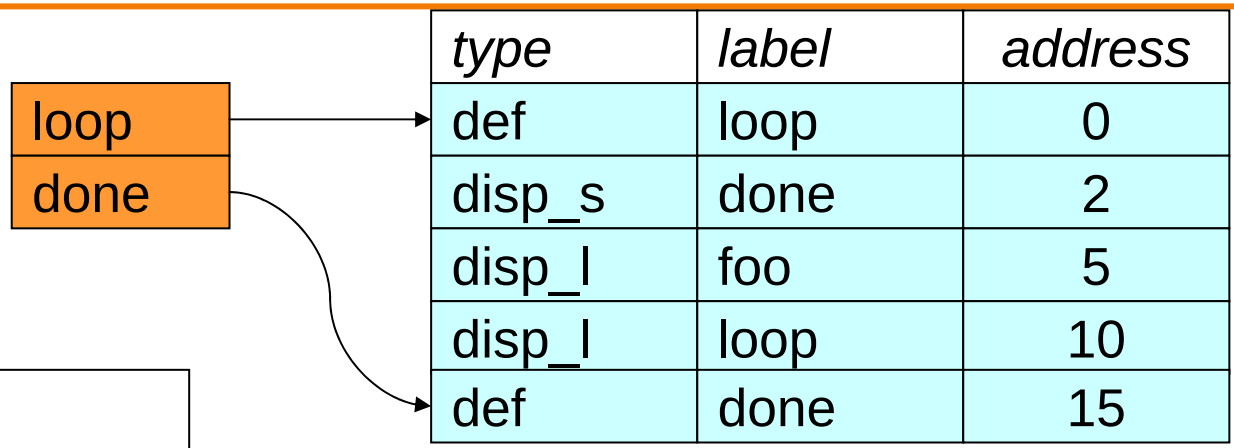
Filling in Local Addresses



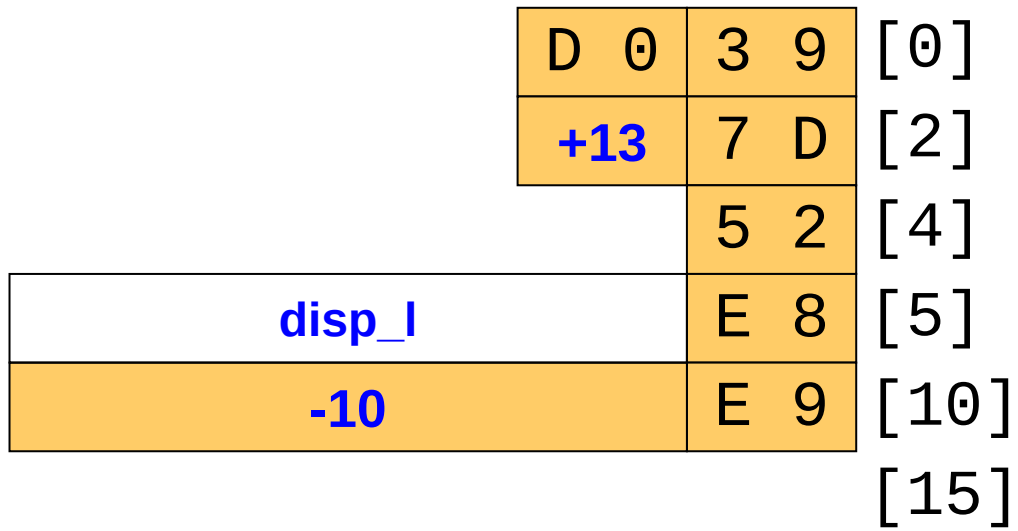

```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```



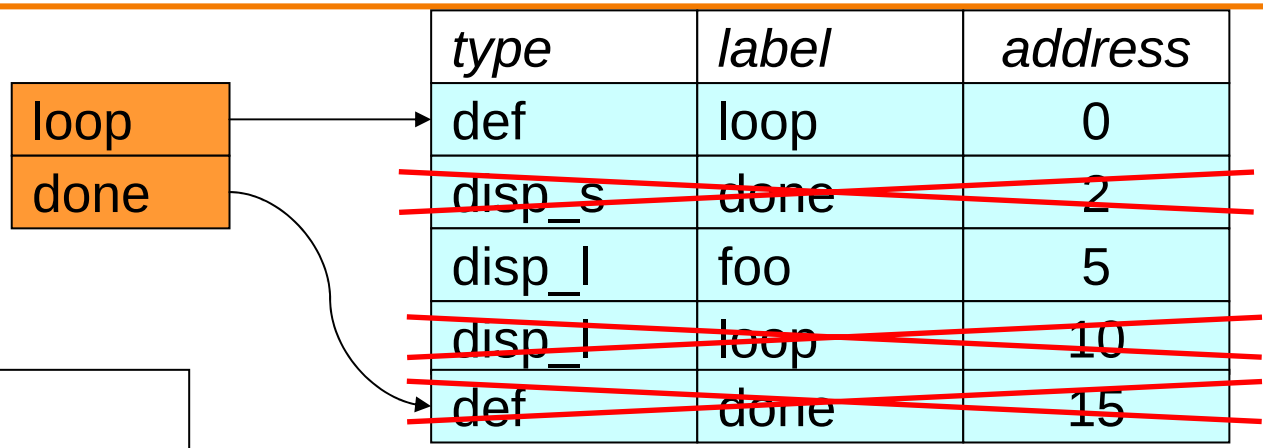
Filling in Local Addresses



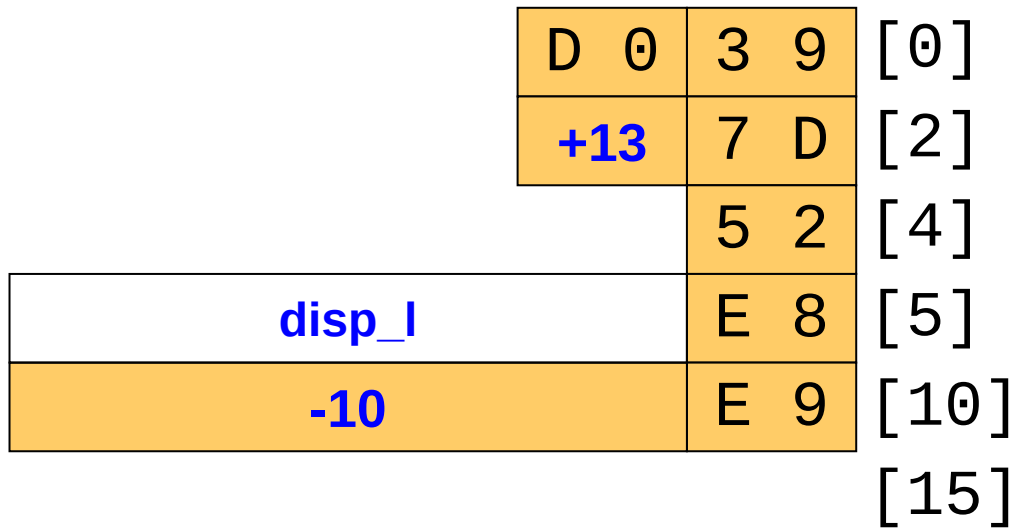

```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```



Filling in Local Addresses

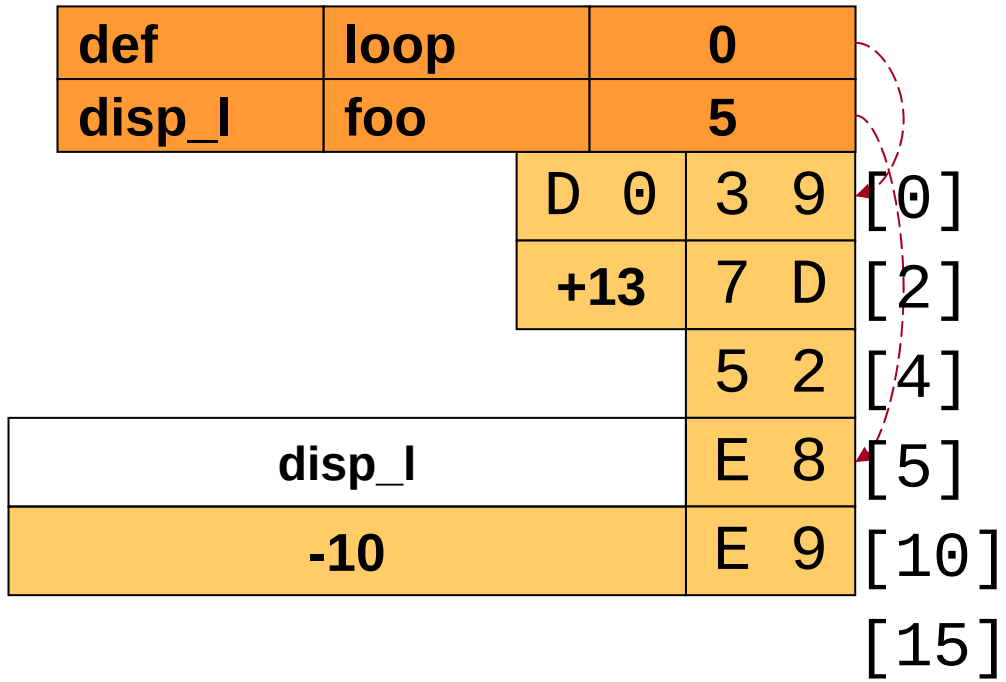



```
.globl loop
loop:
    cmpl %edx, %eax
    jge done
    pushl %edx
    call foo
    jmp loop
done:
```



Relocation Records

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

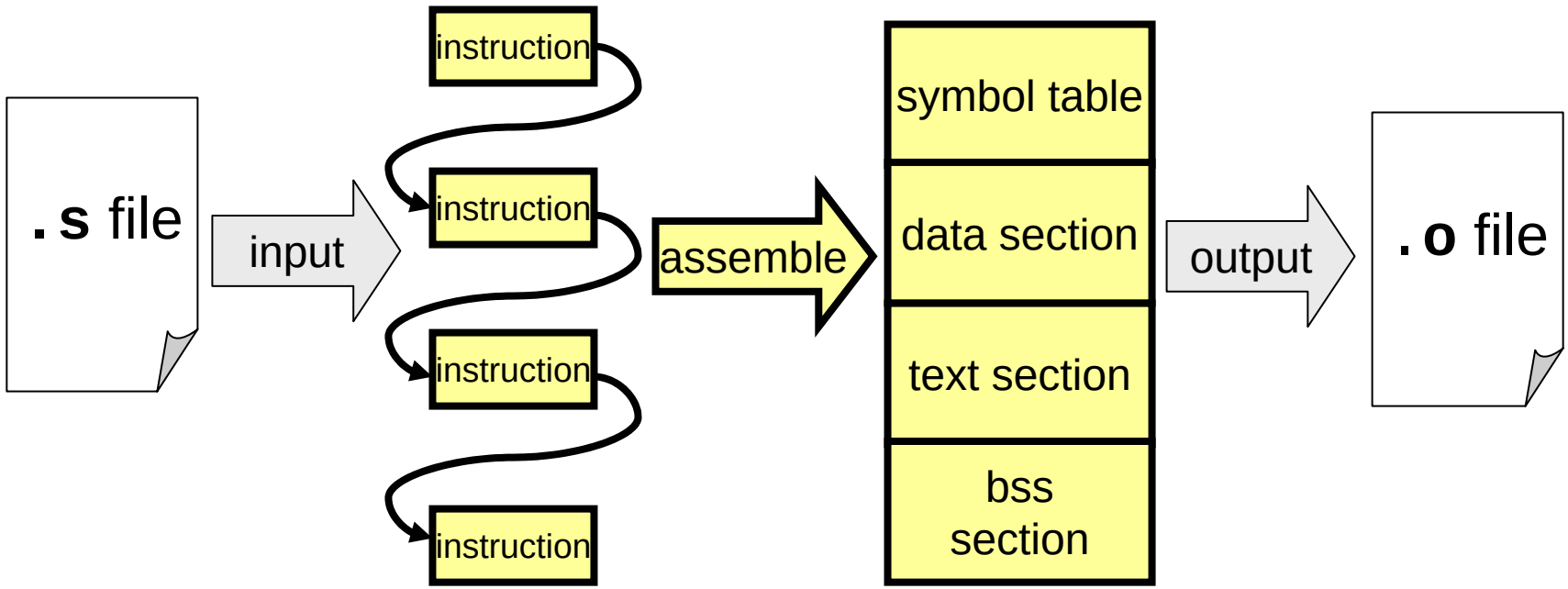


Assembler Directives

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

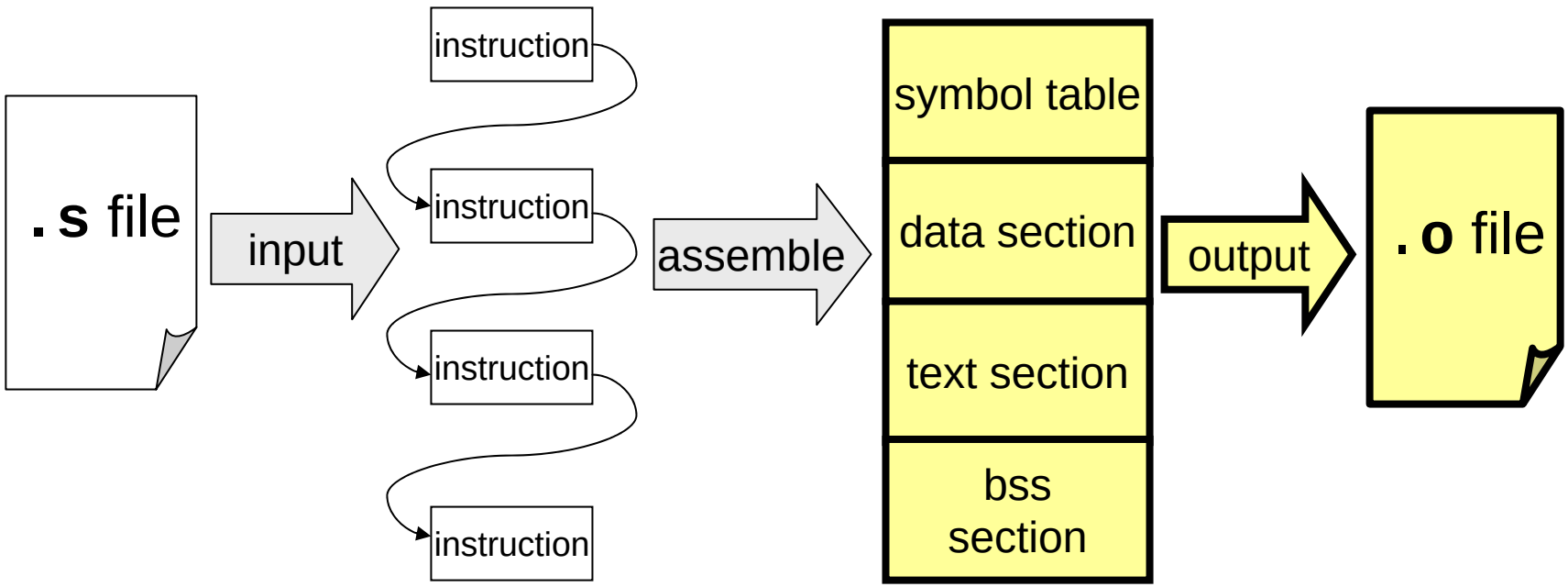
Assemble into Sections

- Process instructions and directives to produce object file output structures



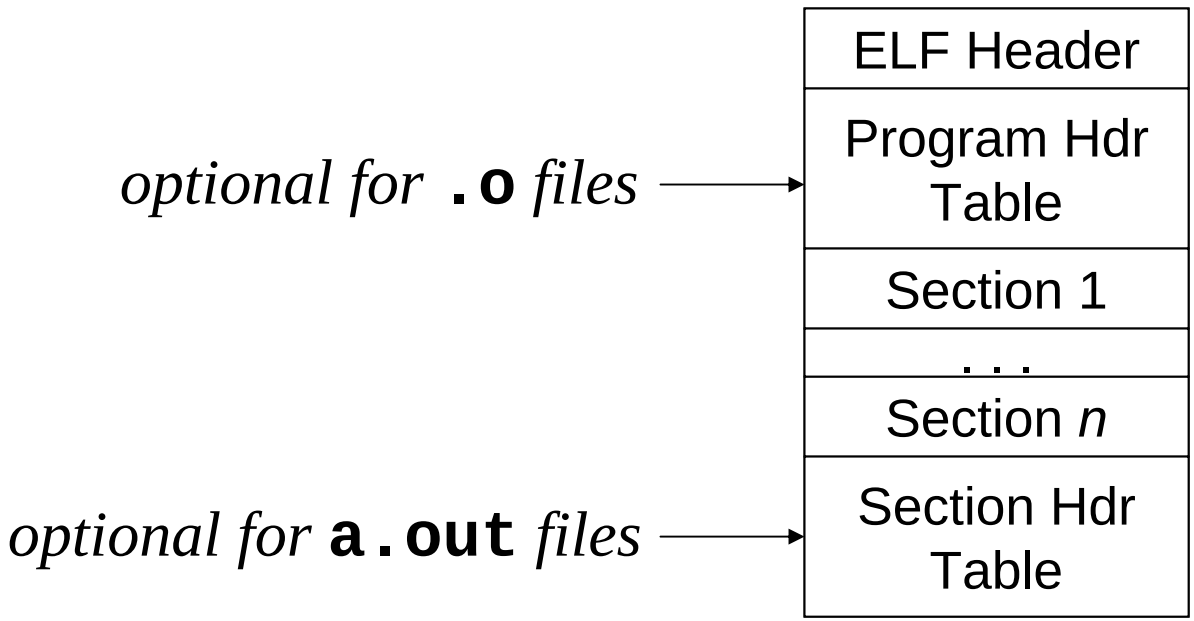
Output Functions

- 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`

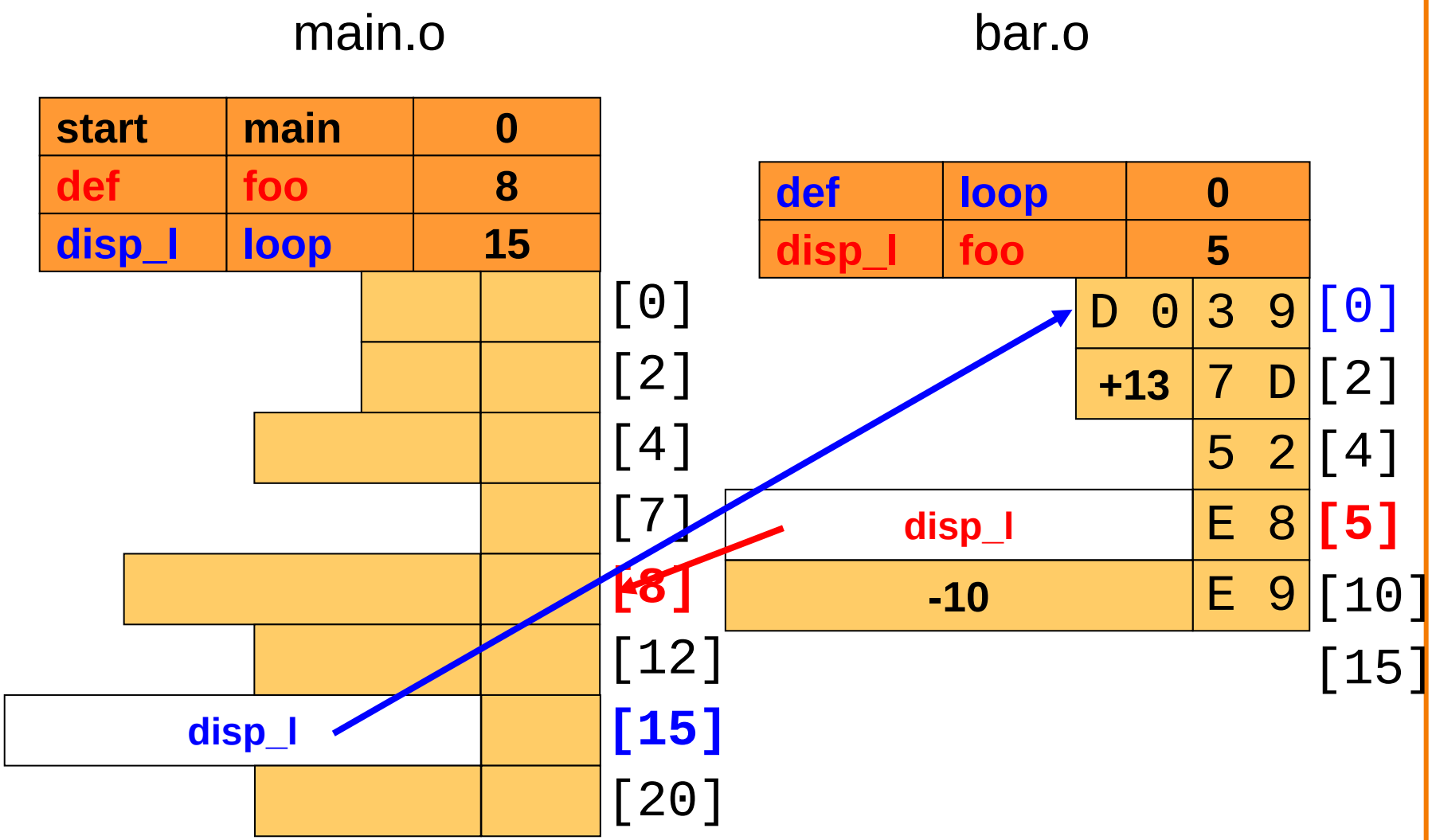
compiled program modules

library (contains more .o files)

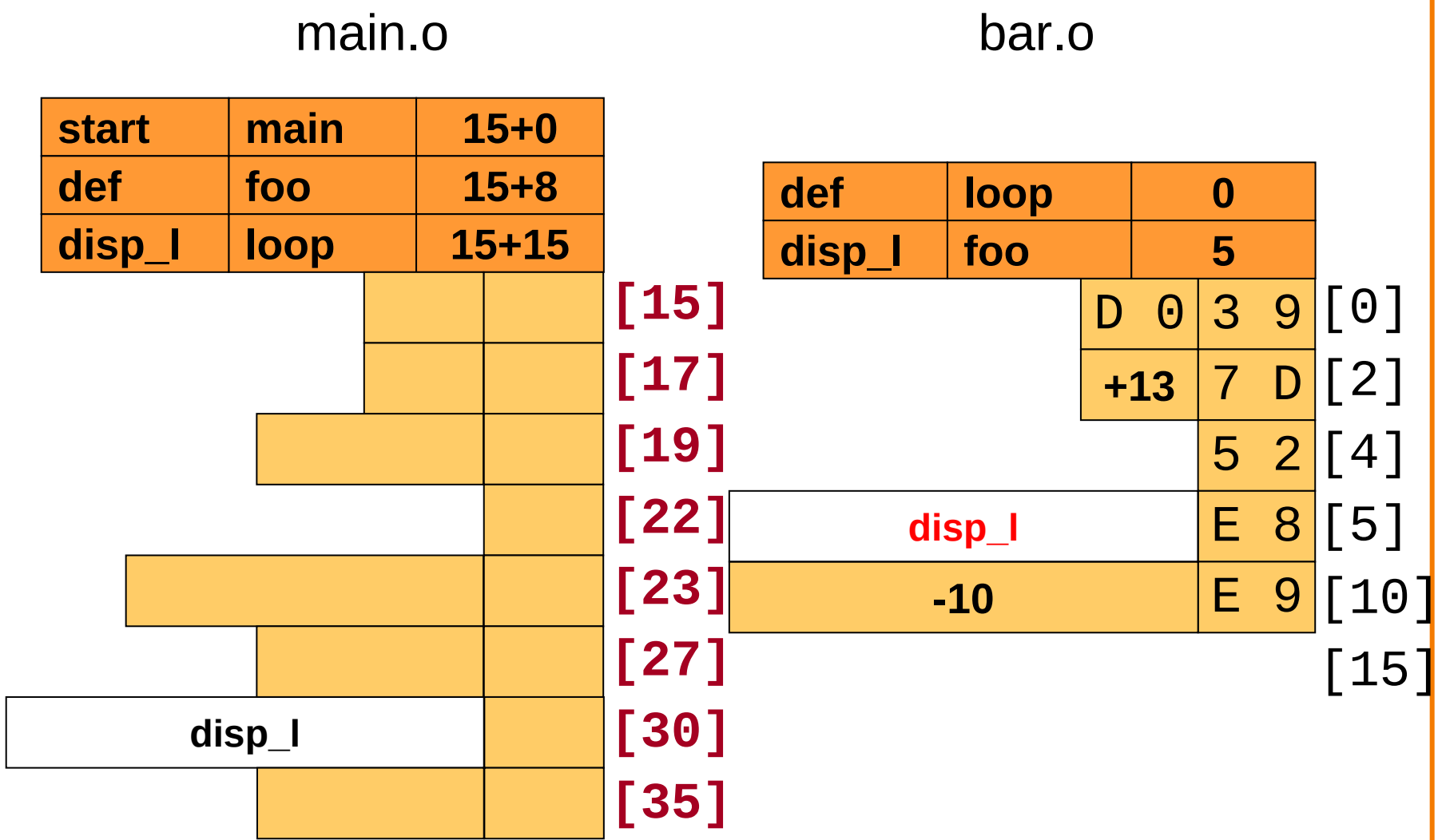
output (also in ".o" format, but no undefined symbols)

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

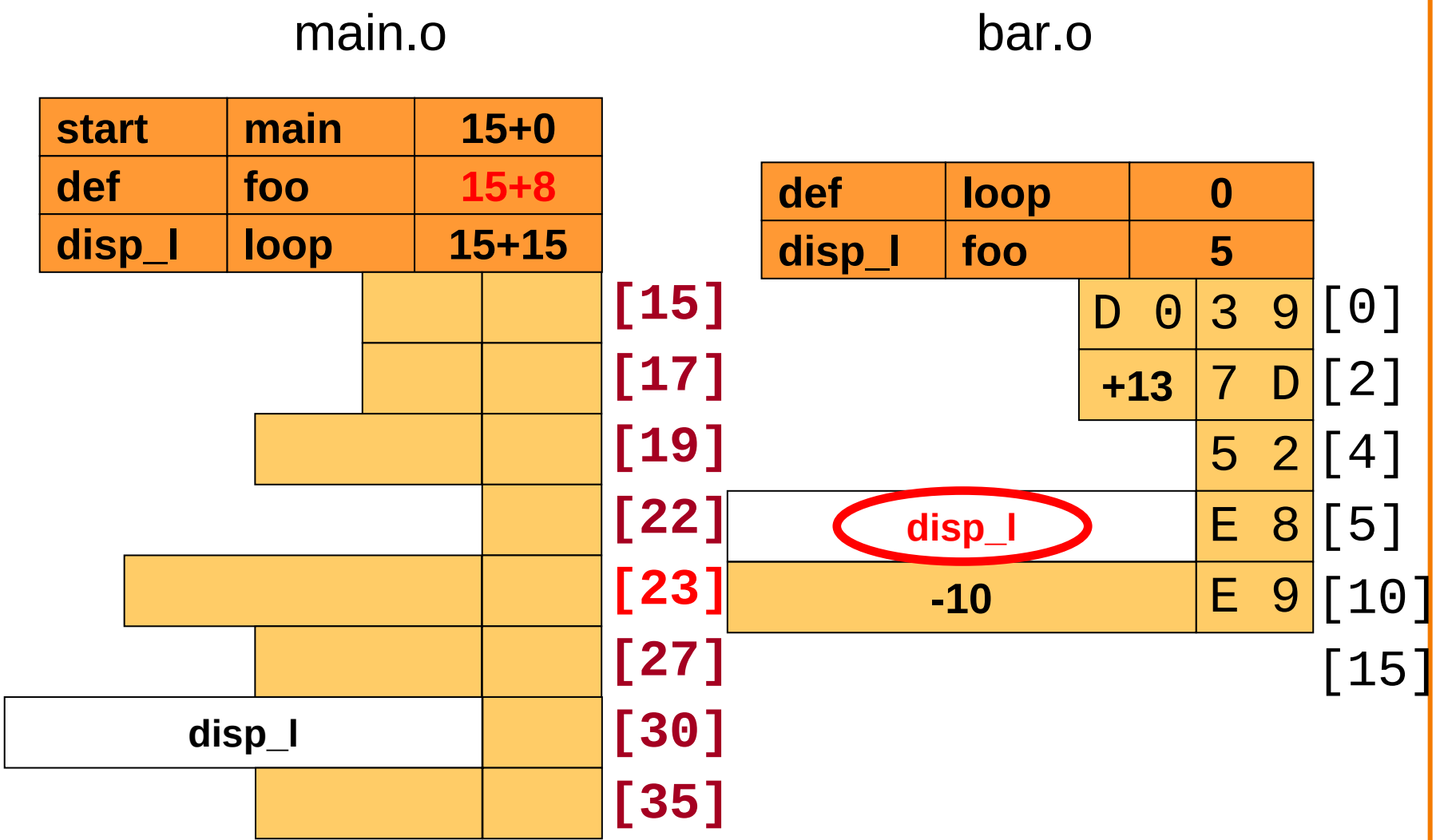
Multiple Object Files



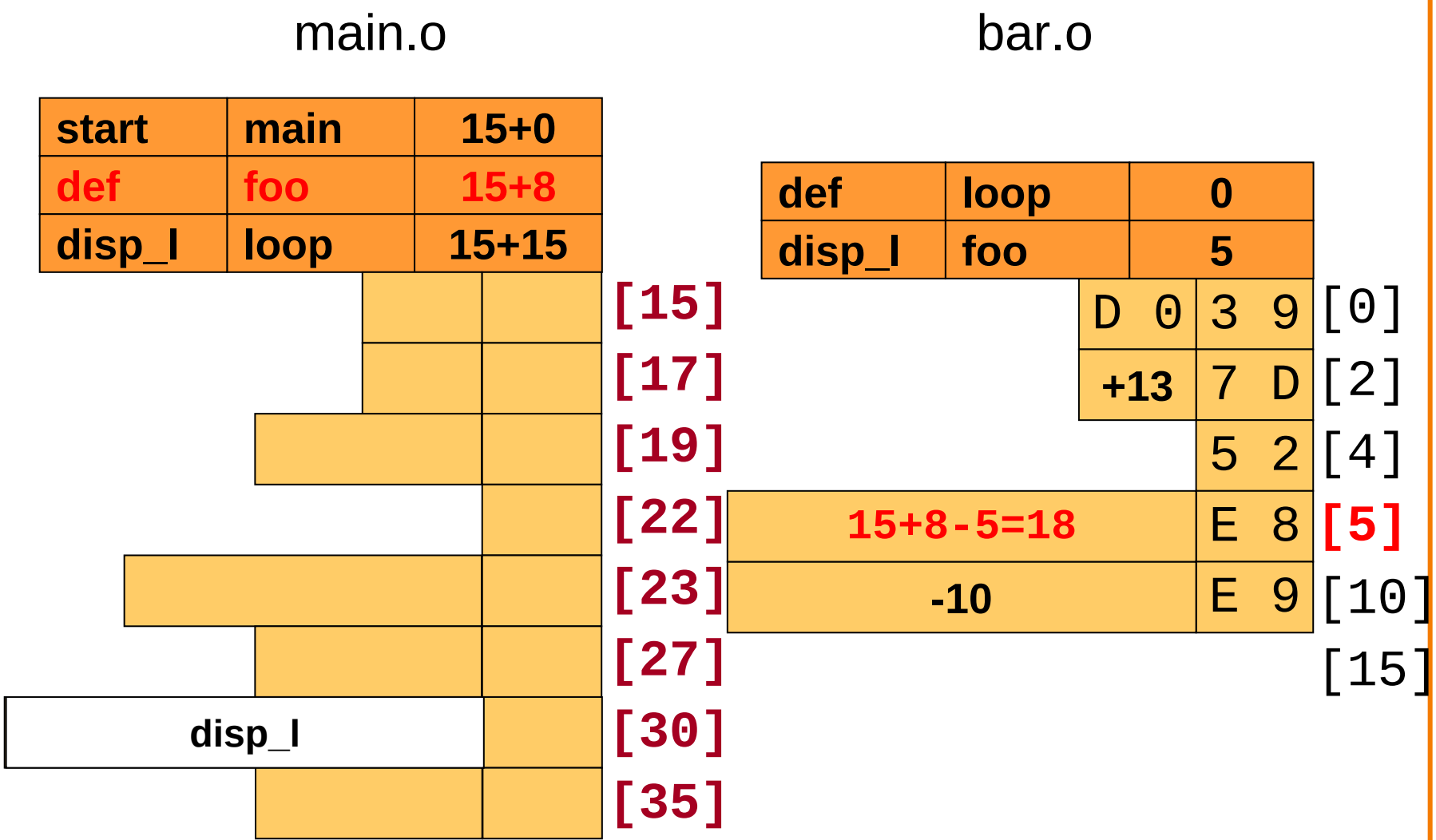
Step 1: Pick An Order



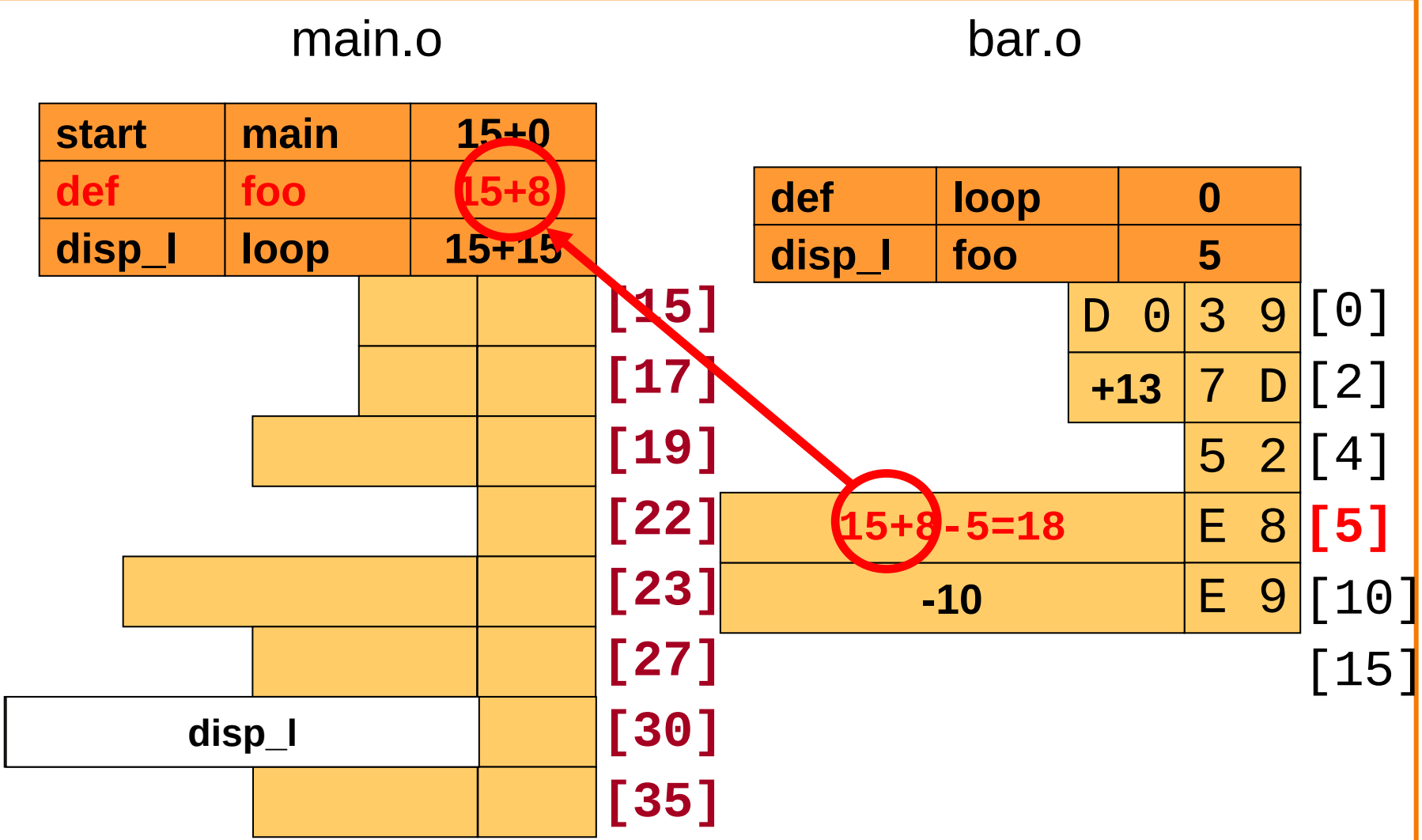
Step 1: Pick An Order



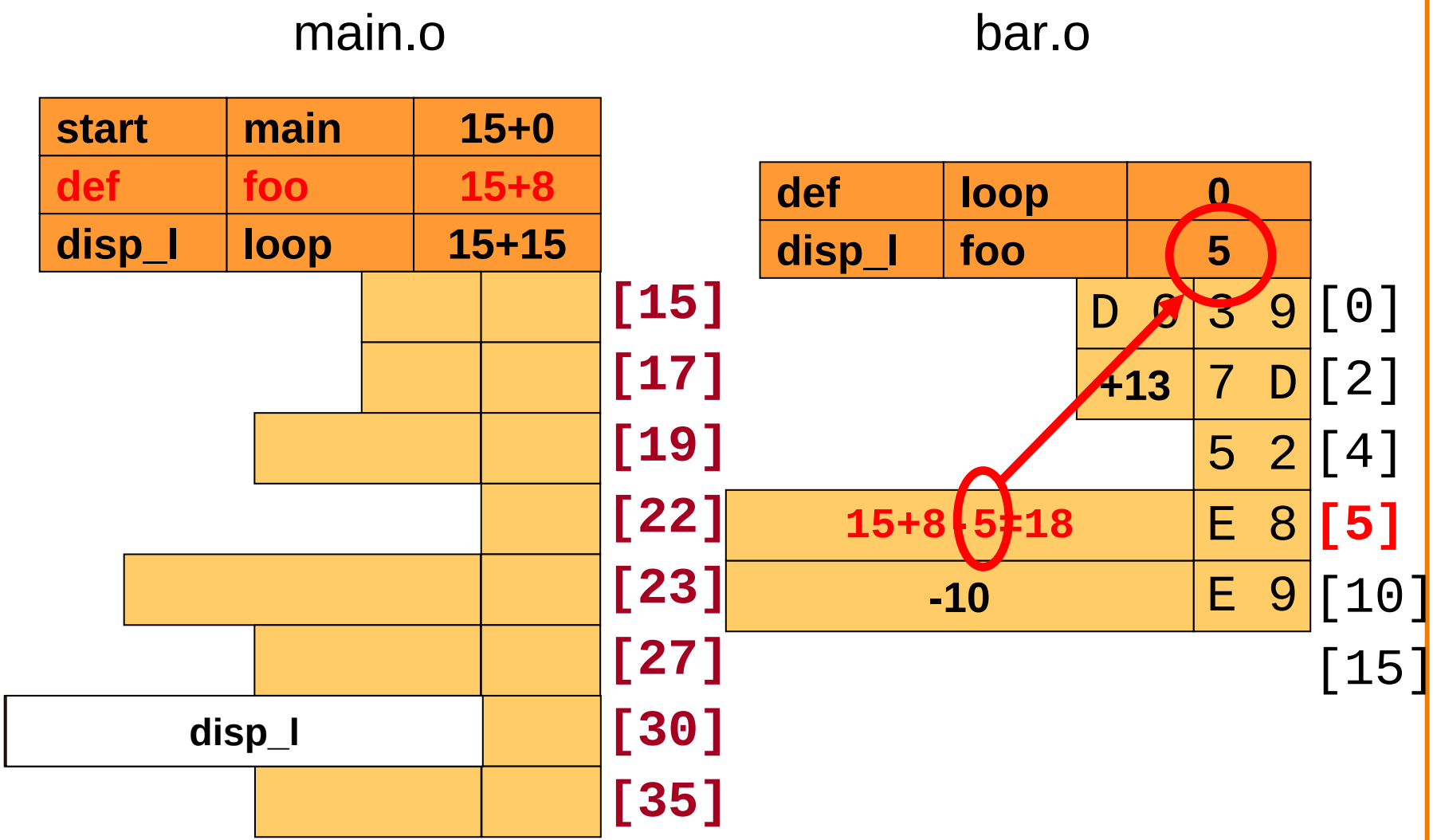
Step 2: Patch


main.o
bar.o
start
main
15+0
def
foo
15+8
disp_l
loop
15+15
[15]
[17]
[19]
[22]
[23]
[27]
disp_l
[30]
[35]
def
loop
0
disp_l
foo
5
D 0
3 9
[0]
+13
7 D
[2]
5 2
[4]
15+8-5=18
E 8
[5]
-10
E 9
[10]
[15]

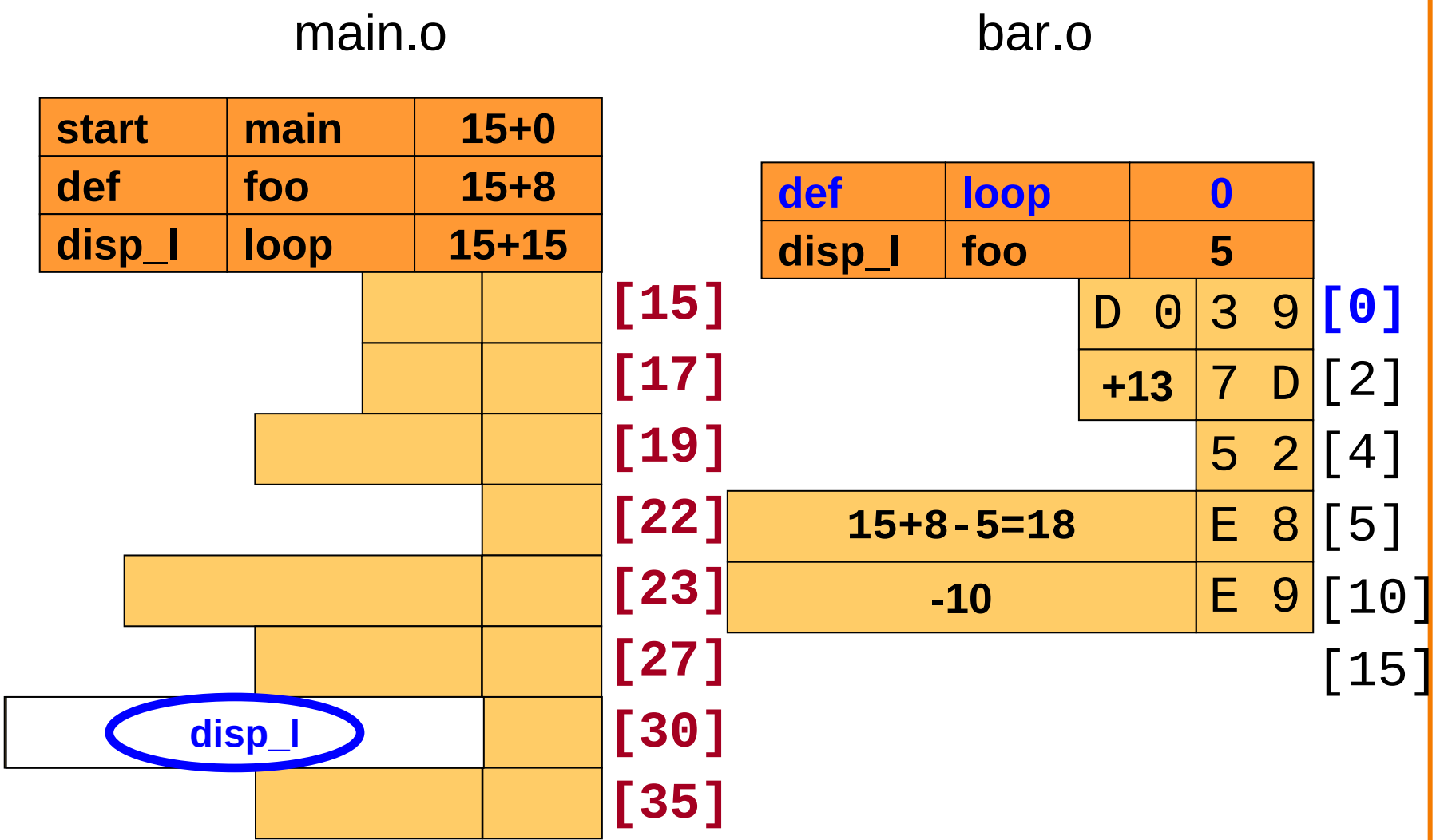
Step 2: Patch


main.o
bar.o
start
main
15+0
def
foo
15+8
disp_l
loop
15+15
def
loop
0
disp_l
foo
5
D 0
3 9
+13
7 D
5 2
15+8-5=18
E 8
-10
E 9
[0]
[2]
[4]
[5]
[10]
[15]
[15]
[17]
[19]
[22]
[23]
[27]
[30]
[35]
disp_l

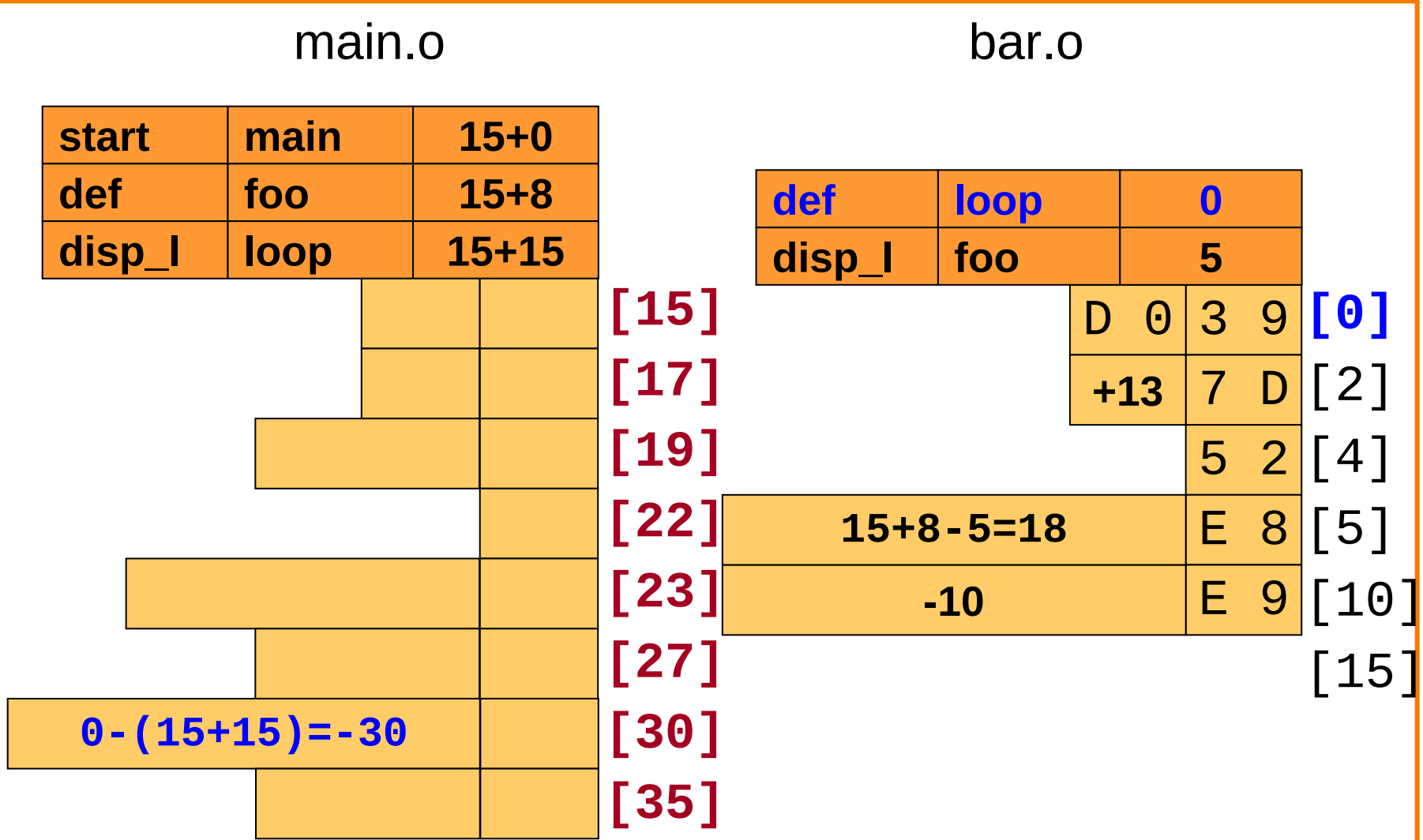
Step 2: Patch


main.o
bar.o
start
main
15+0
def
foo
15+8
disp_l
loop
15+15
def
loop
0
disp_l
foo
5
D 6
3 9
+13
7 D
5 2
15+8-5=18
E 8
-10
E 9
disp_l
[0]
[2]
[4]
[5]
[10]
[15]
[15]
[17]
[19]
[22]
[23]
[27]
[30]
[35]

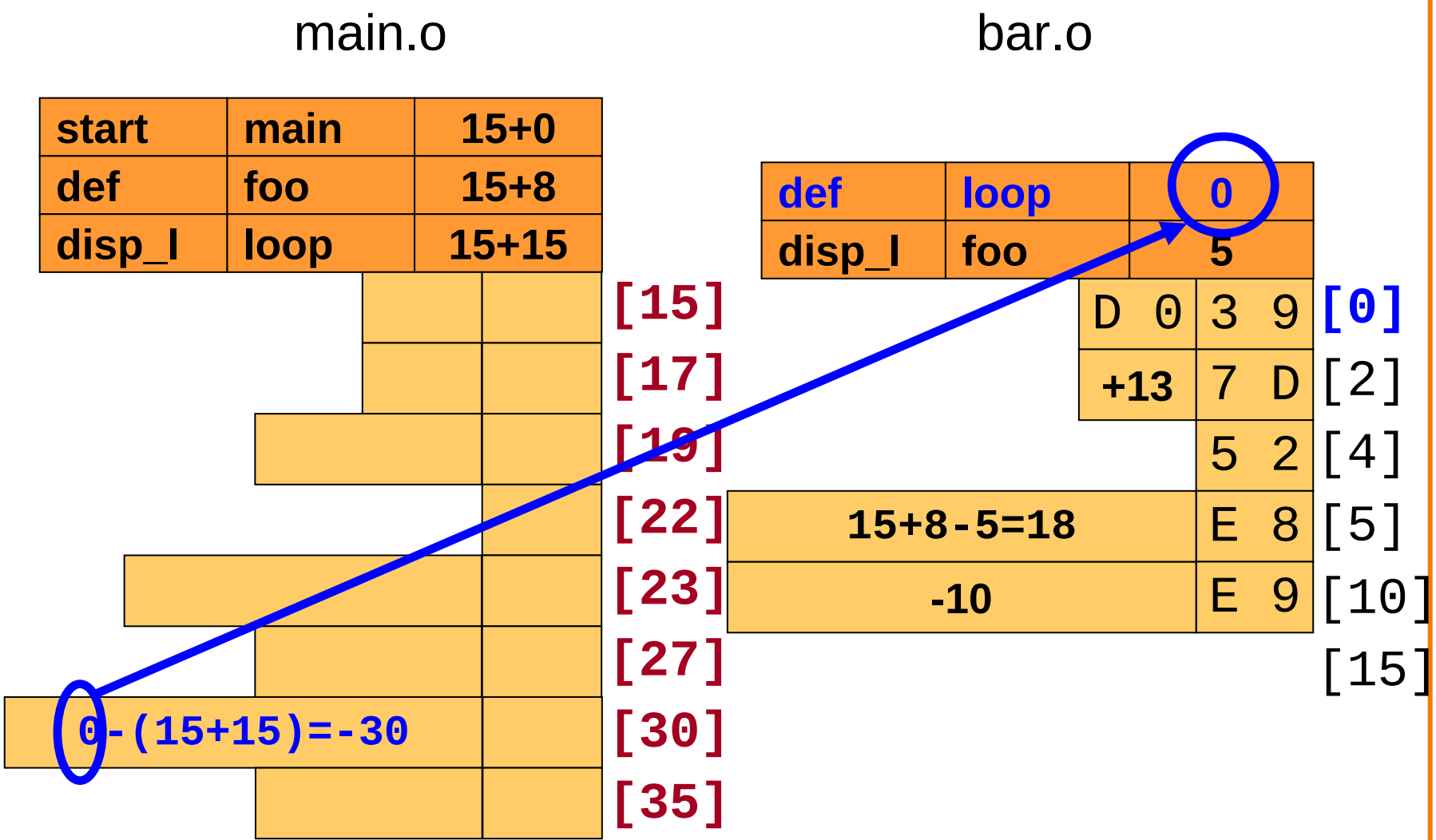
Step 2: Patch


main.o
bar.o
start
main
15+0
def
foo
15+8
disp_l
loop
15+15
[15]
[17]
[19]
[22]
[23]
[27]
disp_l
[30]
[35]
def
loop
0
disp_l
foo
5
D 0
3 9
[0]
+13
7 D
[2]
5 2
[4]
15+8-5=18
E 8
[5]
-10
E 9
[10]
[15]

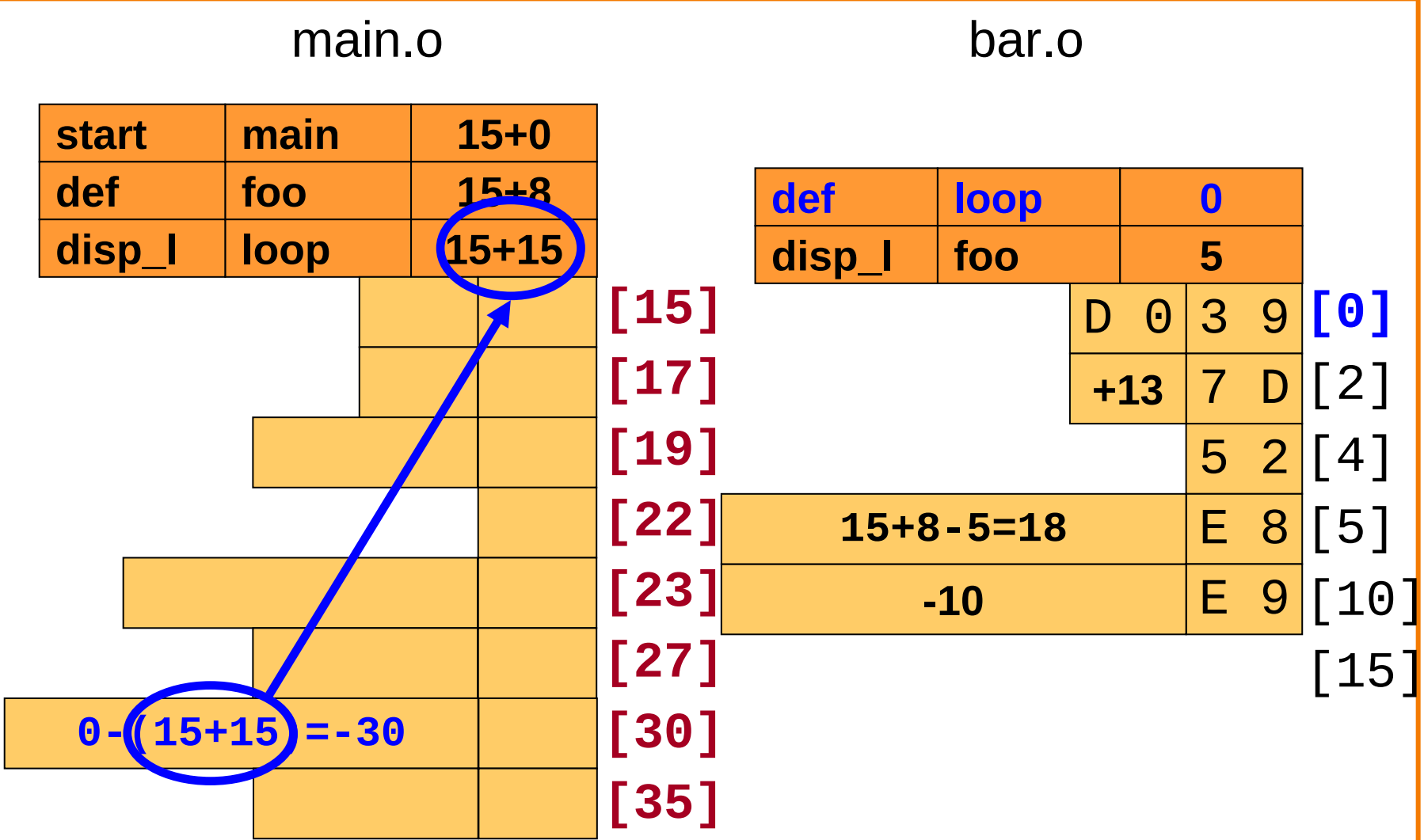
Step 2: Patch


main.o
bar.o
start
main
15+0
def
foo
15+8
disp_l
loop
15+15
[15]
[17]
[19]
[22]
[23]
[27]
0-(15+15)=-30
[30]
[35]
def
loop
0
disp_l
foo
5
D 0
3 9
[0]
+13
7 D
[2]
5 2
[4]
15+8-5=18
E 8
[5]
-10
E 9
[10]
[15]

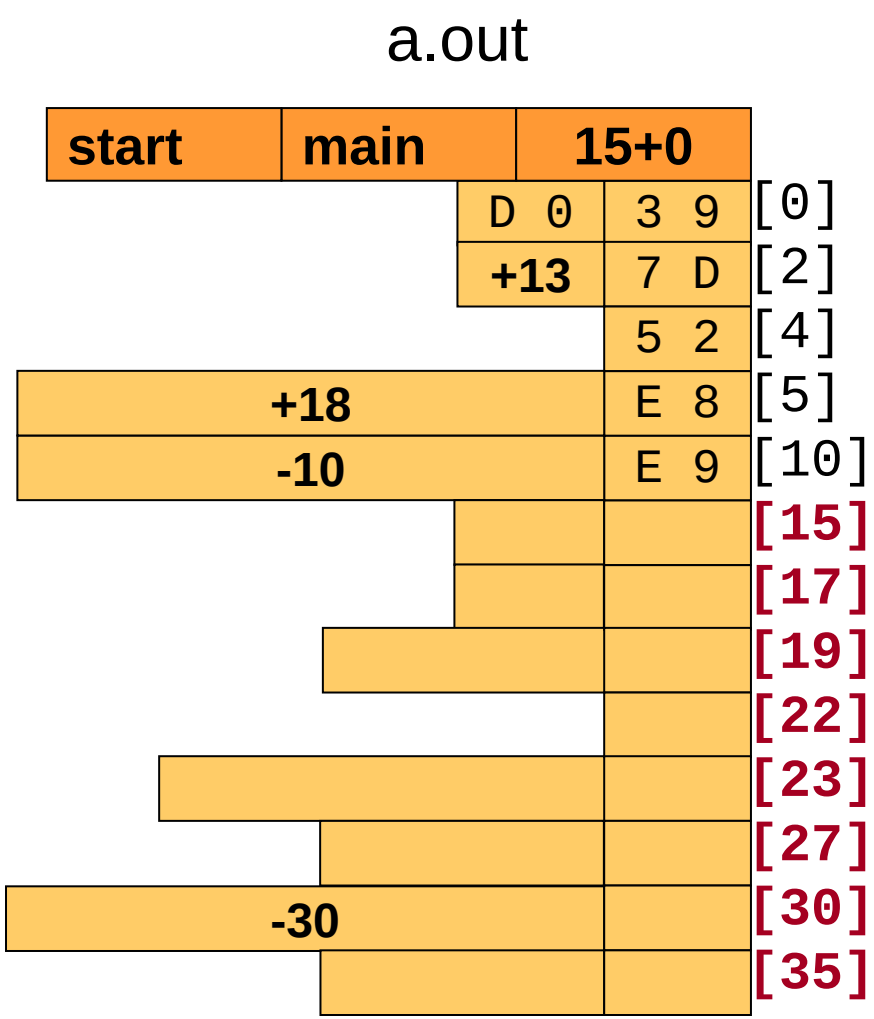
Step 2: Patch


main.o
bar.o
start
main
15+0
def
foo
15+8
disp_l
loop
15+15
def
loop
0
disp_l
foo
5
[15]
[17]
[19]
[22]
[23]
[27]
[30]
[35]
0-(15+15)=-30
D 0
3 9
[0]
+13
7 D
[2]
5 2
[4]
15+8-5=18
E 8
[5]
-10
E 9
[10]
[15]

Step 2: Patch


main.o
bar.o
start
main
15+0
def
foo
15+8
disp_l
loop
15+15
[15]
[17]
[19]
[22]
[23]
[27]
0-(15+15)=-30
[30]
[35]
def
loop
0
disp_l
foo
5
D 0
3 9
[0]
+13
7 D
[2]
5 2
[4]
15+8-5=18
E 8
[5]
-10
E 9
[10]
[15]

Step 3: Concatenate



Summary

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