



Assemblers and Linkers

Prof. David August
COS 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

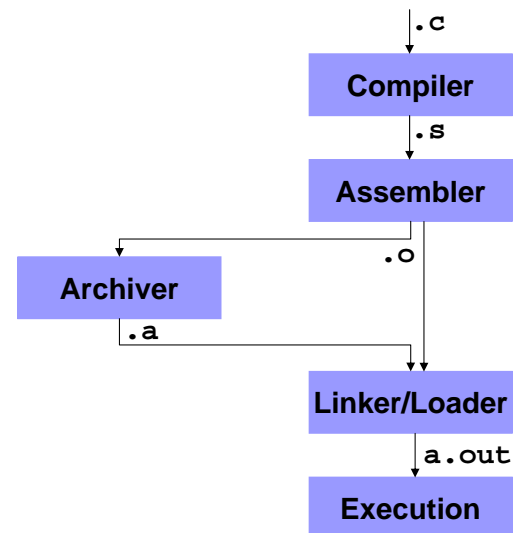


Compilation Pipeline

- **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 (execvp)**
 - 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

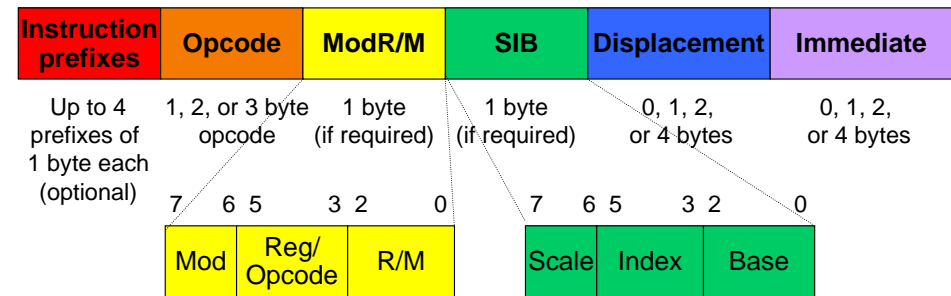
- A symbolic representation of machine instructions

• Machine language

- Contains everything needed to link, load, and execute the program

5

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)

6

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

7

Example: Load Effective Address



• Assembly language:

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

• Machine code:

- Byte 1: 8D (opcode for "load effective address")
- Byte 2: 04 (dest %eax, with scale-index-base)
- Byte 3: 80 (scale=4, index=%eax, base=%eax)

1000 1101

0000 0100

1000 0000

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

8

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:
```

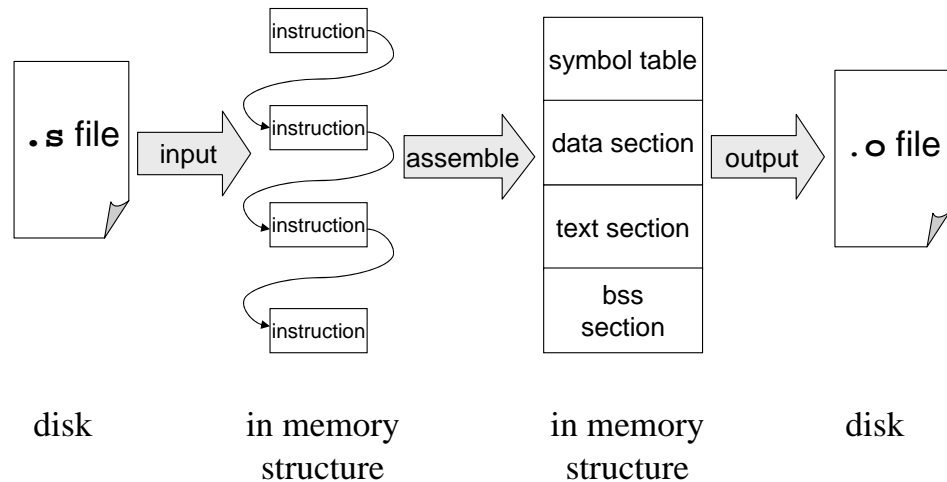
Create labels and remember their addresses
Deal with the "forward reference problem"

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

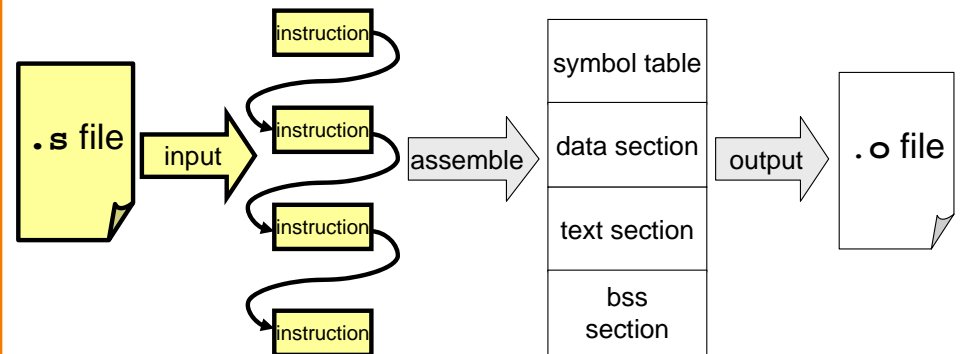
Implementing an Assembler



Input Functions



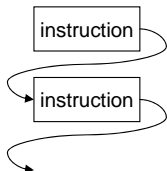
- Read assembly language and produce list of instructions



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]

Annotations: 1 byte → (pointing to the displacement field), 4 bytes → (pointing to the instruction box)

How to compute the address displacements?

Symbol Table



type	label	address
def	loop	0
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
def	loop	0
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



loop
done

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

```
.globl loop
loop:
    cml %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



loop
done

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

```
.globl loop
loop:
    cml %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]

Filling in Local Addresses



loop
done

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

```
.globl loop
loop:
    cml %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]
-10	E 9	[10]
		[15]

Filling in Local Addresses



loop
done

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

```
.globl loop
loop:
    cml %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]
-10	E 9	[10]
		[15]

Filling in Local Addresses



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


```

loop
done
    
```

```

.globl loop
loop:
    cml %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]
-10	E 9	[10]
		[15]

Relocation Records



```

...
.globl loop
loop:
    cml %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

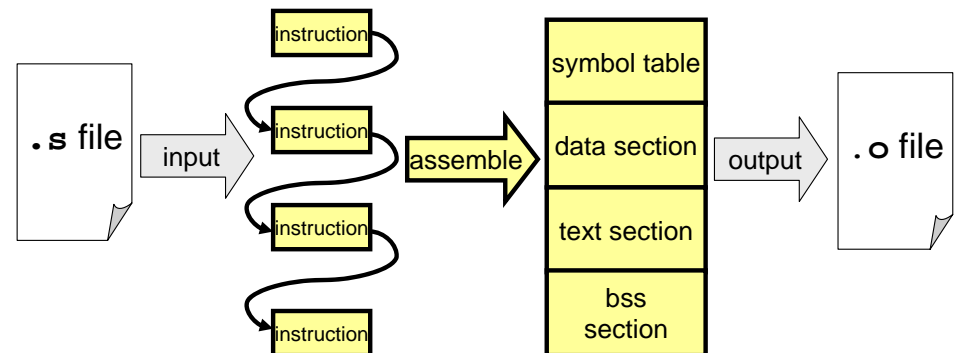


- Delineate segments
 - .section
- Allocate/initialize data and bss segments
 - .word .half .byte
 - .ascii .asciz
 - .align .skip
- Make symbols in text externally visible
 - .global

Assemble into Sections



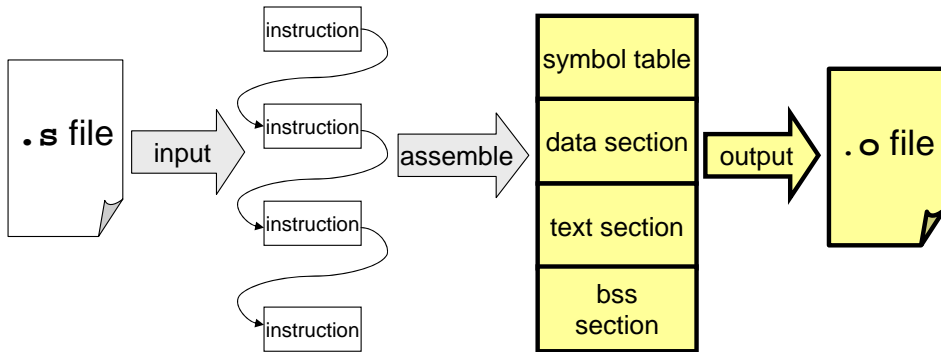
- Process instructions and directives to produce object file output structures



Output Functions



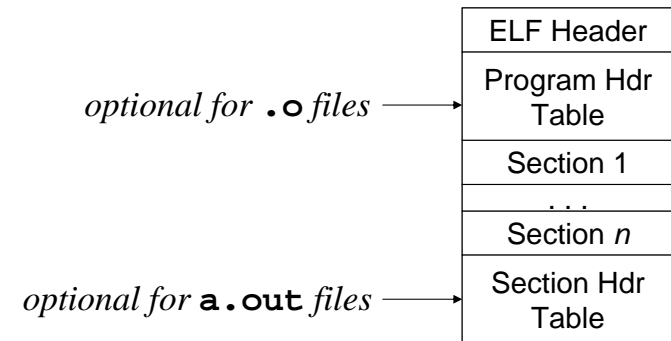
- Machine language output
 - Write symbol table and sections into object file



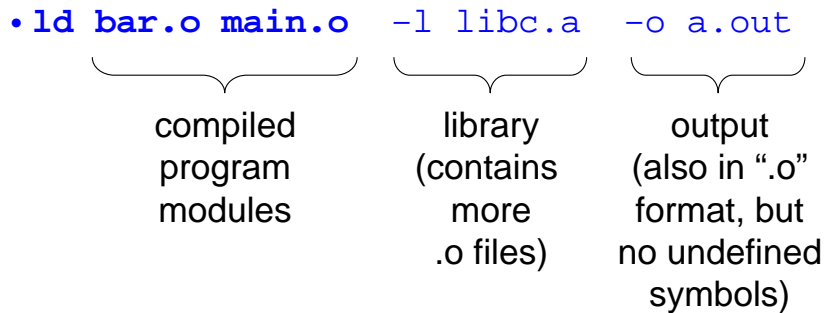
ELF: Executable and Linking Format



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

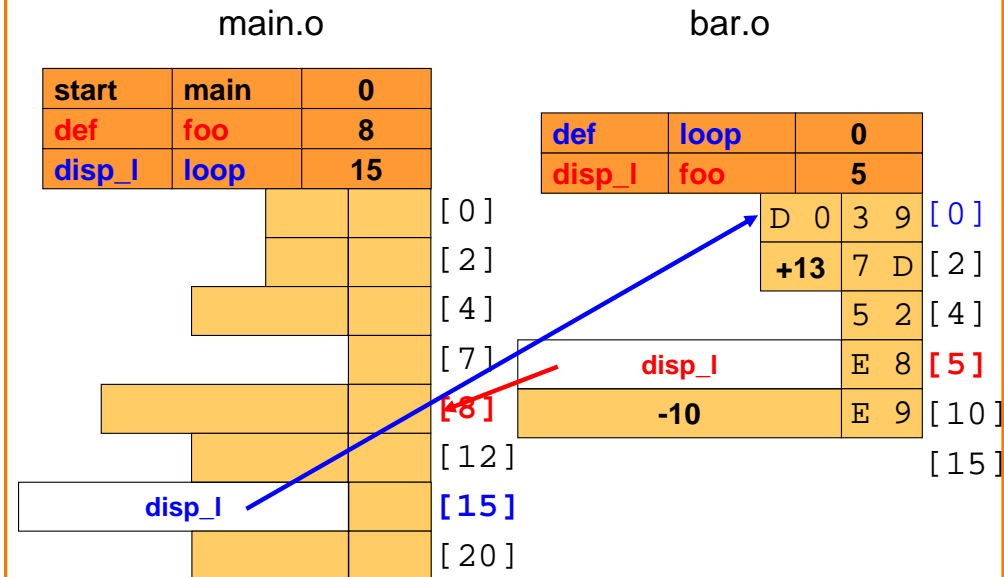


Invoking the Linker



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

Multiple Object Files



Step 1: Pick An Order

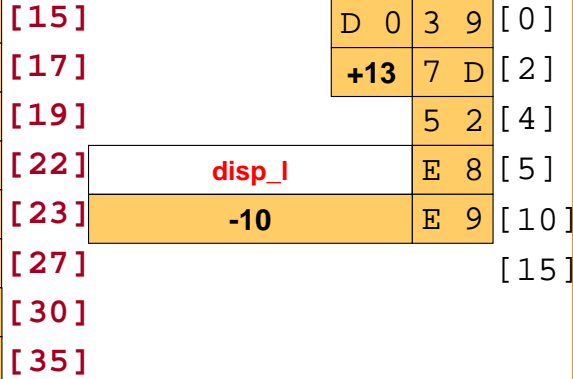


main.o

bar.o

start	main	15+0
def	foo	15+8
disp_l	loop	15+15

def	loop	0
disp_l	foo	5



Step 1: Pick An Order

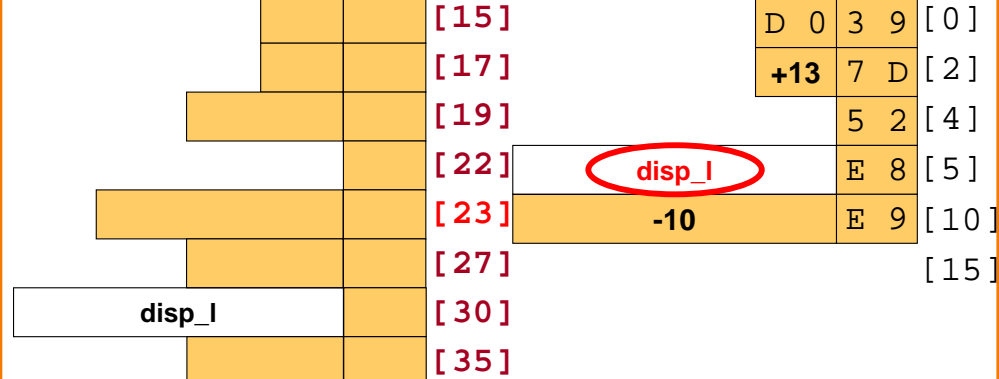


main.o

bar.o

start	main	15+0
def	foo	15+8
disp_l	loop	15+15

def	loop	0
disp_l	foo	5



Step 2: Patch

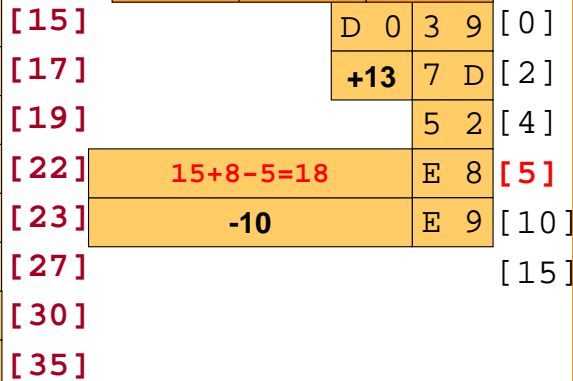


main.o

bar.o

start	main	15+0
def	foo	15+8
disp_l	loop	15+15

def	loop	0
disp_l	foo	5



Step 2: Patch

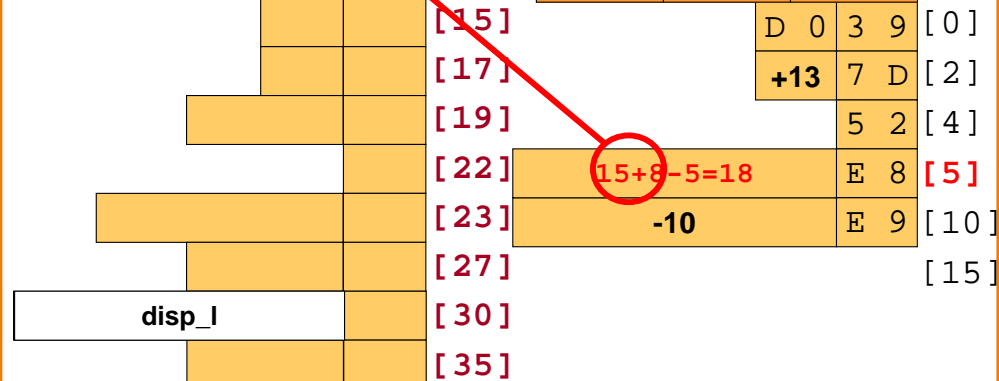


main.o

bar.o

start	main	15+0
def	foo	15+8
disp_l	loop	15+15

def	loop	0
disp_l	foo	5



Step 2: Patch

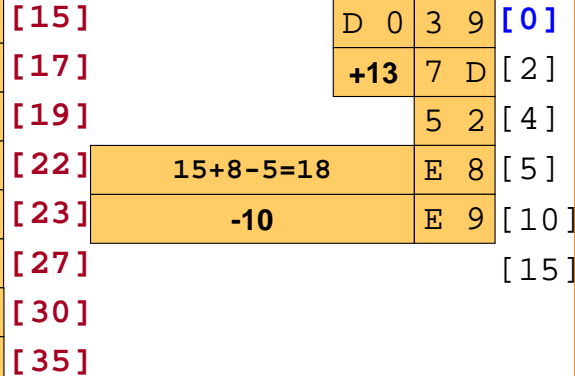


main.o

bar.o

start	main	15+0
def	foo	15+8
disp_l	loop	15+15

def	loop	0
disp_l	foo	5

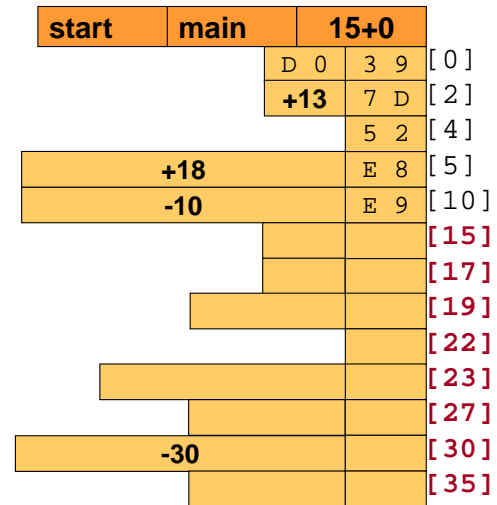


41

Step 3: Concatenate



a.out



42

Summary



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

43