

Lecture 3.1

Basic Assembler Functions
from source program to object program

Md Mahbub E Noor

Assistant Professor, Department of CSE
University of Barishal

```
bash - ~/lab3
$ ./sicas count.asm > count.lst
$ cat count.obj
HCOUNT 0020000000A0
T0020001E042033002033...
T00201E1B0C20363C200F...
T00209D03000001
E002000
```

What you will learn today



Name the five jobs of an assembler and the tables it uses for them (OPTAB, SYMTAB, LOCCTR).



Explain the forward-reference problem, and why reading the program twice solves it.



Run Pass 1 by hand: give every line an address and fill SYMTAB.



Run Pass 2 by hand and write the H, T and E records of the object program.

Today's plan (60 min)

0-5

Recap of Lecture 2

5-15

What an assembler does

15-30

Pass 1

30-45

Pass 2

45-55

The object program

55-60

Checkpoint, homework

Writing the assembler in C (slides 23-25) is the lab.

Quick recap of Lecture 2

Q1

Encode **LDCH TEXT, X** with **LDCH = 50** and **TEXT** at **2027**.

$x = 1$ adds 8000: $2027 + 8000 = A027$, so **50A027**. Today a program does this for every line.

Q2

What does **WORD 65** put in memory?

One word with the value $65 = 41$ hex: **000041**.

Q3

What does **BUFFER RESB 100** put in memory?

Nothing. It only reserves 100 bytes. But the address of the next line moves by 100.

PART 1

What an assembler does

Five jobs, two tables, and one problem that forces it to read the program twice.

The five jobs of an assembler



1. Mnemonic → opcode

LDA becomes 00. Like looking up a word in a dictionary. The dictionary is **OPTAB**.



2. Label → address

ZERO becomes 2033. The assembler must remember every label's address: **SYMTAB**.



3. Build the instruction

Join opcode, x bit and address in the right format: 04 + 2033 = 042033.



4. Constants → bytes

BYTE C 'BANANA' becomes 42 41 4E 41 4E 41; WORD 65 becomes 000041.



5. Write the output

The **object program** for the loader, and a **listing** for humans.

C link: gcc -c does the same five jobs for x86-64 when it runs as on your .s file (Lecture 1).

Our example program: count the A's in BANANA

C version

```
char text[6] = "BANANA";
int total = 0;

for (i = 0; i < 6; i++)
    if (text[i] == 'A')
        total++;
```

DIFFERENT FROM C

Data after code

In C you declare variables before using them. Here the data words come at the end, after the code that uses them.

count.asm (SIC)

```
COUNT    START    2000
FIRST    LDX      ZERO
LOOP     LDA      ZERO
          LDCH     TEXT,X
          COMP     LETTERA
          JEQ      FOUND
NEXT     TIX      LENGTH
          JLT      LOOP
          J        DONE
FOUND    LDA      TOTAL
          ADD      ONE
          STA      TOTAL
          J        NEXT
DONE     J        DONE
TEXT     BYTE     C 'BANANA '
LENGTH   WORD     6
LETTERA   WORD     65
ZERO     WORD     0
TOTAL    WORD     0
BUFFER    RESB    100
ONE       WORD     1
          END      FIRST
```

What it does

X steps through the six bytes of TEXT. Each byte is compared with LETTERA (65 = 'A'). On a match, the program jumps to FOUND and adds ONE to TOTAL.

We ran it: TOTAL = 3, after 50 instructions.

The problem: forward references

2000 FIRST LDX ZERO

forward: ZERO is defined at 2033

2006 LDCH TEXT,X

forward: TEXT is defined at 2027

200C JEQ FOUND

forward: FOUND is defined at 2018

2012 JLT LOOP

backward: LOOP was defined at 2003

2018 FOUND LDA TOTAL

forward: TOTAL is defined at 2036

10 of 13

instructions in COUNT use a label **before** it is defined. When the assembler reaches line 2000, it has no idea where ZERO is.

SAME AS C

C has the same problem

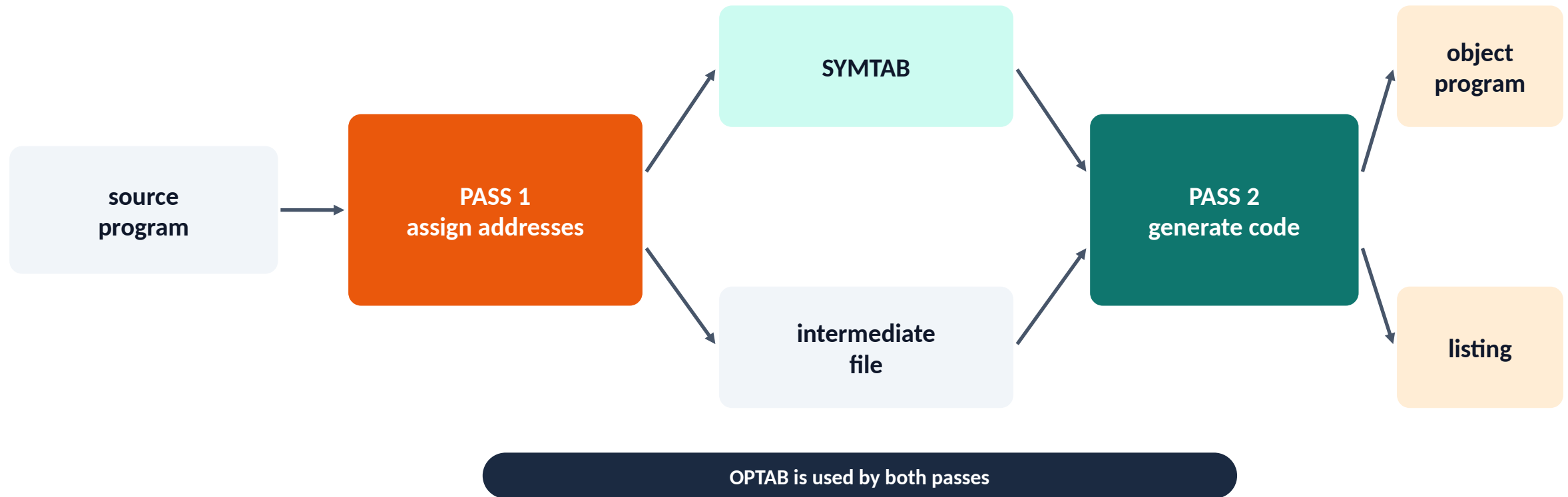
Call `counta()` before its definition and gcc warns: implicit declaration. A name must be known before it is used.

DIFFERENT FROM C

But a different solution

C makes **you** declare first (a prototype, or extern). The assembler instead **reads the program twice**: once to learn the addresses, once to use them.

The solution: read the program twice



Analogy: making the index of a book. First you read the whole book and write down the page of every topic. Only then can you print "see page 47" anywhere, even on page 3.

Book: Beck §2.1.2 (assembler algorithm and data structures)

PART 2

Pass 1: give every line an address

The location counter, the symbol table, and how many bytes each line needs.

Three data structures, and how they look in C

OPTAB

the opcode table

Fixed: the same for every program. Mnemonic → opcode (and size). Usually a hash table built into the assembler.

```
sicasm.c

struct op {
    const char *name;
    int code;
};
const struct op OPTAB[] =
    {{"ADD",0x18}, {"LDA",0x00},
    ...};
```

SYMTAB

the symbol table

Grows during Pass 1, read in Pass 2. Label → address. A hash table, because it is searched for every operand.

```
sicasm.c

struct sym {
    char name[16];
    int  addr;
};
struct sym SYMTAB[200];
int nsyms = 0;
```

LOCCTR

the location counter

One integer: the address the current line will get. Starts at the START operand, grows by each line's size.

```
sicasm.c

int locctr;

locctr = strtol(operand,
               NULL, 16);

...
locctr += 3;
```

In a real assembler OPTAB and SYMTAB are hash tables. In our lab, a simple array and a for-loop search are fine: SIC programs are small.

Book: Beck §2.1.2

The Pass 1 algorithm, in five steps

- 1 Read the first line. If the opcode is **START**, set LOCCTR to its operand (a hex number). Otherwise LOCCTR = 0.
- 2 For each line until **END**: if it has a **label**, search SYMTAB. Found → error "duplicate symbol". Not found → insert (label, LOCCTR).
- 3 Look at the **opcode** to find the size: an instruction or WORD = 3, RESW n = 3n, RESB n = n, BYTE = length of the constant. Unknown → error.
- 4 Write the line, with its LOCCTR, to the intermediate file. Then LOCCTR += size.
- 5 At **END**: program length = LOCCTR – starting address.

```
● ● ● the same in C (sicas.c)
while (fgets(buf, ...)) {
    split(buf, ln);
    ln->loc = locctr;
    if (ln->label[0])
        add_symbol(...);
    if (is_instr(op))
        locctr += 3;
    else if (WORD) ...
    else if (RESW) ...
    else if (RESB) ...
    else if (BYTE) ...
}
length = locctr - start;
```

Book: Beck §2.1.2, Pass 1 algorithm

How many bytes does each line take?

Line	Size	Example from COUNT	LOCCTR before → after
any instruction	3	FIRST LDX ZERO	2000 → 2003
WORD	3	LENGTH WORD 6	202D → 2030
RESW n	3 × n	TABLE RESW 10	+ 30 = + 1E
RESB n	n	BUFFER RESB 100	2039 → 209D
BYTE C'...'	1 per character	TEXT BYTE C'BANANA'	2027 → 202D
BYTE X'...'	1 per 2 hex digits	INPUT BYTE X'F1'	+ 1
START, END	0	COUNT START 2000	LOCCTR ← 2000

Trap: decimal vs hex

The operand of RESB, RESW and WORD is **decimal**. LOCCTR is **hex**.

RESB 100 adds 100 = **64 hex**:
2039 + 64 = 209D

Not 2039 + 100 = 2139!

Program length = 20A0 - 2000 = **A0 hex** = 160 bytes.

Pass 1 on COUNT: every line gets an address

LOC	label	op	operand
	COUNT	START	2000
2000	FIRST	LDX	ZERO
2003	LOOP	LDA	ZERO
2006		LDCH	TEXT,X
2009		COMP	LETTERA
200C		JEQ	FOUND
200F	NEXT	TIX	LENGTH
2012		JLT	LOOP
2015		J	DONE
2018	FOUND	LDA	TOTAL
201B		ADD	ONE

LOC	label	op	operand
201E		STA	TOTAL
2021		J	NEXT
2024	DONE	J	DONE
2027	TEXT	BYTE	C 'BANANA '
202D	LENGTH	WORD	6
2030	LETTERA	WORD	65
2033	ZERO	WORD	0
2036	TOTAL	WORD	0
2039	BUFFER	RESB	100
209D	ONE	WORD	1

SYMTAB	address
FIRST	2000
LOOP	2003
NEXT	200F
FOUND	2018
DONE	2024
TEXT	2027
LENGTH	202D
LETTERA	2030
ZERO	2033
TOTAL	2036
BUFFER	2039
ONE	209D

length
20A0
- 2000
= 00A0

12 labels, all known
at the end of Pass 1.

PART 3

Pass 2: write the machine code

Now SYMTAB is complete, so every operand, even a forward reference, has an address.

The Pass 2 algorithm, in six steps

- 1 Read each line of the intermediate file. Its LOCCTR is already there from Pass 1.
- 2 **Instruction:** opcode from OPTAB. Operand from SYMTAB; not there → error "undefined symbol". If it ends in ,X, set x = 1.
- 3 Assemble: **opcode** (8 bits) | **x** (1 bit) | **address** (15 bits) → 6 hex digits.
- 4 **BYTE / WORD:** convert the constant to hex. **RESB / RESW:** no code, but end the current text record.
- 5 Add the code to the current **T record**. If it would pass 30 bytes, write the record and start a new one.
- 6 At **END:** write the last T record, then the **E record** with the first instruction's address.

```
the same in C (sicas.c)
for (i = 0; i < nlines; i++) {
    opc = find_op(L[i].opcode);
    if (opc >= 0) {
        addr = find_sym(name);
        if (addr < 0) error...
        sprintf(objcode,
            "%02X%04X", opc,
            (x << 15) | addr);
    } else if (WORD) ...
    else if (BYTE) ...
}
```

Book: Beck §2.1.2, Pass 2 algorithm

Pass 2 on COUNT: every line gets its code

LOC	source			object code	how
2000	FIRST	LDX	ZERO	042033	LDX = 04, ZERO = 2033
2003	LOOP	LDA	ZERO	002033	LDA = 00, ZERO = 2033
2006		LDCH	TEXT,X	50A027	LDCH = 50, TEXT = 2027 + 8000 (x=1)
2009		COMP	LETTERA	282030	COMP = 28, LETTERA = 2030
200C		JEQ	FOUND	302018	JEQ = 30, FOUND = 2018
200F	NEXT	TIX	LENGTH	2C202D	TIX = 2C, LENGTH = 202D
2012		JLT	LOOP	382003	JLT = 38, LOOP = 2003
2015		J	DONE	3C2024	J = 3C, DONE = 2024
2018	FOUND	LDA	TOTAL	002036	LDA = 00, TOTAL = 2036
201B		ADD	ONE	18209D	ADD = 18, ONE = 209D
201E		STA	TOTAL	0C2036	STA = 0C, TOTAL = 2036
2021		J	NEXT	3C200F	J = 3C, NEXT = 200F
2024	DONE	J	DONE	3C2024	J = 3C, DONE = 2024
2027	TEXT	BYTE	C 'BANANA '	42414E414E41	ASCII: B=42 A=41 N=4E ...
202D	LENGTH	WORD	6	000006	decimal 6 = hex 000006
2030	LETTERA	WORD	65	000041	decimal 65 = hex 000041
2033	ZERO	WORD	0	000000	decimal 0 = hex 000000
2036	TOTAL	WORD	0	000000	decimal 0 = hex 000000
2039	BUFFER	RESB	100		no code (only reserves 100 bytes)
209D	ONE	WORD	1	000001	decimal 1 = hex 000001

Watch both passes run

The two-pass assembler at work

PASS 1PASS 2

LOC	source line	object code
2000	COUNT START 2000	
2000	FIRST LDX ZERO	042033
2003	LOOP LDA ZERO	002033
2006	LDCH TEXT,X	50A027
2009	COMP LETTERA	282030
200C	JEQ FOUND	
200F	NEXT TIX LENGTH	
2012	JLT LOOP	
2015	J DONE	
2018	FOUND LDA TOTAL	
201B	ADD ONE	
201E	STA TOTAL	
2021	J NEXT	
2024	DONE J DONE	
2027	TEXT BYTE C'BANANA'	
202D	LENGTH WORD 6	
2030	LETTERA WORD 65	
2033	ZERO WORD 0	
2036	TOTAL WORD 0	
2039	BUFFER RESB 100	
209D	ONE WORD 1	
20A0	END FIRST	

SYMTAB

symbol	address
FIRST	2000
LOOP	2003
NEXT	200F
FOUND	2018
DONE	2024
TEXT	2027
LENGTH	202D
LETTERA	2030
ZERO	2033
TOTAL	2036
BUFFER	2039
ONE	209D

COMP LETTERA → 282030

Watch for

1. In **Pass 1** only the LOC column and SYMTAB fill up. No code yet.
2. At **JEQ FOUND**, Pass 1 just counts 3 bytes and moves on.
3. In **Pass 2** the code column fills, using the finished SYMTAB.
4. **RESB** gets an address but no code.
5. The last frame: the **H, T, E** records.

Click the video to play or pause. About 1½ minutes, no sound.

PART 4

The object program

Three kinds of record tell the loader what to put where.

Three record types: Header, Text, End

H Header

one per program

H	program name	start address	length (bytes)
----------	---------------------	----------------------	-----------------------

columns: 1 · 2-7 · 8-13 · 14-19

Tells the loader the program's name, where it starts and how much memory it needs.

T Text

as many as needed

T	start address	length	object code: up to 30 bytes = 60 hex digits
----------	----------------------	---------------	--

columns: 1 · 2-7 · 8-9 · 10-69

Bytes to put into memory, starting at the given address.

E End

one per program

E	first instruction
----------	--------------------------

columns: 1 · 2-7

Where to jump when loading is done: the operand of END.

Book: Beck §2.1.1 (the simple object program format)

The object program of COUNT, field by field

```
H COUNT 002000 0000A0
T 002000 1E 04203300203350A0272820303020182C202D3820033C202400203618209D
T 00201E 1B 0C20363C200F3C202442414E414E41000006000041000000000000
T 00209D 03 000001
E 002000
```

H: Name COUNT (padded to 6), start 002000, length 0000A0 = 160 bytes.

T 2: Starts at 00201E, 1B = **27 bytes**, from STA TOTAL to the last WORD. It stops at the RESB.

E: Start execution at 002000 = FIRST, the operand of END.

T 1: Starts at 002000, 1E = **30 bytes**: full, so the assembler starts a new record.

T 3: Starts at 00209D, **after the 100-byte gap**: just ONE WORD 1.

count.obj and your C object file hello.o

```
cat count.obj (SIC, text)
```

```
HCOUNT 0020000000A0  
T0020001E04203300203350A0272820303020182C202...  
T00201E1B0C20363C200F3C202442414E414E4100000...  
T00209D03000001  
E002000
```

```
od -A x -t x1 hello.o | head -3 (x86-64, binary)
```

```
000000 7f 45 4c 46 02 01 01 00 00 00 ...  
000010 01 00 3e 00 01 00 00 00 00 00 ...  
000020 00 00 00 00 00 00 00 00 00 68 02 ...  
# 7f 45 4c 46 = \177 E L F
```

SAME AS C

Same job

Both carry **machine code plus where it goes**, and both start with a header: the H record, the ELF header. Both are made by an assembler and read by a loader.

DIFFERENT FROM C

Different form

count.obj is **text**: you can read it with `cat`. hello.o is **binary**, and it also stores a symbol table and relocation notes for the linker (Lecture 6).

LAB 3.1

Write your own assembler in C

About 150 lines of C you already know: arrays of structs, strcmp, sprintf and two loops.

The starter program: sicasm_starter.c

```
● ● ● sicasm_starter.c (inside main, Pass 1)
while (fgets(buf, sizeof buf, in)) {
    struct line *ln = &L[nlines];
    if (!split(buf, ln)) continue;
    if (strcmp(ln->opcode, "START") == 0) {
        start = locctr = strtol(..., 16);
        ...
    }
    ln->loc = locctr;
    if (ln->label[0]) {
        /* TODO 1: duplicate? else add
           (label, locctr) to SYMTAB */
    }
    /* TODO 2: locctr += size of line */
    nlines++;
}
```

Already written for you

OPTAB with all 26 SIC opcodes, find_op, find_sym, split (cuts a line into label, opcode, operand), and the whole H/T/E writer.

You write four TODOs

1 insert into SYMTAB · 2 LOCCTR rules · 3 encode an instruction · 4 BYTE constants. Each is 5 to 15 lines.

Test files

count.asm (must give the object program on slide 20), dup.asm and undef.asm (must give the errors on slide 25).

Lab 3.1 tasks, step by step

1

Build and run the starter

`gcc -Wall -o sicas sicas_starter.c` then `./sicas count.asm`. The output is wrong: that is expected.

2

TODO 1: SYMTAB

Insert each label. Test with `dup.asm`: it must stop with "duplicate symbol FIRST".

3

TODO 2: LOCCTR

Apply the size rules (slide 12). Check the LOC column against slide 13, and the length 00A0.

4

TODO 3: instructions

Opcode, x bit, address, `sprintf`. Test with `undef.asm`: "undefined symbol BETA".

5

TODO 4: BYTE

C'BANANA' → 42414E414E41, X'F1' → F1. Then every row of slide 16 must match.

6

Compare

`diff count.obj count_expected.obj` prints nothing when your assembler is right. Then try `sumn.asm`.

Submit: your `sicas.c`, the listing of `sumn.asm`, and `sumn.obj`.

Errors your assembler must report

Duplicate symbol

```
bash
$ ./sicas asm dup.asm
error: duplicate symbol FIRST
$ echo $?
1
```

Found in **Pass 1**, when FIRST is inserted a second time. Like two C functions with the same name.

Undefined symbol

```
bash
$ ./sicas asm undef.asm
error: undefined symbol BETA
$ echo $?
1
```

Found only in **Pass 2**: in Pass 1, BETA might still be defined further down.

The starter's warning

```
bash
$ gcc -Wall -o sicas sicas_starter.c
warning: 'byte_len' defined but not used
```

Not a bug: TODO 2 will use `byte_len`. A good habit: build with `-Wall` and read every warning.

C link: gcc reports the same two kinds of error: "redefinition of" (duplicate) and "undeclared" (undefined). A compiler is an assembler's bigger cousin.

Checkpoint: test yourself

- Q1** Why does an assembler need two passes? → forward references: a label may be used before it is defined
- Q2** At 2039: BUFFER RESB 100. LOCCTR for the next line? → 209D (100 = 64 hex)
- Q3** Which table is the same for every program? → OPTAB
- Q4** What does T00201E1B... say? → text record: start at 201E, 1B = 27 bytes of code
- Q5** Which pass reports "undefined symbol", and why? → Pass 2: only then is SYMTAB complete

Key ideas from today



An assembler does five jobs; only "label → address" needs to look ahead.



Forward references force two passes: Pass 1 learns the addresses, Pass 2 uses them.



Pass 1 = LOCCTR + SYMTAB. Each line adds its size; RESB/RESW operands are decimal.



Pass 2 = OPTAB + SYMTAB → opcode | x | address, plus constants.



Object program: H (name, start, length), T (≤ 30 bytes), E (first instruction).



In C it is just arrays of structs and two loops, which you write in the lab.

Homework 3.1 (due before Lecture 3.2)

1. Pass 1

For `sumn.asm` (in the lab files): write `LOCCTR` for every line, the full `SYMTAB`, and the program length.

2. Pass 2

Write the object code of every line of `sumn.asm`, with the "how" column as on slide 16.

3. Records

Write the complete object program (H, T, E) of `sumn.asm`. Explain where each T record starts and ends.

4. Trace

Run `sumn.asm` by hand. What is `SUM` at the end? Why does the loop body run 5 times, even though `JLT` comes after `STA SUM`?

5. C

Add to your assembler a warning "label never used" for every `SYMTAB` entry no operand refers to. What does it print for `count.asm` and `sumn.asm`?

Next lecture and reading

NEXT: LECTURE 3.2

Machine-dependent assembler features

The same two passes for SIC/XE: choosing formats 1–4, PC-relative and base-relative addressing.

Program relocation: why a program must work at any address, and the M record.

Main text

L. L. Beck, **System Software**, 3rd ed., 1997. Today: §2.1. Read §2.2 before Lecture 3.2.

C link

B. W. Kernighan & D. M. Ritchie, **The C Programming Language**, 2nd ed., ch. 6 (structures, table lookup) — for the lab.

Real assemblers

The **GNU as** manual (sourceware.org/binutils): the chapters on symbols and on sections. Copy it to the lab share for offline use.

Lab files

`lab3_files.zip`: starter, test programs, expected output.