

Lecture 3.3

Machine-Independent Features and assembler design options

Md Mahbub E Noor

Assistant Professor, Department of CSE
University of Barishal

countc.obj (control sections)

```
HMAIN 000000000019
DTEXT 000013TEXTEND000019
RCOUNTA
M00000405+COUNTA
...
M00002506+TEXTEND
M00002506-TEXT
```

What you will learn today



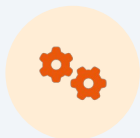
Use literals and LITORG, and show where the assembler places them (LITTAB).



Define symbols with EQU and expressions, and tell absolute from relative ones.



Explain program blocks and control sections, and read D, R and M records with symbols.



Compare one-pass, two-pass and multi-pass assemblers, and patch forward references.

Today's plan (60 min)

0-5

Recap of Lecture 3.2

5-15

Literals

15-25

EQU and expressions

25-40

Blocks and control sections

40-55

One-pass, multi-pass

55-60

Checkpoint, homework

Adding EQU and literals to your C assembler (slide 22) is the lab.

Quick recap of Lecture 3.2

Q1

Why is **STA TOTAL** base-relative in **COUNTX**?

TOTAL is 4127 bytes away; PC-relative reaches only 2047. So it uses (B) = 1041 with disp 000.

Q2

What does **M00002305** mean?

Add the program's start address to the 5 half-bytes starting in byte 0023. Today M records also name a symbol.

Q3

Why do PC-relative instructions need no M record?

They store a distance, and distances do not change when the whole program moves.

PART 1

Literals

Write the constant where you use it; let the assembler find it a home.

A literal is the assembler's string constant

● ● ● C you already write

```
const char *t = "BANANA";  
if (c == 'A') ...  
/* where does "BANANA" live? */  
/* you never named it */
```

● ● ● gcc -S lit.c (real output)

```
.section .rodata  
.LC0:  
.string "BANANA"  
...  
leaq .LC0(%rip), %rax
```

● ● ● SIC: count with a literal

```
COMP =X'000041'  
ADD =X'000001'  
* the assembler makes  
* a word for each one
```

SAME AS C

Same idea as a C string literal

You write the **value** where you need it. The compiler or assembler creates a hidden, unnamed constant in memory (gcc calls it `.LC0`) and puts its **address** into the instruction.

DIFFERENT FROM C

Not the same as # immediate

COMP #65: the value 65 is **inside** the instruction (SIC/XE only, at most 4095).

COMP =X'000041': the value is in a **separate word**; the instruction holds its address. Works in plain SIC and for any size.

Book: Beck §2.3.1 (literals)

LITTAB and LTORG: where literals end up

LOC	source			code
2000	FIRST	LDX	=X'000000'	042024
2003	LOOP	LDA	=X'000000'	002024
2006		LDCH	TEXT,X	50A030
2009		COMP	=X'000041'	282027
200C		JEQ	FOUND	302018
200F	NEXT	TIX	=X'000006'	2C202A
2012		JLT	LOOP	382003
2015	DONE	J	DONE	3C2015
2018	FOUND	LDA	TOTAL	002036
201B		ADD	=X'000001'	18202D
201E		STA	TOTAL	0C2036
2021		J	NEXT	3C200F
	LTORG			
2024	*	=X'000000'		000000
2027	*	=X'000041'		000041
202A	*	=X'000006'		000006
202D	*	=X'000001'		000001
2030	TEXT	BYTE	C'BANANA'	42414E414E41
2036	TOTAL	WORD	0	000000

Pass 1 meets =X'000041'

Not in LITTAB yet: add it, **without** an address. A second use of the same literal is not added again.

Pass 1 meets LTORG

Every waiting literal gets the next addresses (2024, 2027, 202A, 202D), like data written right here.

Pass 2

The instruction uses the literal's address from LITTAB: COMP =X'000041' → 282027.

No LTORG?

The literals go after END, at the very end of the program. LTORG lets you keep them **near** the code that uses them.

PART 2

Symbol definitions and expressions

EQU, ORG, and the arithmetic an assembler is allowed to do.

EQU: the assembler's #define

● ● ● C

```
#define MAXLEN 6
enum { A = 0, X = 1, L = 2 };
char text[] = "BANANA";
int tlen = sizeof text - 1; /* 6 */
```

● ● ● SIC/XE

```
MAXLEN    EQU    6
RA         EQU    0      ; register names
TEXT      BYTE  C'BANANA'
TEXTEND    EQU    *
TLEN       EQU    TEXTEND-TEXT    ; = 6
```

A name for a number

LDT #MAXLEN assembles to 750006, exactly like LDT #6. Change the 6 in one place and every use follows.

A name for an address

TEXTEND EQU * gives TEXTEND the current LOCCTR (here 0039). * means "here".

Computed by the assembler

TEXTEND - TEXT = 0039 - 0033 = 6.
The program never computes it at run time; the length is fixed when you assemble.

Book: Beck §2.3.2 (symbol-defining statements)

Absolute and relative expressions: C pointer rules

Expression	Result	Example	C pointer rule
number	absolute	MAXLEN EQU 6	an int
label	relative	TEXT	a pointer
label - label	absolute	TEXTEND - TEXT = 6	p - q is an int (ptrdiff_t)
label ± number	relative	TEXT+3	p + 3 is a pointer
label + label	ERROR	TEXT+TEXTEND	p + q is not allowed
label × number	ERROR	TEXT*2	p * 2 is not allowed

Only relative symbols ever need an M record, because only they change when the program moves.

```
gcc -c ptr.c (real)
```

```
char *bad = p + q;  
error: invalid operands  
to binary + (have  
'char *' and 'char *')
```

Why the rule?

When the program moves, a relative value moves too. label - label stays the same, so it is safe. label + label would move **twice**: no loader could fix it.

Book: Beck §2.3.3 (expressions)

PART 3

Program blocks and control sections

Arrange the pieces of one program, or split a program into pieces.

Program blocks: write in any order, store in groups

countb.asm (parts)

```
FIRST  STL  RETADR
      ...
      J    @RETADR
      USE  CDATA
RETADR RESW 1
CNT     WORD 0
      USE  CBLKS
BUFFER  RESB 4096
      USE  CDATA
TEXT    BYTE C 'BANANA '
      USE
FOUND   LDA  CNT
      ...
```

Block	Number	Start	Length
(default)	0	0000	0027
CDATA	1	0027	000C
CBLKS	2	0033	1000

(default) code · 0000

CDATA · 0027

CBLKS · 0033

the program in memory

How it works

Pass 1 keeps **one LOCCTR per block**. At the end, blocks are placed one after another, and each block's start is added to the addresses inside it.

SAME AS C

Your C program has blocks

size blocks.o (real):

text 170 · data 4 · bss 4128

Code, initialised data and the big uninitialised buffer, kept apart.

Book: Beck §2.3.4 (program blocks)

The object program of COUNTB

HCOUNTB000000001033

T0000001B172024B410750006B40053A020290041332008B8503B2FF03E200C

T00002A0900000042414E414E41

T00001B0C03200C1900010F20063F2FEC

E000000

Records follow the source, not the addresses: T 000000 (code), then T 00002A (CDATA: CNT and TEXT), then T 00001B (the code after USE). The loader puts each one at its own address; order does not matter.

CBLKS makes no record: BUFFER is RESB: it only adds 1000 to the program length (H: 001033).

No M records: Every instruction is PC-relative or immediate. Putting the buffer last made format 4 unnecessary.

Control sections: separately assembled parts, like C files

main.c

```
char text[] = "BANANA";
int counta(void);

int main(void) {
    printf("%d\n", counta());
}
```

counta.c

```
extern char text[];
int counta(void) {
    ... count 'A' in text ...
}
```

countc.asm (parts)

```
MAIN    START    0
        EXTDEF   TEXT,TEXTEND
        EXTREF   COUNTA
FIRST   STL      RETADR
        +JSUB    COUNTA
        STA      CNT
        ...
TEXT    BYTE     C 'BANANA '
TEXTEND EQU      *
COUNTA CSECT
        EXTREF   TEXT,TEXTEND
        ...
        +LDCH    TEXT,X
LEN     WORD     TEXTEND-TEXT
```

C	SIC/XE	Meaning
a .c file	CSECT	a part assembled on its own, with LOCCTR from 0
a global (non-static) definition	EXTDEF	"others may use this name"
extern / a prototype	EXTREF	"this name is defined elsewhere"
nm: T, D	D record	the names this part defines
nm: U	R record	the names this part needs

Book: Beck §2.3.5 (control sections and program linking)

D, R and M records: the linker's to-do list

```
HMAIN 000000000019
DTEXT 000013TEXTEND000019
RCOUNTA
T0000000D17200A4B1000000F20063E2000
T0000130642414E414E41
M00000405+COUNTA
E000000
HCOUNTA0000000000028
RTEXT TEXTEND
T0000001DB410B44077201EB40053900000290...
T00001D0B01000190043F2FEE000000
M00000A05+TEXT
M00002506+TEXTEND
M00002506-TEXT
E
```

D Define

DTEXT 000013TEXTEND000019: name + address, for every EXTDEF. Like the T and D lines of nm.

R Refer

RCOUNTA, RTEXT TEXTEND: the names this section needs. Like nm's U lines.

M with a symbol

M00000405+COUNTA: add the address of COUNTA (not of this section) to the 5 half-bytes at 0004.

Two M for one word

LEN WORD TEXTEND - TEXT is assembled as 000000 with M...+TEXTEND and M...-TEXT. The linker adds one, subtracts the other: $0019 - 0013 = 6$.

PART 4

Assembler design options

Must an assembler read the program twice? Is twice always enough?

One-pass assemblers: remember the holes

A second pass costs time: the whole program is read (or stored) twice. A **one-pass** assembler writes code at once, and for a label that is not known yet it leaves a **hole** plus a note: "fill me in later".

1 Data before code

Easy part: define data first, so data references are never forward. Jumps forward (an if, a loop exit) are still possible.

2 A list per symbol

SYMTAB marks an unknown symbol with * and keeps a **linked list** of the places that used it.

3 Patch when defined

When the label finally appears, walk the list and write its address into every hole.

4 At END

Any symbol still marked * is undefined: an error.

SAME AS C

Like a C linked list

```
struct fix { int addr; struct fix *next; }; One list  
per undefined symbol: the same code you wrote in Data Structures.
```

SAME AS C

Like gcc + ld

gcc's assembler leaves 00 00 00 00 for puts (Lecture 1) and the linker fills it. A one-pass assembler fills its own holes.

Book: Beck §2.4.1 (one-pass assemblers)

Watch a one-pass assembler fill its holes

A one-pass assembler: patch the holes later

LOC	source line			code in memory
2000	FIRST	LDX	ZERO	040000
2003	LOOP	LDA	ZERO	000000
2006		LDCH	TEXT,X	50A027
2009		COMP	LETTERA	280000
200C		JEQ	FOUND	302018
200F	NEXT	TIX	LENGTH	2C0000
2012		JLT	LOOP	382003
2015		J	DONE	3C2024
2018	FOUND	LDA	TOTAL	000000
201B		ADD	ONE	180000
201E		STA	TOTAL	0C0000
2021		J	NEXT	3C200F
2024	DONE	J	DONE	3C2024

SYMTAB

symbol	address	still waiting at
FIRST	2000	
ZERO	*	2000 → 2003
LOOP	2003	
TEXT	2027	
LETTERA	*	2009
FOUND	2018	
NEXT	200F	
LENGTH	*	200F
DONE	2024	
TOTAL	*	2018 → 201E
ONE	*	201B

SYMTAB

symbol	address	still waiting at
FIRST	2000	
ZERO	*	2000 → 2003
LOOP	2003	
TEXT	2027	
LETTERA	*	2009
FOUND	2018	
NEXT	200F	
LENGTH	*	200F
DONE	2024	
TOTAL	*	2018 → 201E
ONE	*	201B

Watch for

1. **Orange** code = a hole (address 0000 written).
2. SYMTAB's **waiting** column: the linked list of holes.
3. **JLT LOOP**: a backward reference is complete at once.
4. When **ZERO** is defined, two holes are filled together.
5. At END no symbol is waiting: TOTAL = 3.

TEXT is defined at 2027. Go back to every waiting place (2006) and fill in 2027.

Click the video to play or pause. About 75 seconds, no sound.

The forward references of COUNT, and their patches

Symbol	Used (holes) at	Defined at	Patched code	Patch as a T record
FOUND	200C	2018	302018	T00200D022018
DONE	2015	2024	3C2024	T002016022024
TEXT	2006	2027	50A027	T00200702A027
LENGTH	200F	202D	2C202D	T00201002202D
LETTERA	2009	2030	282030	T00200A022030
ZERO	2000, 2003	2033	042033, 002033	T002001022033 T002004022033
TOTAL	2018, 201E	2036	002036, 0C2036	T002019022036 T00201F022036
ONE	201B	209D	18209D	T00201C02209D

Read a patch: T00200D022018 = "at 200D, write 2 bytes: 20 18". It overwrites the 0000 left in JEQ FOUND (302018 at 200C). The loader applies records in order, so the patch wins.

When two passes are not enough: multi-pass assemblers

● ● ● multipass.asm

```
MULTI    START    0
HALF     EQU      TLEN/2
TLEN     EQU      TEXTEND-TEXT
FIRST    LDA      #HALF
          J        FIRST
TEXT     BYTE     C 'BANANA '
TEXTEND  EQU      *
```

● ● ● our two-pass assembler (real)

```
line 2: undefined symbol TLEN
(EQU needs every symbol defined earlier)
```

SAME AS C

C does the same

```
enum { HALF = TLEN / 2 }; first → gcc: 'TLEN'
undeclared
```

Why Pass 1 is stuck

EQU must have a value in Pass 1, because it may change LOCCTR-dependent things later. HALF needs TLEN, TLEN needs TEXTEND: both still unknown.

The multi-pass answer

Store HALF = TLEN/2 with the list of symbols it waits for. When TEXTEND (000C) is defined, TLEN = 000C - 0006 = 6 becomes known, then HALF = 3. Each new value can unlock others.

Result

LDA #HALF → 010003. Reordered so a two-pass assembler can do it, our assembler gives exactly this.

Book: Beck §2.4.2 (multi-pass assemblers)

One, two or many passes?

	One-pass	Two-pass	Multi-pass
Forward jumps and data	patched later (holes)	yes	yes
EQU that uses a later symbol	no	no	yes
Reads the source	once	twice (or once + stored copy)	as often as needed
Extra memory	lists of holes	intermediate file	dependency lists
Typical use	load-and-go, teaching, fast test runs	most assemblers (Beck's SIC/XE, our lab)	rare: complex symbol definitions

C link: a C compiler reads your file once, which is why C makes **you** declare before use. The assembler takes on that work itself.

Book: Beck §2.4

LAB 3.3

EQU and literals in your assembler

Version 2 of your C assembler: two new tables and one new directive.

Lab 3.3 tasks, step by step

1

EQU with a number

MAXLEN EQU 6: put MAXLEN into SYMTAB with value 6. Its LOC column shows 0006.

2

EQU with * and A-B

TEXTEND EQU *, TLEN EQU TEXTEND-TEXT. Write eval(): number, *, SYMBOL, SYMBOL-SYMBOL, SYMBOL+number.

3

LITAB

struct lit { char text[40]; int addr; int placed; }; Add a literal the first time it is seen.

4

LTORG and END

Give every waiting literal an address and output it like a BYTE line marked *.

5

Pass 2

An operand starting with = takes its address from LITAB. Test: countlit.asm must match countlit_expected.obj.

6

Break it

Put TLEN EQU TEXTEND-TEXT before TEXT. Your assembler must stop with an error, not print rubbish.

Submit: sicasm_v2.c, the listing and object program of countlit.asm, and the output of task 6.

Checkpoint: test yourself

Q1

COMP #65 and COMP =X '000041': what is the difference?

→ # puts 65 in the instruction; = makes a word in memory and uses its address

Q2

Where do literals go if there is no LTORG?

→ after END, at the end of the program

Q3

TEXT = 0033, TEXTEND = 0039. Is TEXTEND-TEXT absolute or relative? Value?

→ absolute, 6

Q4

Which C keyword is closest to EXTREF?

→ extern

Q5

Why can a two-pass assembler handle LDA #HALF before HALF, but not HALF EQU TLEN/2 before TLEN?

→ an EQU needs its value in Pass 1; an operand only in Pass 2

Key ideas from today



A literal is a hidden, unnamed constant, like a C string literal. LORG says where to put the pool.



EQU names a value. label – label is absolute; label + label is an error, like C pointers.



Program blocks: one LOCCTR per block, then the blocks are stacked: C's .text, .data, .bss.



Control sections are assembled apart, like .c files. D, R and M+symbol records tell the linker what to join.



One-pass: leave holes and patch them. Multi-pass: resolve symbol definitions that point forward.



Your C assembler now has OPTAB, SYMTAB, LITAB and an expression evaluator.

Homework 3.3 (due before Lecture 4)

1. Literals

Rewrite `sumn.asm` (Lecture 3.1) so it uses literals instead of `ZERO` and `N`. Assemble it by hand: `LITTAB`, listing, object program.

2. Expressions

With `TEXT = 0033`, `TEXTEND = 0039`, `MAXLEN = 6`: is each legal? Absolute or relative? `TEXTEND - TEXT + MAXLEN`, `TEXT + MAXLEN`, `MAXLEN - TEXT`, `TEXT + TEXTEND - TEXT`.

3. Blocks

Move `BUFFER RESB 4096` in `countx.asm` into its own block with `USE`. Which instructions no longer need format 4 or base-relative addressing?

4. One pass

Run the one-pass method by hand on `sumn.asm`. List every hole and the patch T record for each.

5. C

Write `main.c` and `counta.c` (slide 13) yourself. Run `nm` and `readelf -r` on both `.o` files and match each line to a D, R or M record of `countc.obj`.

Next lecture and reading

NEXT: LECTURE 4

Macro processors

Teaching the assembler new instructions: macro definition and expansion (NAMTAB, DEFTAB, ARGTAB).

The C preprocessor and `#define` as a real macro processor: `gcc -E`.

Main text

L. L. Beck, **System Software**, 3rd ed. Today: §2.3 and §2.4. Read §4.1 before Lecture 4.

C side

CS:APP, 3rd ed., §7.5–7.6 (symbols and symbol resolution): EXTDEF/EXTREF in ELF.

Tools

`nm`, `readelf -r`, `size` on the lab PC: try them on any `.o` file.

Lab files

`lab3_files.zip`: `countlit.asm`, `countl.asm`, `countb.asm`, `countc.asm`, `blocks.c`, `main.c`, `counta.c`.