

Lecture 5.1

Loaders

absolute, relocating and linking

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

```
./sicload -l 4000 main.obj counta.obj  
data.obj
```

Load map (ESTAB)

Section	Symbol	Address	Length
MAIN		004000	000011
COUNTA		004011	000028
DATA		004039	00000C
	TEXT	004039	
	LEN	00403F	
	TOTAL	004042	

What you will learn today



Explain the three jobs of a loader: loading, relocation and linking.



Trace an absolute loader and a bootstrap loader byte by byte.



Relocate a program with M records and with bit masks.



Run the two-pass linking loader with ESTAB, and see gcc and ld do the same.

Today's plan (60 min)

0-5

Recap of Lecture 4

5-15

Absolute and bootstrap loaders

15-30

Relocation

30-50

Linking loader

50-60

C side, checkpoint, homework

Writing the loader in C is the lab (TODO 1-3 today, TODO 4-5 in 5.2).

Quick recap of Lecture 4

Q1

After macro expansion, who works next?

The assembler. It writes an object program: H, T, M, E records (and D, R for control sections). Today we read those records back.

Q2

What did an M record mean in Lecture 3?

"This address field is relative to the start of the program; fix it when you know where the program is." Today the loader does that fix.

Q3

What do EXTDEF and EXTREF do?

EXTDEF exports a name, EXTREF imports one. Like a global variable in one C file and `extern` in another. Today the linking loader joins them.

PART 1

What a loader does

From an object program on disk to bytes in memory that the CPU can run.

Three jobs: loading, relocation, linking



Loading

Copy the bytes of the T records into memory, then jump to the address in the E record.

absolute loader



Relocation

The program was assembled for address 0 but goes to 5000: fix every field that holds an address.

relocating loader



Linking

MAIN uses COUNTA from another file: find where COUNTA ended up and put that address into MAIN.

linking loader

C link: what happens when you type `./count`

ld (the linker) already did the linking when you ran `gcc`: it joined `main.o`, `counta.o` and `data.o` into one file. When you run it, the **kernel** loads the program's pieces into memory and **ld.so** loads and links the shared libraries (`printf` is in `libc.so.6`). Same three jobs, split between three programs.

Book: Beck §3.1 (basic loader functions)

What the loader reads: the six record types

Record	Holds	Needed by	C / Linux equivalent
H	program name, start address, length	every loader	the header of the .o / executable file
T	bytes of code and data, with their address	every loader	contents of .text and .data
E	address of the first instruction	every loader	the entry point (readelf -h: _start)
M	an address field that must be fixed	relocating, linking	relocation entries (readelf -r)
D	names this file exports (EXTDEF)	linking	defined symbols (nm: T, D, B)
R	names this file needs (EXTREF)	linking	undefined symbols (nm: U)

One difference: our object programs are **text** (two hex characters per byte) so you can read them. A real .o file is **binary**, and you need `nm`, `readelf` or `objdump` to read it. The information inside is the same.

Book: Beck §2.3.5 (D and R records), §3.1

PART 2

The absolute loader and the bootstrap loader

No relocation, no linking: put the bytes exactly where the records say.

The absolute loader: about ten lines of logic

count.obj (Lab 3 assembler, START 2000)

```
HCOUNT 0020000000A0
T0020001E04203300203350A0272820303020182C202D3820033...
T00201E1B0C20363C200F3C202442414E414E410000060000410...
T00209D03000001
E002000
```

1 Read **H**: name COUNT, start 2000, length A0. Check it fits in memory.

2 For each **T**: two hex characters → one byte; store at the address in columns 2–7.

3 At **E**: jump to 002000. The program runs.

the heart of it in C (sicload.c)

```
int start = hexn(r + 1, 6);
int len   = hexn(r + 7, 2);
for (int i = 0; i < len; i++)
    mem[start + i] = hexn(r + 9 + 2*i, 2);
```

SAME AS C

Like memcpy in a loop

A T record is `memcpy(mem + start, bytes, len)`.
The loader does not understand the program; it only copies bytes. Code and data look the same to it.

Book: Beck §3.1.1 (design of an absolute loader)

Memory after the absolute load

```
./sicload count.obj (real)
$ ./sicload count.obj
002000 04203300 203350A0 27282030 3020182C
002010 202D3820 033C2024 00203618 209D0C20
002020 363C200F 3C202442 414E414E 41000006
002030 00004100 00000000 00xxxxxx xxxxxxxx
002040 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
002050 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
002060 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
002070 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
002080 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
002090 xxxxxxxx xxxxxxxx xxxxxxxx xx000001
with -x: Ran 51 instructions, A=000003
```

Each line = 16 bytes

The address, then 4 groups of 4 bytes. 04 2033 at 2000 is LDX ZERO (opcode 04, address 2033).

xx = never written

BUFFER RESB 100 has no T record, so the loader never touches 2039–209C. Its contents are whatever was there before.

C link: .bss

int total; in data.c is not stored in the file either (nm: B). The OS fills it with zeros; SIC does not promise that.

The weak point: the address 2000 is fixed when you **assemble**. A second program that also wants 2000 cannot be in memory at the same time.

Who loads the loader? The bootstrap loader

boot.asm (our version of Beck's idea)

```
FIRST  CLEAR  A
      LDX    #128      . X = 80 hex
LOOP   JSUB   GETC      . 1st hex digit
      RMO    A,S
      SHIFTL S,4      . S = digit*16
      JSUB   GETC      . 2nd hex digit
      ADDR   S,A      . A = one byte
      STCH   0,X      . store at X
      TIXR   X,X      . X = X + 1
      J      LOOP
GETC   TD     INPUT      . wait for device
      JEQ    GETC
      RD     INPUT
      COMP   #4      . end of input?
      JEQ    128      . yes: go to 80
      ... '0'..'9','A'..'F' → 0..15 ...
```

The chicken-and-egg problem

A loader is a program, so something must load it. The bootstrap is so small that hardware (or a ROM) puts it at a **fixed** address and starts it.

It is an absolute loader

No H, T or E records: just a stream of hex characters from device F1, stored from address 80 upward.

End of input = go

Character X'04' (end of transmission) makes it jump to 80. Whatever was loaded there now runs.

Book: Beck §3.1.2 (a simple bootstrap loader), §3.4.3

Booting the count program (real simulator run)

● ● ● device F1: 84 hex characters, then X'04'

```
05000053A01B2900413320092D00063B2FF13F2FFD
03200F1900010F20093F2FEB42414E414E41000000
<04>
```

42

bytes loaded
(84 characters)

1286

bootstrap instructions
before the jump

3

TOTAL after
count runs

● ● ● memory from 80 after boot and run

```
000080 05000053 A01B2900 41332009 2D00063B
000090 2FF13F2F FD03200F 1900010F 20093F2F
0000A0 EB42414E 414E4100 0003xxxx xxxxxxxx
```

● ● ● a real PC: boot.s → boot.bin (nasm)

```
$ od -A x -t x1z boot.bin | tail -3
*
0001f0 00 00 00 ... 00 00 55 aa
000200
```

Same idea on your PC: the firmware reads the first 512-byte sector of the disk to address 7C00 and jumps there, if the sector ends with the bytes **55 AA**. That tiny program loads a bigger loader (like GRUB), which loads the kernel. Each stage loads the next: "pulling yourself up by your bootstraps".

PART 3

Relocation: load the program anywhere

Two methods: modification records (SIC/XE) and bit masks (SIC).

The problem: addresses inside the code

Loc	Source	Object code	Address inside
0000	FIRST LDX ZERO	040033	0033
0003	LOOP LDA ZERO	000033	0033
0006	LDCH TEXT,X	508027	8027
0009	COMP LETTERA	280030	0030
000C	JEQ FOUND	300018	0018

count0.asm is assembled with **START 0**. LDA ZERO became 000033: "read the word at 0033".

Load it at 5000 without changes: ZERO is now at 5033, but the instruction still reads 0033: someone else's memory. Every address field must get **+5000**; the opcodes and the data must not.

```
● ● ● where.c: PIE (default), run twice (real)
```

```
$ ./where
main 0x55f030bf3199
global 0x55f030bf6010
$ ./where
main 0x55f50b9af199
global 0x55f50b9b2010
```

```
● ● ● gcc -no-pie, run twice (real)
```

```
main 0x401186 (both runs)
global 0x404028 (both runs)
```

Only the base moves (...3199 and ...6010 stay): a relocating loader at work. -no-pie is an absolute program.

Book: Beck §3.2.1 (relocation)

Method 1: modification records

```
count0.obj (sicas -m, Lab 3)
```

```
HCOUNT 00000000003C
```

```
T0000001E0400330000335080272...
```

```
T00001E1E0C00363C000F3C00244...
```

```
M00000104
```

```
M00000404
```

```
M00000704
```

```
M00000A04
```

```
M00000D04
```

```
M00001004
```

```
... 13 M records in all
```

```
E000000
```

1 **delta** = PROGADDR - start in H = 5000 - 0 = 5000

2 T: load the bytes at their address + **delta**

3 M M00000104: the 4 half-bytes at 0001 + **delta** → add delta to them

4 E: jump to entry + **delta** = 5000

```
./sicload -v -r 5000 count0.obj (real, first 3 M records)
```

```
M record: at 005001 0033 -> 5033
```

```
M record: at 005004 0033 -> 5033
```

```
M record: at 005007 8027 -> D027
```

Look at 8027 → D027: LDCH TEXT, X has the **x bit** set (8000). Adding 5000 leaves it set: 8027 + 5000 = D027.

Book: Beck §3.2.1 (relocation by modification records)

Loaded at 5000, and it still counts the A's

```
./sicload -x -r 5000 count0.obj (real)

Memory after loading:
005000 04503300 503350D0 27285030 3050182C
005010 502D3850 033C5024 00503618 50390C50
005020 363C500F 3C502442 414E414E 41000006
005030 00004100 00000000 00000001 xxxxxxxx
Ran 51 instructions, stopped at 005024: A=000003 X=000006
L=000000
Memory after running:
005030 00004100 00000000 03000001 xxxxxxxx
```

Instruction	assembled	at 5000
LDX ZERO	040033	045033
LDCH TEXT,X	508027	50D027
J DONE	3C0024	3C5024
WORD 65	000041	000041

Opcodes and constants did **not** change.

SAME AS C

Same as C pointers

An address is "base + offset". Relocation keeps every offset and changes the base, like moving an array: `&a[i]` changes, `i` does not.

DIFFERENT FROM C

Different in a PIE

x86-64 code mostly uses **RIP-relative** addresses (like SIC/XE's PC-relative), so the code needs almost no fixing at load time.

SIC/XE: which fields need an M record?

Kind of field	Example (real code)	Relocate?	Why
PC-relative	JLT LOOP 3B2FEF	no	a distance: same at any address
base-relative	LDA BUF (b=1)	no	a distance from B; LDB sets B at run time
immediate constant	COMP #65 290041	no	a number, not an address
format 4, own label	+JSUB RDREC	yes, 05	a full 20-bit address
format 4, external	+JSUB COUNTA 4B100000	yes, 05 +COUNTA	address known only after linking
WORD with a label	PTR WORD TEXT	yes, 06	a whole word holding an address

Result: a SIC/XE program needs few M records, a SIC program needs one for almost every instruction. **C link:** `readelf -r main.o` lists 5 relocations in `.text` (R_X86_64_PC32 / PLT32): mostly references to **other files**, which no assembler could know.

Book: Beck §3.2.1, §2.2.2 (program relocation)

Method 2: a relocation bit mask in each T record

count0 with bit masks (reference assembler)

```
T0000001E FFC 0400330000335080272800303000182C002D3800033C0024000036180039
T00001E1E E00 0C00363C000F3C002442414E414E41000000600004100000000000000001
```

1	1	1	1	1	1	1	1	1	1	0	0
LDX (1)	LDA (1)	LDCH (1)	COMP (1)	JEQ (1)	TIX (1)	JLT (1)	J (1)	LDA (1)	ADD (1)	- (1)	- (1)

FFC = 1111 1111 1100: ten instructions, all with an address

1	1	1	0	0	0	0	0	0	0	0	0
STA (1)	J (1)	J (1)	BAN (1)	ANA (1)	6 (1)	65 (1)	0 (1)	0 (1)	1 (1)	- (1)	- (1)

E00 = 1110 0000 0000: three instructions, then data

One bit per 3-byte word

Bit 1: add the load address to this word. Bit 0: leave it. 12 bits fit in 3 hex characters, so at most 12 words (Beck uses 10) per T record.

Alignment matters

A RESB gap or a BYTE item that is not a multiple of 3 breaks the word grid: the assembler starts a new T record there.

Book: Beck §3.2.1 (relocation by bit mask)

M records or bit masks?

	Modification records	Bit mask
Unit	any field: 05 or 06 half-bytes, at any address	whole 3-byte words only
Cost for count0	13 records × 9 characters = 117	2 masks × 3 characters = 6
Good when	few fields need fixing (SIC/XE: PC-relative code)	almost every word needs fixing (SIC: direct addresses)
Can say +SYMBOL	yes: the linking loader needs it	no: only "add the load address"
Real-world cousin	ELF relocation entries (readelf -r)	old formats; today's systems use entries

Rule of thumb: a list of exceptions is short when exceptions are rare (M records); a bitmap is short when the property is common (bit mask). You met the same choice in C between a list of indexes and an array of flags.

PART 4

The linking loader

Three files, assembled separately, joined in memory with one table: ESTAB.

One program in three files, in SIC/XE and in C

main.asm

```
EXTREF COUNTA,TOTAL,TEXT
+JSUB COUNTA
+STA TOTAL
PTR WORD TEXT
DIST WORD TOTAL-TEXT
```

counta.asm

```
EXTREF TEXT,LEN
+LDT LEN
+LDCH TEXT,X
COMP #65 ...
RSUB . count in A
```

data.asm

```
EXTDEF TEXT,LEN,TOTAL
TEXT BYTE C'BANANA'
LEN WORD 6
TOTAL WORD 0
```

main.c (demo folder)

```
extern int total;
int counta(void);
total = counta();
```

counta.c (demo folder)

```
extern char text[];
extern int len;
int counta(void) {...}
```

data.c (demo folder)

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

Same: EXTREF = extern, EXTDEF = a global (non-static) name. Each file is assembled / compiled **alone**, so +JSUB COUNTA and call counta both get address 0 plus a note for later.

Book: Beck §3.2.2 (program linking)

What the assembler leaves for the linking loader

main.obj (real)

```
HMAIN 000000000011
RCOUNTATOTAL TEXT
T000000114B1000000F1000003F2FFD000000000000
M00000105+COUNTA
M00000505+TOTAL
M00000B06+TEXT
M00000E06+TOTAL
M00000E06-TEXT
E000000
```

nm main.o ; nm data.o (real)

```
U counta
0000000000000000 T main
U printf
U total

--- data.o
0000000000000008 D len
0000000000000000 D text
0000000000000000 B total
```

data.obj (real)

```
HDATA 000000000000C
DTEXT 000000LEN 000006TOTAL 000009
T0000000C42414E414E41000006000000
E
```

4B100000: +JSUB with address 0. M00000105+COUNTA says: add COUNTA's address to the 5 half-bytes at 0001.

M00000E06+TOTAL and M00000E06-TEXT: DIST = TOTAL - TEXT.
Two M records on one word.

The linking loader's data: ESTAB and three addresses

ESTAB

External Symbol TABLE: every section name and every D-record name, with its **real** address. Like the assembler's SYMTAB, but for the whole program.

PROGADDR

Where the whole program starts. Given by the operating system (here 4000).

CSADDR

Where the current control section starts. First = PROGADDR; next = previous CSADDR + previous length.

CSLTH

Length of the current section, from its H record. Used to compute the next CSADDR.

C link: `ld` builds the same kind of table from the symbol tables of all `.o` files. Ask it to print it: `gcc -no-pie main.o counta.o data.o -Wl, -Map=count.map` writes a **load map** with every section and symbol address (two slides ahead).

Book: Beck §3.2.3 (algorithm and data structures for a linking loader)

Pass 1: give every section an address, fill ESTAB

1 CSADDR = PROGADDR = 4000

2 H record: enter the section name with CSADDR and CSLTH

3 D record: enter each name with CSADDR + its address

4 next section: CSADDR = CSADDR + CSLTH

5 Same name twice? Error: duplicate external symbol

Section	Symbol	Address	Length	How
MAIN		4000	11	PROGADDR
COUNTA		4011	28	4000 + 11
DATA		4039	C	4011 + 28
	TEXT	4039		4039 + 0
	LEN	403F		4039 + 6
	TOTAL	4042		4039 + 9

Pass 1 loads nothing. It reads only H and D records. R, T and M wait for Pass 2. The result is the **load map** that `./sicontrol -l` prints first; it matches this table exactly.

Book: Beck §3.2.3 (Pass 1)

Pass 2: load T records, apply M records with ESTAB

1 H: CSADDR = this section's address from ESTAB

2 T: store the bytes at CSADDR + address

3 M: look up the name in ESTAB; add or subtract its address

4 E of the main section: remember the entry point

Read row 1: at 4001 the instruction 4B100000 (+JSUB, address 0). ESTAB says COUNTA = 4011. Add it: **104011**. The flag half-byte 1 (e = 1) is outside the 5 half-bytes, so it stays.

M record says	at	before	after
+COUNTA	004001	100000	104011
+TOTAL	004005	100000	104042
+TEXT	00400B	000000	004039
+TOTAL	00400E	000000	004042
-TEXT	00400E	004042	000009
+LEN	004016	100000	10403F
+TEXT	00401C	900000	904039

-TEXT comes right after **+TOTAL** on the same word: $004042 - 4039 = 000009$, the distance from TEXT to TOTAL. It does not depend on PROGADDR.

Book: Beck §3.2.3 (Pass 2)

Watch the linking loader work

Linking loader: Pass 1 builds ESTAB, Pass 2 loads and fixes

PASS 1PASS 2RUN

```
main.obj
HMAIN 0000000011
RCOUNTATOTAL TEXT
T00000114B100000F100003F2FFD00000000000
M00000105+COUNTA
M00000505+TOTAL
M00000B06+TEXT
M00000E06+TOTAL
M00000E06-TEXT
E000000

counta.obj
HCOUNTA0000000028
RTEXT LEN
T0000001EB410B44077100000B4005390000029004133200AB8503B2FEFAC404F0000
T00001E0AAC40190001AC043F2FEC
M00000505+LEN
M00000B05+TEXT
E

data.obj
HDATA 0000000000C
DTEXT 00000LEN 000006TOTAL 000009
T0000000C42414E414E41000006000000
E
```

ESTAB (external symbol table)

section	symbol	address	length
MAIN		004000	000011
COUNTA		004011	000028
DATA		004039	00000C
	TEXT	004039	
	LEN	00403F	
	TOTAL	004042	

CSADDR = 4000

memory

004000	4B	10	40	11	0F	10	40	42	3F	2F	FD	00	00	00	00
004010	00	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx
004020	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx
004030	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx
004040	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

M00000505+TOTAL: at 4005, the address field of a format 4 instruction (5 half-bytes). Look up TOTAL in ESTAB: 4042. Add it: 100000 becomes 104042.

Watch for

1. In **PASS 1** memory stays empty: only ESTAB grows.
2. Each **CSADDR** = previous one + previous length.
3. In **PASS 2** each T record fills orange bytes.
4. Each M record changes only a few bytes (teal).
5. **RUN**: TOTAL becomes 03.

Click the video to play or pause. About 1¾ minutes, no sound.

The linked program in memory, before and after running

after loading (./sicload -l 4000 ...)

```
004000 4B104011 0F104042 3F2FFD00 40390000
004010 09B410B4 40771040 3FB40053 90403929
004020 00413320 0AB8503B 2FEFAC40 4F0000AC
004030 40190001 AC043F2F EC42414E 414E4100
004040 00060000 00xxxxxx xxxxxxxx xxxxxxxx
```

4000: 4B104011

+JSUB 4011: the call now goes to COUNTA.

400B: 004039

PTR = the address of TEXT (like `char *ptr = text;`).

after running (56 instructions)

```
004000 4B104011 0F104042 3F2FFD00 40390000
004010 09B410B4 40771040 3FB40053 90403929
004020 00413320 0AB8503B 2FEFAC40 4F0000AC
004030 40190001 AC043F2F EC42414E 414E4100
004040 00060000 03xxxxxx xxxxxxxx xxxxxxxx
```

400E: 000009

DIST = TOTAL - TEXT = 9.

4042: 000003

TOTAL after the run: 3 A's in BANANA.

Memory past 4044 is xx: the program is exactly 45 hex bytes (11 + 28 + C).

gcc and ld do exactly this (real output)

```
objdump -d main.o (before linking)

8: e8 00 00 00 00 call d <main+0xd>
d: 89 05 00 00 00 00 mov %eax,0x0(%rip)
readelf -r: R_X86_64_PLT32 counta - 4
```

```
gcc -no-pie -o count main.o counta.o data.o

40113e: e8 29 00 00 00 call 40116c <counta>
401143: 89 05 df 2e 00 00 mov %eax,... # 404028 <total>
```

The arithmetic: a call stores the distance from the **next** instruction:
 $40116C - 401143 = 29$. The same as SIC/XE PC-relative: target - PC.

```
count.map (-Wl,-Map, real excerpt)

.text address size file
.text 0x401136 0x36 main.o
.text 0x40116c 0x44 counta.o
.data 0x404018 0xc data.o
.bss 0x404028 0x4 data.o
symbols:
0x401136 main 0x40116c counta
0x404018 text 0x404020 len
0x404028 total
```

SAME AS C

This is the load map

Sections placed one after another, then every symbol gets base + offset: CSADDR and ESTAB.

Book: Beck §3.2.2–3.2.3

When linking fails: the same two errors everywhere

1. A definition is missing

```
● ● ● our loader  
$ ./sicload -l 4000 main.obj counta.obj  
error: undefined external symbol TOTAL
```

```
● ● ● gcc / ld  
$ gcc -no-pie main.o counta.o  
undefined reference to `total'  
undefined reference to `text'
```

Pass 2 finds +TOTAL in an M record, but no file defined TOTAL. Fix: link data.obj / data.o too.

2. A name is defined twice

```
● ● ● our loader  
$ ./sicload -l 4000 ... data.obj data.obj  
error: duplicate external symbol DATA
```

```
● ● ● gcc / ld  
$ gcc ... data.o data2.o  
multiple definition of `text';  
data.o: first defined here
```

Pass 1 finds the same name in two H or D records. Which address is right? The loader cannot guess.

LAB 5.1

Write the loader in C

sicload.c: absolute and relocating loading today, linking in Lab 5.2.

Lab 5.1 tasks, step by step

1

Build the starter

`gcc -Wall -o sicload sicload_starter.c`
then `./sicload count.obj`. Memory is all `xx`: nothing is loaded yet.

2

TODO 1: load_T

Copy the bytes of a T record into memory. `./sicload count.obj | diff - expected_abs.txt` must print nothing.

3

TODO 2: patch

Add delta to a field of 4, 5 or 6 half-bytes, keeping the other bits. Test by hand: $100000 + 4011 \rightarrow 104011$.

4

TODO 3: M records

One line in `load_one`. Then `./sicload -v -x -r 5000 count0.obj`: TOTAL must become 000003.

5

Try other addresses

Load `count0.obj` at 0, 1234 and 7000 with `-x`. Same result every time? Then your relocation is right.

6

C side

In the demo folder: `nm, readelf -r` and `objdump -d` on `main.o`, then link with `-Wl, -Map` and find the call to `counta`.

Extra: TODO 6 loads bit-mask records (`-b 5000 count0_mask.obj`). Submit: `sicload.c` and the output of task 4.

Checkpoint: test yourself

Q1

Which records does an absolute loader read?

→ H, T and E (M, D, R are ignored)

Q2

count0 is loaded at 3000. LDA ZERO was 000033. Now?

→ 003033

Q3

Why does J LOOP (PC-relative) need no M record?

→ it stores a distance, which does not change

Q4

Mask E00: which words of the record are relocated?

→ the first three (1110 0000 0000)

Q5

DATA has length C and starts at 4039. Where does the next section start?

→ 4045 (4039 + C, in hex)

Key ideas from today



A loader does up to three jobs: loading, relocation, linking.



The bootstrap is a tiny absolute loader at a fixed address that loads the real loader.



Relocation adds the load address to address fields only: chosen by M records or by a bit mask.



Linking loader Pass 1 builds ESTAB from H and D; Pass 2 loads T and applies M with ESTAB.



EXTDEF/EXTREF are C's globals and extern; ld's load map is ESTAB.



Undefined (Pass 2) and duplicate (Pass 1) symbols: the same errors in sicload and ld.

Homework 5.1 (due before Lecture 5.2)

1. Absolute load

Write the memory dump (16 bytes per line, from 80) that an absolute loader makes from `countboot.obj` (lab files). Mark bytes never written as `xx`.

2. Relocate

`count0.obj` is loaded at 7000. Give the new values of the first three M-record fields, and explain why the third one starts with F.

3. Find the bug

`hw51.asm` has `PTR WORD TOTAL`. Lab 3's `sicasm -m` writes M records only for instructions. What goes wrong at 3000, and which M record is missing? Also write its bit mask.

4. Link by hand

Link `data.obj`, `counta.obj`, `main.obj` (in this order) at `PROGADDR 6000`. Give `ESTAB`, the entry point, and the words at `PTR` and `DIST`.

5. C side

Link in another order: `gcc -no-pie -o c3 counta.o main.o data.o`. Which addresses in `nm c3` change? What are the 4 bytes after `e8` in `main` now, and why do they end in `ff ff`?

Next lecture and reading

NEXT: LECTURE 5.2

Loader design options and dynamic linking

Library search and loader options, linkage editors vs linking loaders, and dynamic linking: shared libraries, ld.so, and the PLT/GOT trick behind every call to printf.

Main text

L. L. Beck, **System Software**, 3rd ed. Today: §3.1–3.2. Read §3.3–3.4 before Lecture 5.2.

C side

R. E. Bryant & D. R. O'Hallaron, **Computer Systems: A Programmer's Perspective**, 3rd ed., Chapter 7 (Linking).

Tools

nm, readelf -r, objdump -d, gcc -Wl, -Map: all on the lab PC, all offline.

Lab files

lab5_files.zip: starter, object programs, expected outputs, C demos and run_all.sh.