

Lecture 4

Macro Processors from SIC macros to the C preprocessor

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

● ● ● a macro and its expansion

```
INCR      MACRO    &VAR  
          LDA      &VAR  
          ADD      ONE  
          STA      &VAR  
          MEND
```

```
... FOUND INCR TOTAL becomes  
FOUND    LDA  TOTAL
```

What you will learn today



Write a macro definition and its calls, and expand them by hand.



Explain the one-pass algorithm with NAMTAB, DEFTAB and ARGTAB.



Use concatenation, unique labels, keyword parameters, IF and WHILE.



Compare SIC macros with the C preprocessor, including its classic traps.

Today's plan (60 min)

0-5

Recap of Lecture 3

5-15

What a macro is

15-30

The algorithm

30-45

Extra features

45-55

Design options, C traps

55-60

Checkpoint, homework

Writing the macro processor in C (slides 28-29) is the lab.

Quick recap of Lecture 3

Q1

What does EQU do?

It gives a name to a value, like C's `#define` for a number. Today we go further: a name for a whole block of lines.

Q2

Why did the assembler need two passes?

Forward references. Today's macro processor needs only one, because a macro must be defined before it is used.

Q3

What is a literal?

A constant the assembler creates for you. Today, another tool that writes source lines for you: the macro processor.

PART 1

What a macro is

A named block of source lines, copied in wherever its name appears.

A macro processor runs before the assembler



C, Lecture 1:

SAME AS C

You have used one already

In Lecture 1, `gcc -E` turned 7 lines of `hello.c` into 820: that was the C preprocessor pasting `#include` files and expanding macros. It is a macro processor.

DIFFERENT FROM C

It only copies text

A macro processor knows nothing about instructions or addresses. It replaces names by lines of text. The assembler (or compiler) sees only the result.

Book: Beck §4.1

Defining and calling a macro

the two definitions (maccnt.asm)

```
CHKCH    MACRO    &CH,&HIT
          LDA      ZERO
          LDCH     TEXT,X
          COMP     &CH
          JEQ      &HIT
          MEND
INCR      MACRO    &VAR
          LDA      &VAR
          ADD      ONE
          STA      &VAR
          MEND
```

two calls

```
LOOP     CHKCH    LETTERA,FOUND
FOUND    INCR      TOTAL
```

Prototype: CHKCH MACRO &CH,&HIT: the macro's name and its parameters. Parameter names start with &.

Body: The lines between MACRO and MEND. They are **stored, not assembled**.

MEND: Marks the end of the definition.

Call (macro instruction): The macro's name used like an opcode, with arguments: CHKCH LETTERA, FOUND.

DIFFERENT FROM C

Not C syntax

& marks a parameter; C uses plain names.

the C version

```
#define CHKCH(ch, hit) \
    if (text[i] == (ch)) \
        goto hit;

CHKCH('A', found);
```

SAME AS C

Same parts as C

#define NAME(params) is the prototype; the rest of the line is the body; the end of the line (or \) is MEND.

Book: Beck §4.1.1 (macro definition and expansion)

Expansion: the calls are replaced by the body

● ● ● source (after the definitions)

```
FIRST  LDX  ZERO
LOOP   CHKCH LETTERA, FOUND
NEXT   TIX  LENGTH
        JLT  LOOP
DONE    J    DONE
FOUND   INCR TOTAL
        J    NEXT
```

Assembled and run: TOTAL = 3, the same as COUNT in Lecture 3.1.

● ● ● expanded source (real output)

```
FIRST  LDX  ZERO
.LOOP   CHKCH LETTERA, FOUND
LOOP    LDA  ZERO
        LDCH TEXT, X
        COMP LETTERA
        JEQ  FOUND
NEXT     TIX  LENGTH
        JLT  LOOP
DONE     J    DONE
.FOUND   INCR TOTAL
FOUND    LDA  TOTAL
        ADD  ONE
        STA  TOTAL
        J    NEXT
```

The call becomes a comment
.LOOP CHKCH LETTERA, FOUND
stays in the listing so you can see where the lines came from.

Arguments replace parameters
&CH → LETTERA, &HIT → FOUND.
Pure text: the processor does not check that LETTERA is a word.

The label moves to line 1
LOOP was on the call; now it is on the first generated line, so JLT LOOP still works.

Macro or subroutine? Real numbers

Count A's and N's in BANANA	Macros (CHKCH, INCR)	Subroutines (JSUB ... RSUB)
Instructions executed	81	139
Bytes of code	60	75
Bytes each extra use of CHKCH adds	12 (the whole body)	9 (LDA, JSUB, JEQ)
Passing arguments	free: text is copied in	extra code: put the argument in A, save it
Smaller from ...	1 to 6 uses	the 7th use on

Rule of thumb: a short piece used often and needing speed → macro. A long piece used in many places → subroutine.

```
gcc -S sumrun.c (real)
```

```
movl $7, %edi
call square
movl %eax, %edx
movl $49, %esi
# the macro became 49:
# no call at all
```

SAME AS C

C: macro vs function

A macro is copied in (fast, grows the code). A function is called (one copy, but call + return each time). C's `inline` tries to get both.

Book: Beck §4.1.1

PART 2

How a macro processor works

Three tables and one pass through the source.

Three tables, and how they look in C

NAMTAB

the name table

Macro name → where its lines are in DEFTAB.
Searched for every opcode: is this a macro?

```
macro.c
struct mac {
    char name[16];
    int start, end;
};
struct mac NAMTAB[50];
```

DEFTAB

the definition table

The stored lines of every macro, with
parameters written as positions: ?1, ?2.

```
macro.c
char DEFTAB[500][200];

/* DEFTAB[3] holds
   "COMP ?1"      */
```

ARGTAB

the argument table

The arguments of the call being expanded right
now. Filled at each call, then thrown away.

```
macro.c
char ARGTAB[10][64];
int nargs;

/* ARGTAB[0] = "LETTERA" */
```

Why ?1 instead of &CH? The name &CH only matters inside its own macro. Positions make the substitution simple: "put argument 1 here". C's preprocessor also records each parameter by its position.

Book: Beck §4.1.2 (macro processor algorithm and data structures)

The tables after reading the two definitions

#	DEFTAB line		
0	CHKCH	MACRO	&CH,&HIT
1		LDA	ZERO
2		LDCH	TEXT,X
3		COMP	?1
4		JEQ	?2
5		MEND	
6	INCR	MACRO	&VAR
7		LDA	?1
8		ADD	ONE
9		STA	?1
10		MEND	

NAMTAB name	start	end
CHKCH	0	5
INCR	6	10

Read the orange rows

DEFTAB[3] COMP ?1: the first argument goes here.

DEFTAB[4] JEQ ?2: the second.

DEFTAB[7] and [9] ?1: INCR's only parameter, used twice.

Lines without ? (like LDA ZERO) are copied unchanged at every call.

The one-pass algorithm

1

GETLINE: read the next line, from the source file, or from DEFTAB while a macro is being expanded.

2

PROCESSLINE: look up the opcode in NAMTAB. Found → EXPAND. Opcode is MACRO → DEFINE. Otherwise copy the line to the output.

3

DEFINE: put the name in NAMTAB and the prototype in DEFTAB. Store each body line with &names → ?n, until MEND.

4

EXPAND: fill ARG TAB from the call, write the call as a comment, then GETLINE the body from DEFTAB, replacing ?n by ARG TAB[n].

5

Nested definitions: a counter LEVEL goes up at MACRO and down at MEND, so MEND of an inner definition is not taken as the end.

SAME AS C

Why one pass is enough

A macro must be **defined before it is called**, so DEFTAB is ready when a call arrives. No forward references, unlike labels in Lecture 3.

SAME AS C

C has the same rule

Use a `#define` above its definition and the name is left as it is: the preprocessor reads the file once, top to bottom.

Book: Beck §4.1.2 (one-pass macro processor algorithm)

Expanding LOOP CHKCH LETTERA,FOUND step by step

ARGTAB	value
?1	LETTERA
?2	FOUND

Before line 1: write `.LOOP CHKCH LETTERA,FOUND` as a comment.
Line 1: the call had the label LOOP and this line has none, so LOOP goes here.
After line 4: reaching MEND (DEFTAB[5]) ends the expansion; back to the source.

DEFTAB	stored line	what happens	output line
1	LDA ZERO	no ?: copy	LOOP LDA ZERO
2	LDCH TEXT,X	no ?: copy	LDCH TEXT,X
3	COMP ?1	?1 → LETTERA	COMP LETTERA
4	JEQ ?2	?2 → FOUND	JEQ FOUND

Substitution order: replace ?10 before ?1, or ?10 would become ARGTAB[1] followed by "0". The C lab code counts down from nargs for this reason.

Watch the macro processor work

The macro processor: define, then expand

DEFINEEXPAND

source: maccnt.asm

MACCNT START 2000
CHKCH MACRO &CH,&HIT
LDA ZERO
LDCH TEXT,X
COMP &CH
JEQ &HIT
MEND
INCR MACRO &VAR
LDA &VAR
ADD ONE
STA &VAR
MEND
FIRST LDX ZERO
LOOP CHKCH LETTERA,FOUND
NEXT TIX LENGTH
JLT LOOP
DONE J DONE
FOUND INCR TOTAL
J NEXT
TEXT BYTE C'BANANA'
LENGTH WORD 6
LETTERA WORD 65
ZERO WORD 0
TOTAL WORD 0
ONE WORD 1
END FIRST

DEFTAB

0	CHKCH	MACRO	&CH,&HIT
1		LDA	ZERO
2		LDCH	TEXT,X
3		COMP	?1
4		JEQ	?2
5		MEND	
6	INCR	MACRO	&VAR
7		LDA	?1
8		ADD	ONE
9		STA	?1
10		MEND	

NAMTAB

CHKCH start 0 end 5
INCR start 6 end 10

ARGTAB

?1 = LETTERA ?2 = FOUND

expanded output

MACCNT START 2000
FIRST LDX ZERO
.LOOP CHKCH LETTERA,FOUND
LOOP LDA ZERO
LDCH TEXT,X

DEFTAB[2] has no parameter: copy it as it is.

Watch for

1. In **DEFINE**, DEFTAB fills but the output stays empty.
2. Parameter names turn into ?1, ?2.
3. **MEND** adds a line to NAMTAB.
4. In **EXPAND**, ARGTAB fills at each call.
5. The orange output line is the DEFTAB line with ?n replaced.

PART 3

Machine-independent macro features

Concatenation, unique labels, conditional expansion, loops and keyword parameters.

Concatenation: build names from parameters

SIC/XE macro (macfeat.asm)

```
SHOW      MACRO    &ID,&K
           LDA      CNT&ID
           STA      RES&K
           MEND
. call:    SHOW A,1  expands to:
           LDA      CNTA
           STA      RES1
```

C: ## and # (gcc -E, real)

```
#define COUNTER(id) cnt_ ## id
#define SHOW(x) printf(#x " = %d\n", x)
int COUNTER(a) = 3;
/* becomes */
int cnt_a = 3;
printf("COUNTER(a)" " = %d\n", cnt_a);
```

SAME AS C

Same as C's

CNT&ID glues the argument to fixed text: CNT + A = CNTA. In C you must write ##; in SIC the & already marks where the parameter is.

DIFFERENT FROM C

When text follows

X&ID1 is unclear: parameter &ID1, or &ID then 1? A separator solves it: X&ID->1 → XA1. Our processor removes -> after substitution. (C's #x turns an argument into a string; SIC has no equivalent.)

Book: Beck §4.2.1 (concatenation of macro parameters)

Unique labels: a macro with a loop, called twice

inside CNTCH (macfeat.asm)

```
$LOOP    CLEAR    A  
          LDCH      &TXT,X  
          COMP      #&CH  
          JEQ       $HIT  
          ...  
          JLT       $LOOP
```

Without \$

Two calls → two lines labelled LOOP. Our assembler stops: duplicate symbol LOOP.

With \$

Each call gets its own two letters: \$AA..., \$AB..., \$AC.... The labels can never clash.

expansions (real)

```
. 1st call  
$ABLOOP CLEAR    A  
          JEQ     $ABHIT  
.  
. 2nd call  
$ACLOOP CLEAR    A  
          JEQ     $ACHIT
```

nasm does it too

%%loop: in a nasm macro became .@3.loop: and .@4.loop: (nasm -E, real).

C has no labels in macros

...but the same clash happens with variables: SWAP(t, x) with an inner int t does nothing. Real run: t=9 x=2.

Book: Beck §4.2.2 (generation of unique labels)

Conditional expansion: IF, ELSE, ENDIF

```
● ● ● end of CNTCH
CNTCH  MACRO &TXT,&LEN,&CH,&RES=
      ...
      IF    (&RES NE '')
$DONE  STS   &RES
      ELSE
$DONE  RMO   S,A
      ENDF
```

```
● ● ● CNTCH TEXT,6,65,RES=CNDA
... (13 lines)
$ABDONE STS   CNDA
```

```
● ● ● CNTCH TEXT,6,66
... (13 lines)
$ADDONE RMO   S,A
```

Same macro, different code: the IF is decided **while expanding**, using this call's arguments. The IF lines themselves never reach the assembler.

SAME AS C

Like C's `#if / #ifdef`

`gcc -DDEBUG` prints "debug: starting"; without it the line is not even compiled (real run).

DIFFERENT FROM C

But stronger

C's `#if` cannot test a macro's **arguments**. SIC's IF can: one macro, different code for each call.

Book: Beck §4.2.3 (conditional macro expansion)

Macro-time loops: SET and WHILE

```
● ● ● FILL (macfeat.asm)

FILL      MACRO    &TAB,&N,&VAL=0
&OFF      SET      0
&I        SET      0
          LDA      #&VAL
          WHILE    (&I LT &N)
          STA      &TAB+&OFF
&OFF      SET      &OFF+3
&I        SET      &I+1
          ENDW
          MEND
```

```
● ● ● FILL RES1,3,VAL=7 (real)

LDA      #7
STA      RES1+0
STA      RES1+3
STA      RES1+6
```

How it runs

&I and **&OFF** are **macro-time variables**: they exist only while expanding.

The WHILE body is copied 3 times, with &OFF = 0, 3, 6.

DIFFERENT FROM C

C cannot do this

The C preprocessor has **no loop**. To repeat code you write it out, or use a run-time `for`.

SAME AS C

Like loop unrolling

The result is what an optimising compiler makes from `for (i = 0; i < 3; i++) res[i] = 7;`

Book: Beck §4.2.3 (SET and WHILE)

Keyword parameters: arguments by name, with defaults

Prototype	Call	Meaning
<code>FILL &TAB,&N,&VAL=0</code>	<code>FILL RES1,3</code>	VAL takes its default: 0
<code>FILL &TAB,&N,&VAL=0</code>	<code>FILL RES1,3,VAL=7</code>	VAL given by name: 7
<code>CNTCH ...,&RES=</code>	<code>CNTCH TEXT,6,66</code>	RES empty → IF takes the ELSE path
<code>CNTCH ...,&RES=</code>	<code>CNTCH TEXT,6,65,RES=CNTA</code>	RES = CNTA → STS CNTA

DIFFERENT FROM C

C has no keyword arguments

C macros and functions take arguments **by position only**, and none may be left out. A macro with 8 parameters, most of them usually the same, is hard to call correctly.

SAME AS C

But you have seen the idea

Python's `print(x, end=" ")` passes a keyword argument; `end` has a default value (a newline).

Book: Beck §4.2.4 (keyword macro parameters)

Everything together: macfeat.asm

the main program: 8 lines

```
FIRST  STL      RETADR
        FILL    RES1,3,VAL=7
        CNTCH   TEXT,6,65,RES=CNТА
        CNTCH   TEXT,6,78,RES=СNTN
        CNTCH   TEXT,6,66
        SHOW    A,1
        SHOW    N,2
        J       @RETADR
```

6

macro calls

52

instructions after expansion

151

instructions executed

Word	CNТА	CNTN	RES1	RES2	RES3
After the run	3	2	3	2	7

3 A's and 2 N's in BANANA; RES3 keeps the 7 from FILL. Loaded at 0000 and at 4000: the same results.

The point: 8 readable lines became 52 instructions with no copy-paste mistakes. The macro processor did the boring part; the assembler from Lecture 3 did the rest. This is why real programs in assembly lean heavily on macros.

PART 4

Macro processor design options

Macros inside macros, and where the macro processor lives.

A macro that calls a macro: why one ARGTAB is not enough

● ● ● nested.asm

```
INCR  MACRO  &VAR  (as before)
ADD2  MACRO  &A,&B
      INCR  &B
      INCR  &A
      MEND
FIRST ADD2  TOTALA,TOTALN
```

What went wrong: expanding `INCR &B` put `TOTALN` into `ARGTAB[1]`, over `ADD2`'s own `TOTALA`. When `ADD2` continued with `INCR &A`, ?1 said `TOTALN`. No error message, just a wrong answer.

● ● ● one global ARGTAB (real)

```
FIRST  LDA  TOTALN
      ADD  ONE
      STA  TOTALN
      LDA  TOTALN
      ADD  ONE
      STA  TOTALN
run: TOTALA = 0
     TOTALN = 2
```

● ● ● one ARGTAB per call (real)

```
FIRST  LDA  TOTALN
      ADD  ONE
      STA  TOTALN
      LDA  TOTALA
      ADD  ONE
      STA  TOTALA
run: TOTALA = 1
     TOTALN = 1
```

The fix: keep a **stack** of ARGTABs (or write the processor recursively, so each call has its own local table). This is what C does: every function call gets its own local variables on the stack.

Book: Beck §4.3.1 (recursive macro expansion)

The C preprocessor: nested yes, recursive no

```
rec.c → gcc -E -P (real)
```

```
#define foo    foo + 1
#define A      B + 1
#define B      A * 2
int x = foo;
int y = A;
/* output: */
int x = foo + 1;
int y = A * 2 + 1;
```

Compare with SIC macros

Our processor lets a macro call **another** macro (ADD2 → INCR). A macro that calls **itself** needs IF to stop, or it never ends: the same rule as recursion in C, which needs a base case.

Nested calls work

A uses B: the result of one expansion is scanned again for more macro names. COUNTER(a) inside SHOW(...) on slide 16 worked the same way.

Self-reference stops

Inside its own expansion, foo is not expanded again, so $\text{foo} \rightarrow \text{foo} + 1$ and stops. Otherwise it would never end.

Indirect recursion stops too

$A \rightarrow B + 1 \rightarrow A * 2 + 1$: when A appears again inside its own expansion, it is left alone.

Where does the macro processor live?

Design	How it works	Real example	Trade-off
Separate preprocessor	Runs first, writes an expanded file, then the translator runs	cpp for C; our lab program	simple, works for any translator, but two passes over the text
Line-by-line, inside the translator	Expands one line and hands it straight to Pass 1	most macro assemblers	no intermediate file; faster
Integrated	The translator itself understands macros and their arguments	nasm %macro with %%labels	can check arguments; tied to one language
General-purpose	Not tied to any language: works on any text	m4 (installed on the lab PC)	flexible, but knows nothing about the language

```
m4 count.m4 (real)

define(`GREET', `Hello, $1! You are in $2.')
```

Hello, students! You are in CSE 4105.

C link: gcc hides the separate preprocessor: `gcc -E` shows you its output, and `cpp hello.c` runs it alone.

Book: Beck §4.3.2–4.3.3 (general-purpose macro processors; macro processing within translators)

Three classic C macro traps, all from "just copying text"

1. Missing parentheses

```
● ● ● real gcc output
#define SQUARE(x) x * x
SQUARE(a + 1) /* a = 3 */
/* expands to */
a + 1 * a + 1
result 7, not 16
```

Fix: `((x) * (x))` gives 16. The processor copies text; it does not know about precedence.

2. Side effects twice

```
● ● ● real gcc output
#define MAX(a, b) \
    ((a) > (b) ? (a) : (b))
m = MAX(i++, j); /* i=5, j=2 */
((i++) > (j)) ? (i++) : (j))
m = 6, i = 7
```

`i++` was copied twice, so it runs twice. Never pass `++` or a function call into a macro.

3. Name clash

```
● ● ● real gcc output
#define SWAP(a, b) \
    { int t = a; a = b; b = t; }
SWAP(t, x); /* t=9, x=2 */
{ int t = t; t = x; x = t; }
t=9 x=2: no swap
```

The macro's `t` hides yours: the unique-label problem in C. Fix: an inline function.

LAB 4

Write a macro processor in C

About 200 lines: NAMTAB, DEFTAB, ARGTAB and one loop over the source.

Lab 4 tasks, step by step

1

Build the starter

`gcc -Wall -o macro macro_starter.c` then
`./macro maccnt.asm`. The output still says `COMP &CH`.

2

TODO 1: DEFTAB

Replace each parameter name by `?n` when storing a line. Print `DEFTAB` and compare with slide 11.

3

TODO 2 + 3: ARGTAB

Split the call's operand at commas; replace `?n` by `ARGTAB[n-1]`, counting down.

4

TODO 4: the label

Put the call's label on the first generated line. `diff` with `maccnt_expected.asm` must print nothing.

5

Into the assembler

`./macro maccnt.asm > m.asm` then your Lab 3
`./sicas m.asm`. The object program must match slide 7's run.

6

TODO 5: \$ labels

Make `twice.asm` work: `CNTCH` called twice. Without TODO 5 your assembler reports `duplicate symbol LOOP`.

Submit: macro.c, the expanded twice.asm, and its object program from your Lab 3 assembler.

Mistakes and the messages they produce

No unique labels

```
bash
$ ./macro nodollar.asm >
x.asm
$ ./sicas x.asm
error: duplicate symbol
LOOP
```

The macro was called twice, so LOOP was generated twice. Use \$LOOP and finish TODO 5.

Macro defined twice

```
bash
$ ./macro twodef.asm
error: macro M1 defined
twice
```

Like two C functions with the same name. Rename one of them.

?1 left in the output

```
bash
# output contains
COMP ?1
```

TODO 3 is missing or counts up: ?
10 must be replaced before ?1.

Wrong numbers, no error

```
bash
# nested.asm, one ARG TAB
TOTALA = 0, TOTALN = 2
```

A macro inside a macro overwrote ARG TAB (slide 23). The lab version does not support nesting: say so in your report.

Checkpoint: test yourself

Q1

Why does a macro processor need only one pass?

→ a macro must be defined before it is called

Q2

DEFTAB holds STA ? 1. Call: INCR TOTAL. Output line?

→ STA TOTAL

Q3

A macro with a loop label is called 3 times. What breaks, and what fixes it?

→ duplicate labels; \$ labels (\$AA..., \$AB..., \$AC...)

Q4

#define SQUARE(x) x*x — what is SQUARE(2+3)?

→ 2+3*2+3 = 11, not 25

Q5

Macro or subroutine for a 100-line routine used 20 times?

→ subroutine: 20 copies of 100 lines is too big

Key ideas from today



A macro is a named block of text, copied in at every call, before assembly.



One pass, three tables: NAMTAB (names), DEFTAB (bodies, ?n), ARGTAB (this call's arguments).



Concatenation (CNT&ID), unique labels (\$), keyword parameters (&VAL=0).



IF and WHILE run at expansion time: one macro, different code for each call.



Macros inside macros need one ARGTAB per call, i.e. a stack.



C's preprocessor is a macro processor: its traps all come from pure text copying.

Homework 4 (due before Lecture 5.1)

1. Expand

Expand `twice.asm` (lab files) by hand: `DEFTAB`, `NAMTAB`, `ARGTAB` for each call, and every output line.

2. Write a macro

Write SIC macro `MOVE &FROM,&TO` that copies one word, and `SWAPW &A,&B` that swaps two words using `MOVE`. Expand `SWAPW X1,X2`.

3. Trade-off

`CHKCH` costs 12 bytes per use as a macro, and 9 bytes per use + 18 bytes once as a subroutine. From how many uses is the subroutine smaller? Show the inequality.

4. C traps

Predict, then run: `SQUARE(2 + 3)`, `10 * DOUBLE(2)` with `#define DOUBLE(x) x + x`, and `MAX(x++, y)` with `x = 4`, `y = 3` (print `m` and `x`).

5. Design

Explain in 5–8 sentences how you would let the lab's `macro.c` expand a macro inside a macro. What must become a stack?

Next lecture and reading

NEXT: LECTURE 5.1

Loaders: absolute, relocating, linking

Putting the object program into memory: the absolute loader, relocation with M records and with bit masks, and the linking loader that joins control sections with ESTAB.

The bootstrap loader that starts it all.

Main text

L. L. Beck, **System Software**, 3rd ed. Today: §4.1–4.3. Read §3.1–3.2 before Lecture 5.1.

C side

B. W. Kernighan & D. M. Ritchie, **The C Programming Language**, 2nd ed., §4.11 (the C preprocessor).

Tools

`gcc -E`, `cpp`, `nasm -E`, `m4`: all on the lab PC, all work offline.

Lab files

`lab4_files.zip`: starter, test programs, expected outputs, C/nasm/m4 demos.