C and Linking
LC3 C compiler, lcc; hand linking with assembly code; function call interfaces and protocols.

1. Build lcc, C compiler for LC3.

see src/Makefile for instructions:

% make

Assume working dir is your LC3trunk/.
Assume we already built other tools e.g., lc3as, and installed in /bin.

We also assume gcc is installed.

Build problems?

--- cpp/unix.c and string.h

memmove() redefined in unix.c,
might cause
build problem for compiling unix.c.

1. Possible solution: remove memmove() definition from bin/lcc-1.3/cpp/unix.c. Then, rebuild.

Do not unzip again, that would clobber your change.

2. Try compiling a test file.

```
cd ../run

cp ../bin/fig.16.4.c   f.c

lcc f.c
f.c:24: warning: missing return value
STARTING PASS 1
0 errors found in first pass.
STARTING PASS 2
0 errors found in second pass.
```

If all has gone well, the lcc compiler gives warning (why?). Then lc3as runs ("STARTING..."). This produces 3 output files:

a.asm The compiler's assembly language output.
a.sym The symbol table from assembling a.asm.
a.obj The LC3 load module from assembling a.asm.

3. The a.obj LC3 load module can be loaded and run by one of the LC3 simulators, e.g., PennSim.jar. We will want to convert a.obj to a.bin so we can run it on our LC3 verilog simulation.

C code, f.c, is from Fig. 16.4 in Patt&Patel.

Build/install C compiler, lcc:

--- unzip source to bin/lcc-1.3/
--- configure Makefiles
--- do "make" to compile
--- do "make install" to copy executables to bin/

NB--Do not have path names with spaces.

```
sh configure   --installdir   ~/my work/bin
```

or

```
sh configure   --installdir   "~/my\ work/bin"
```

spaces in path

In bin/lcc-1.3/, we can
- sh configure
- make
- make install

But, configure has a line w/

TOP_DIR = `pwd`

has spaces

```
cd ~
ln -s   "~/my\ work/bin"   LC3trunk-bin
```

edit

TOP_DIR = ~/LC3trunk-bin

Add it to your PATH

```
% PATH= ~/LC3trunk-bin:${PATH}
```

also, ln fixes setting search PATH

This fixes that problem. Now do,

sh configure --installdir ~/LC3trunk-bin

4. Create readmem-readable LC3 exectutable object file

f.obj → f.bin

Converting can be done by hand.
The "grep" command dumps lines matching a regular expression (1st arg). This is just a handy way of seeing the command syntax we've used in src/Makefile.

```
grep "obj2bin " src/Makefile
grep "sed" src/Makefile

   obj2bin <  a.obj  > a.bin

sed '1d' < a.bin > prog.bin
```

--- Linking separate asm sources.
--- Compiler output as .asm
--- Running LC3 code in verilog simulations.
--- Combining C and ASM.
--- call frames, local vars, return values, text-data-stack layout.

%> more f.c

```
#include <stdio.h>
#define MAX_NUMS 10

   ...

int main()
{
  int index;
  int numbers[MAX_NUMS];

  printf("Enter %d numbers.\n", MAX_NUMS);

    ...
```

==> f.c uses operating system's services.

==> HALT and OUT.

==> NOT in f.c's C code!

==> printf()? IS THAT C TOO?

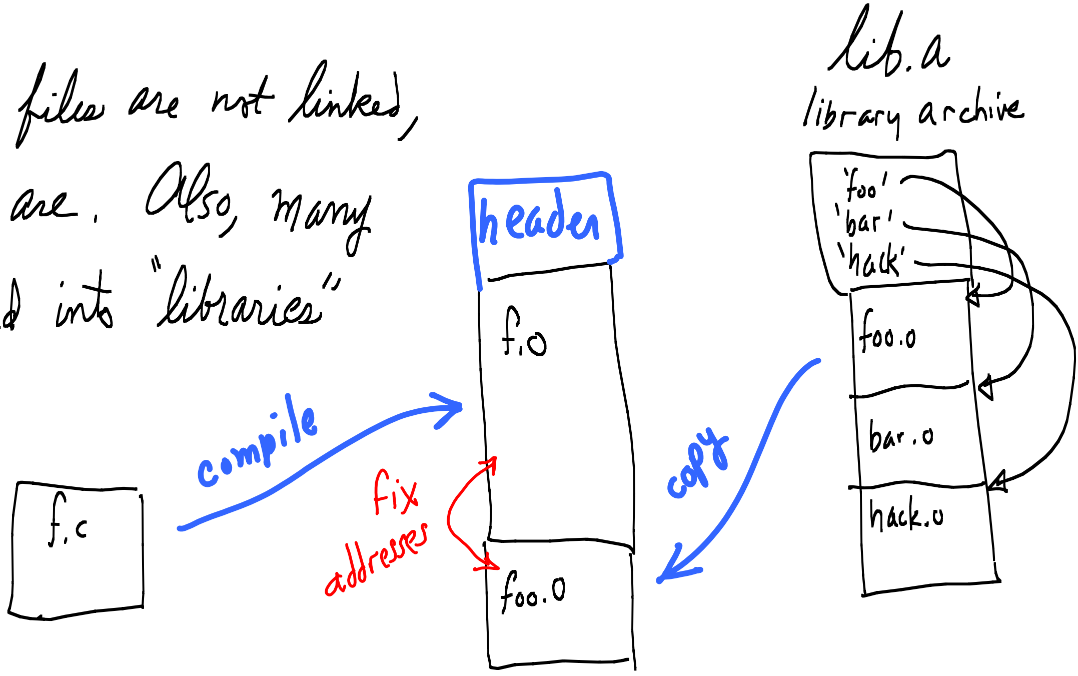
==> printf.asm is linked in.

cd bin/lcc-1.3/lc3lib/

ls

```
getchar.asm    printf.asm
putchar.asm    scanf.asm
stdio.asm      stdio.h
```

Usually, the .asm files are not linked, but the .obj files are. Also, many .obj files are collected into "libraries"



Static Linking:

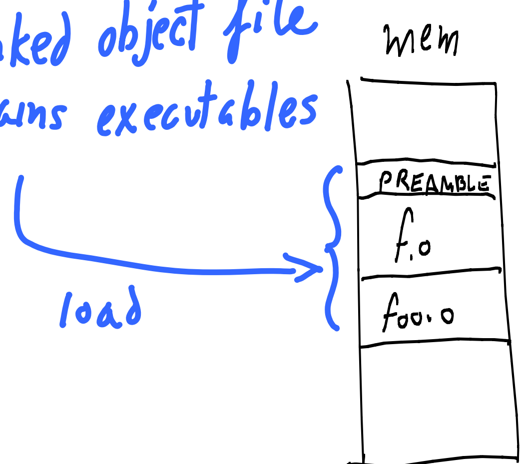
Library header provides pointers to sections of code.

Sections extracted and copied to form one file.

Loader:

Headers (.o file headers) stripped, executble code copied to memory, along w/ preamble.

References (addresses) fixed
- ---- at link time
- ---- at load time



Here's a .o file produced by gcc.

You won't find printf here, it hasn't been linked yet.

Assembly is in ATT syntax: destination on right.

You can guess at meaning.

```
%> gcc -S fo.c
%> more f.s

        .file   "f.c"
        .def    ___main;    .scl    2;    .type   32;
        .endef
        .section .rdata,"dr"
LC0:
        .ascii "Enter %d numbers.\12\0"

        .text
.globl _main
        .def    _main;      .scl    2;    .type   32;
        .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $104, %esp
        andl    $-16, %esp
        movl    $0, %eax
```

ASCII x0A = LF

ASCII x00 = NUL

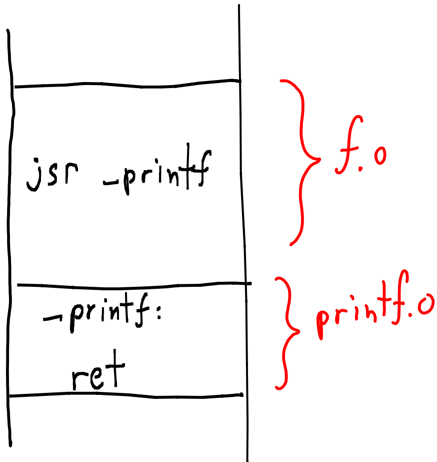
— save BP : push BP

— new BP : SP → BP

— make space : SP − 104

— align : x0 → SP[3:0]

. . . — more preamble



After linking and loading,

printf() becomes

jsr _printf

_printf is a label (i.e., an offset)

Contrast:

Dynamic linking (.DLL)

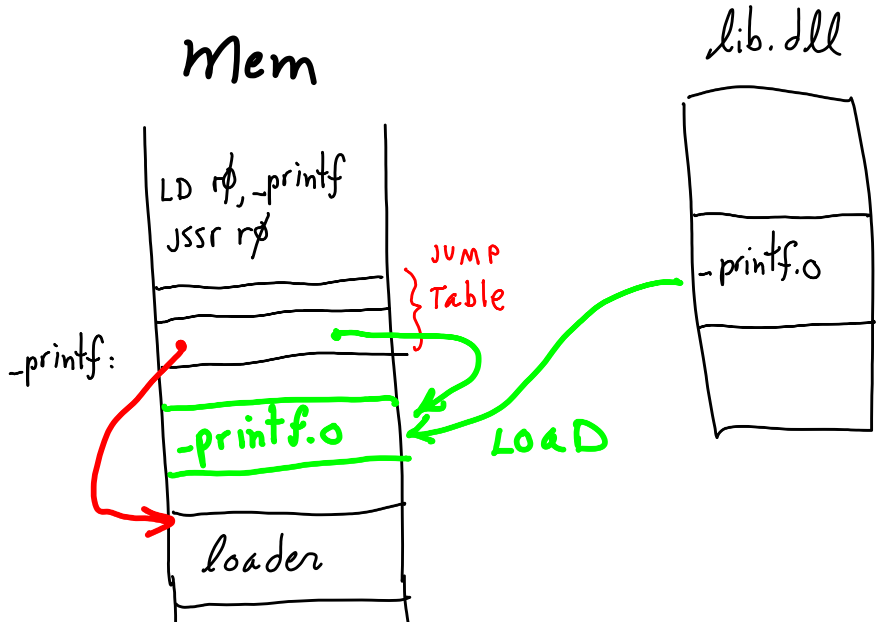
--- call is via a jump table

--- jump table filled in as needed at runtime

--- 1ST jump goes to loader

--- executable loaded

--- next time, jumps to executable



Combining C w/ assembly, LC3

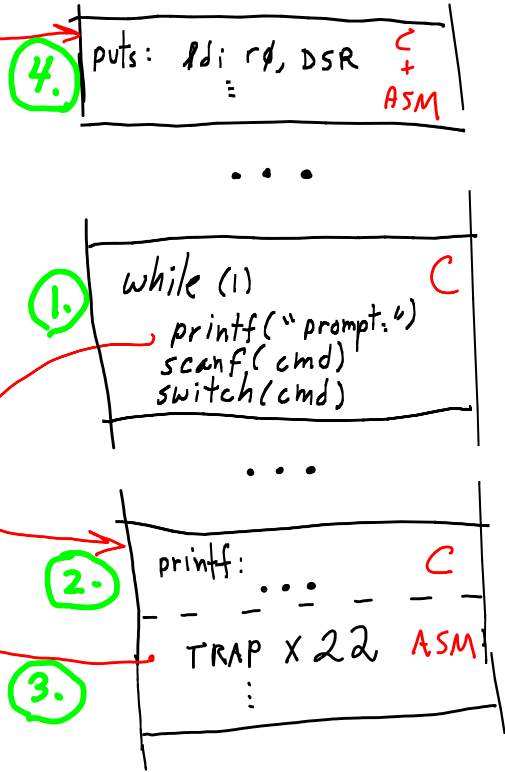
1. Need low-level operations → .asm
2. Use C when possible

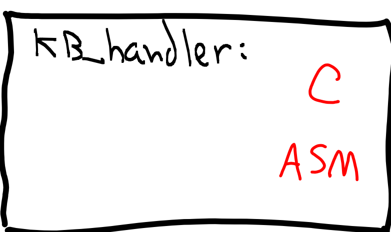
1. **user main() C code:**
   "printf()" ===> "jsr printf"

2. **_printf C code:**
   (handle formats, conversions, ...)

3. **_printf ASM code:**
   (do prep)
   trap x22 (jump to "puts")

4. **PUTS C + ASM code:**
   ...
   ldi r0, DSR
   ...

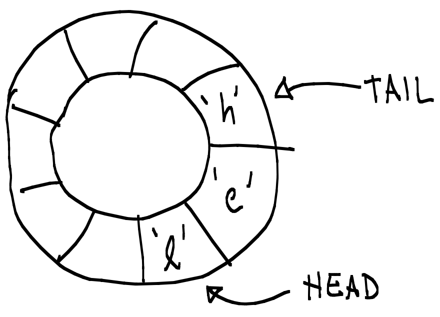




Things to handle in ASM

- r0 ← KBDR
- Turn on interrupts
- set up stack for call
- put R0 content on stack
- RTI

Things to handle in C



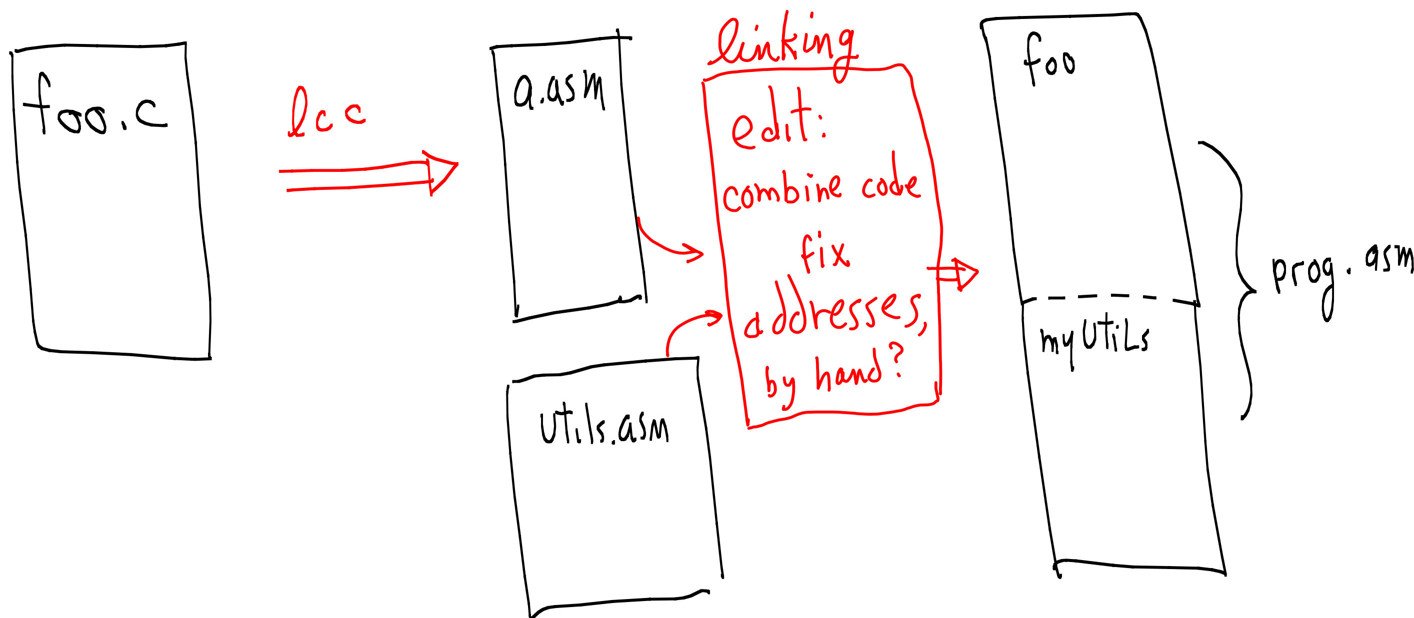

```
char buff[N];
head++;
if (head > N) head = 0;
if (head == tail) {
    head--;
    overflow();
else
    buff[head] = ch;
```

Linking C and ASM makes life easier

Can build more interesting stuff faster

Link higher-level languages too

Source code linking (LC3)



lc3as
obj2bin
sed 'ld'

run/prog.bin

How we run our prog.bin

1. Electric.open : test.jelib : lc3run : Tools.Simulation.WriteVerilogDeck
   ===> run/lc3run.v

2. cd run; iverilog lc3run.v
   ===> a.out

3.a (run w/o keyboard support):
   vvp a.out

3.b (run w/ keyboard support):
   ../bin/kb
   [esc]

To end simulation

* only need to do 1. and 2. once, then use same a.out for 3.

** check what lc3run reads into LC3 memory,
   os.bin w/ prog.bin? only os.bin?

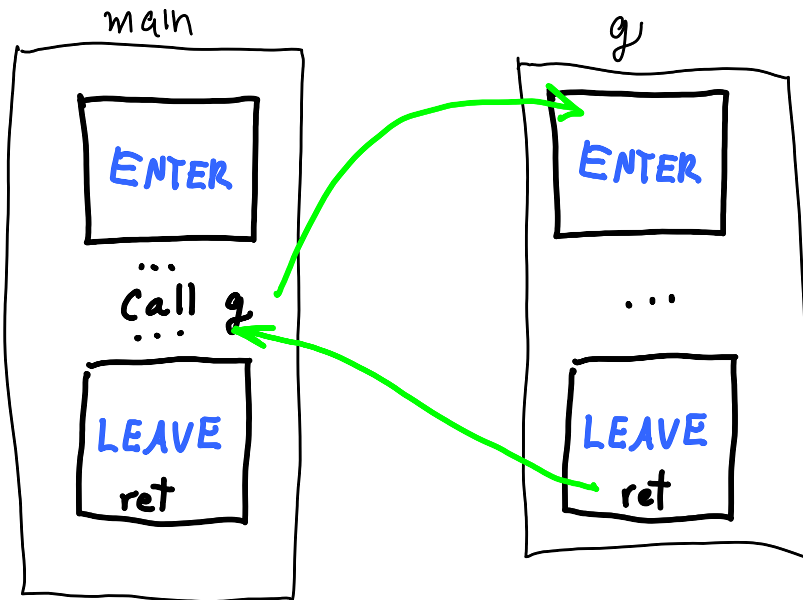
(W/ both OS and user progs):
```
readmemb("os.bin", mem.data[x0200]);
readmemb("prog.bin", mem.data[x3000]);
```

(Or, only foo.bin runs):
```
readmemb("foo.bin", mem.data[x0200]);
```

Or, use .obj in PennSim.
- No interrupts, but
- Good for debugging otherwise

refs:
traps in lc3 (w/ memory protection, user-super modes):
http://www.cis.upenn.edu/~milom/cse240-Fall05/handouts/Ch09-a.pdf

C conventions
lc3 lcc style

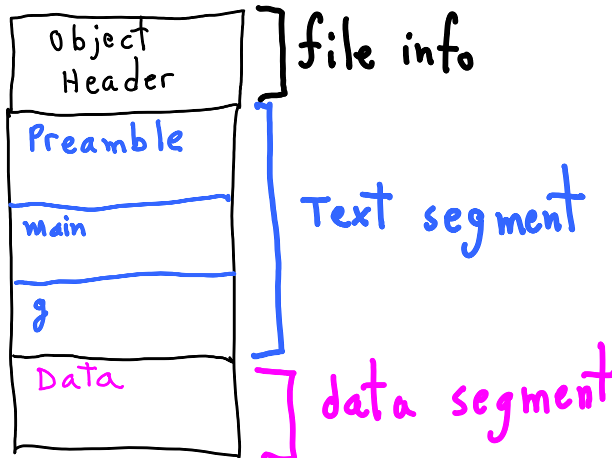


Standardized protocol

1. ENTER, set up stack
2. do stuff
3. LEAVE, unwind stack

PROG.OBJ

OS conventions



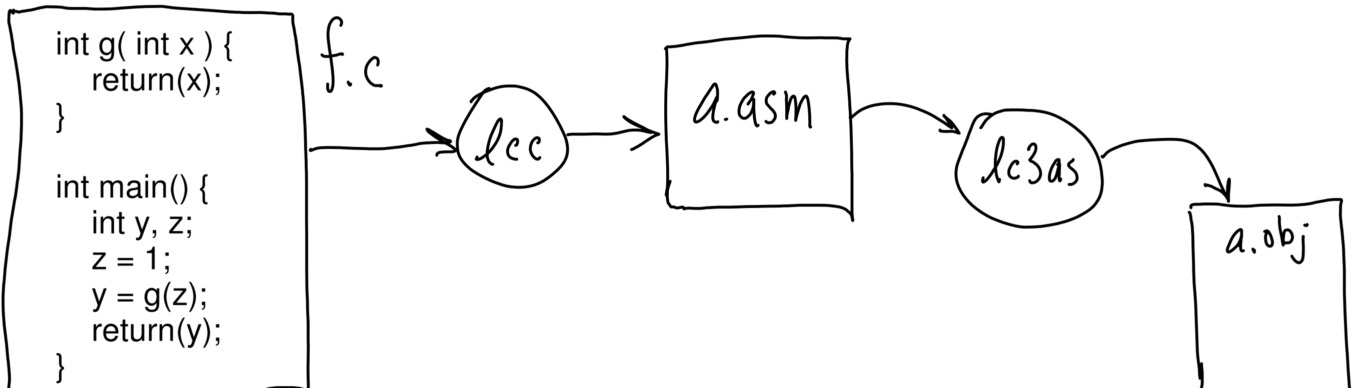
Object structure (.o or .obj)

0. Header(s)
   Pointers, offsets, types

1. Preamble inserted
   Handles OS conventions

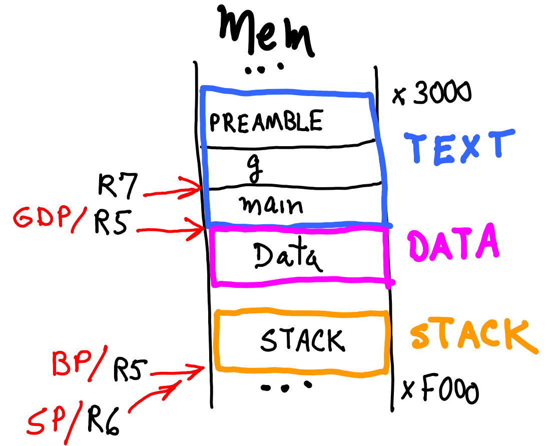
2. Text Segment(s)
   machine instructions

3. Data Segment(s)
   pointers to functions
   constants' data
   global variables
   variables' initial values



```
.Orig x3000
INIT_CODE     ;;----------------- PREAMBLE
LD R6, STACK_POINTER
LD R5, STACK_POINTER
LD R4, GLOBAL_DATA_POINTER
LD R7, GLOBAL_MAIN_POINTER
jsrr R7
HALT
GLOBAL_DATA_POINTER .FILL GLOBAL_DATA_START
GLOBAL_MAIN_POINTER .FILL main ;;-- pointer var.
STACK_POINTER .FILL xF000

;;---------------------------- TEXT SEGMENT
    ...  ( main's and g()'s text)  ...

;;---------------------------- DATA SEGMENT
GLOBAL_DATA_START:
g     .FILL lc3_g  ;;-- Pointer variable to g()
L1_f .FILL lc3_L1_f
L4_f .FILL lc3_L4_f
L3_f .FILL #2      ;;-- CONST 2
L5_f .FILL #1      ;;-- CONST 1
L2_f .FILL #5      ;;-- CONST 5
.END
```

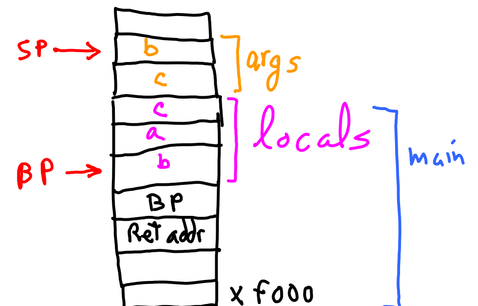
```
int g( int x, int w ) {
    int y, z;
    y = x+5+w;
    z = y+2;
    return(z);
}

int main(void){
    int a, b, c;
    a = 1;
    b = 2;
    c = a+b;
    return( g(b, c) );
}
```




```
main
    ;;------- BEGIN ENTER --------
ADD R6, R6, #-1   ;;-- allocate ret val space
ADD R6, R6, #-1   ;;-- SP--
STR R7, R6, #0    ;;-- push ret addr
ADD R6, R6, #-1   ;;-- SP--
STR R5, R6, #0    ;;-- push BP
ADD R5, R6, #-1   ;;-- set new BP
                  ;;----- allocate locals
ADD R6, R6, #-3
    ;;------- END ENTER ----------
```

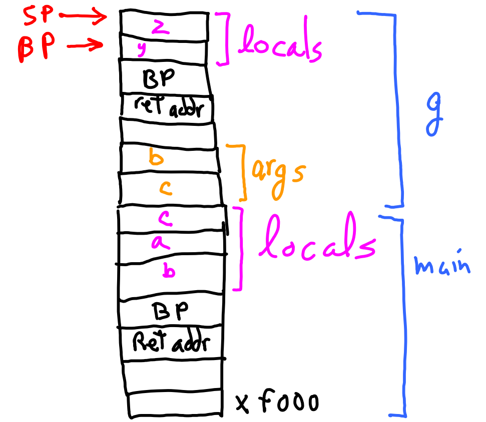
```
ldr R7, R5, #0   ;-- R7 <== b
ldr R3, R5, #-1  ;-- R3 <== a
add R3, R3, R7   ;-- R3 <== a+b
str R3, R5, #-2  ;-- c  <== R3
ldr R3, R5, #-2  ;--
ADD R6, R6, #-1  ;-- sp--
STR R3, R6, #0   ;-- push c
ADD R6, R6, #-1  ;-- sp--
STR R7, R6, #0   ;-- push b
ADD R0, R4, #0   ;-- R0 <== address of g() pointer
LDR R0, R0, #0   ;-- R0 <== address of g()
jsrr R0          ;-- call g()
```

} do arithmetic

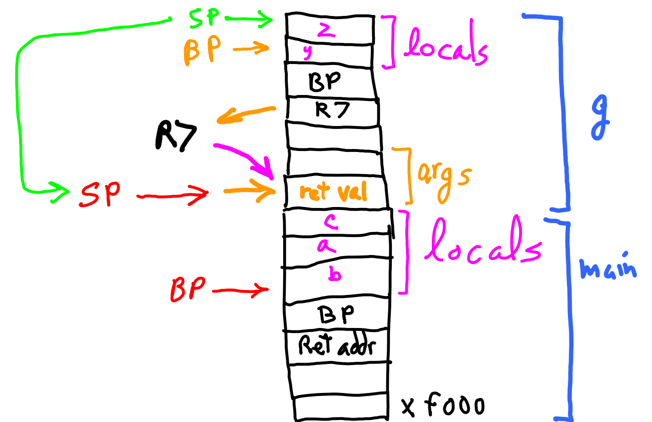


On call, SP points to 1st arg

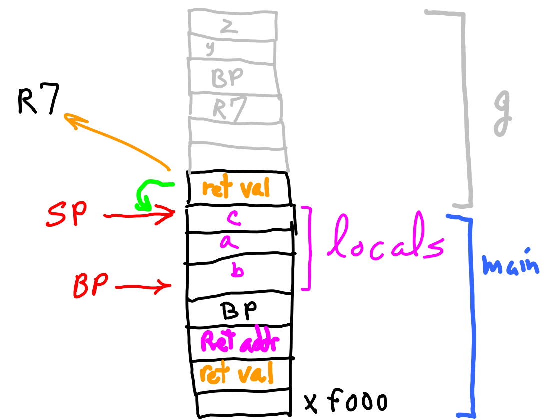
```
lc3_g
     ;;-------- BEGIN ENTER ---------
ADD R6, R6, #-1  ;;-- allocate ret val space
ADD R6, R6, #-1  ;;-- SP--
STR R7, R6, #0   ;;-- push ret addr
ADD R6, R6, #-1  ;;-- SP--
STR R5, R6, #0   ;;-- push BP
ADD R5, R6, #-1  ;;-- set new BP
                 ;;----- allocate locals
ADD R6, R6, #-2
     ;;-------- END ENTER ----------
```




```
     ;;-------- BEGIN-LEAVE ---------
LDR R6, R5, #5   ;;-- SP to last arg
STR R7, R6, #0   ;;-- ret val to last arg
LDR R7, R5, #2   ;;-- get saved ret addr
LDR R5, R5, #1   ;;-- restore BP
     ;;-------- END-LEAVE ---------
RET
```




```
LDR R7, R6, #0    ;-- pop ret val to R7
ADD R6, R6, #1    ;-- SP++
```



On return, pop result
(or pop void result)

Linking C and ASM | minimal approach

f.c

```c
#include "foo.h"

int main (void) {
   char ch;
   ch = getchar();
   putchar( ch );
   return(0);
}
```

foo.h

```c
char getchar(void);
void putchar(char);
```

Following a few conventions, we can write ASM which lcc links for us:

lcc f.c foo.asm

foo.asm

```
.global getchar
; char getchar(void)
LC3_GFLAG getchar LC3_GFLAG .FILL lc3_getchar

lc3_getchar

STR R7, R6, #-3   } save R7, R0
STR R0, R6, #-2
GETC  }
OUT   } --> { R0 <- KBDR; echo R0 }
STR R0, R6, #-1   --> Ret Val to STACK
LDR R0, R6, #-2   } Restore regs
LDR R7, R6, #-3
ADD R6, R6, #-1   --> point SP to ret val
RET

.global putchar
; void putchar(char)
LC3_GFLAG putchar LC3_GFLAG .FILL lc3_putchar

lc3_putchar

STR R7, R6, #-3   } save R7, R0
STR R0, R6, #-2

LDR R0, R6, #0    --- ( R0 <= arg )
OUT

LDR R0, R6, #-2   } restore R7, R0
LDR R7, R6, #-3
ADD R6, R6, #-0
RET

.END
```

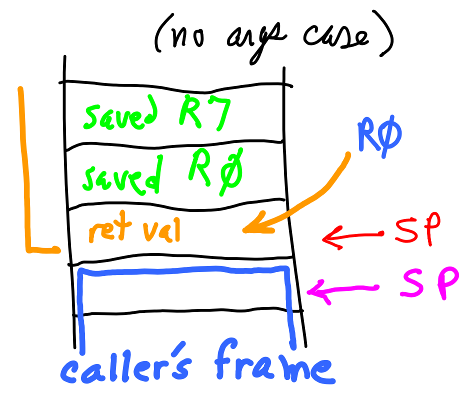
Convention on ENTRY

SP → 1st arg

or

SP → top of caller

(no args case)

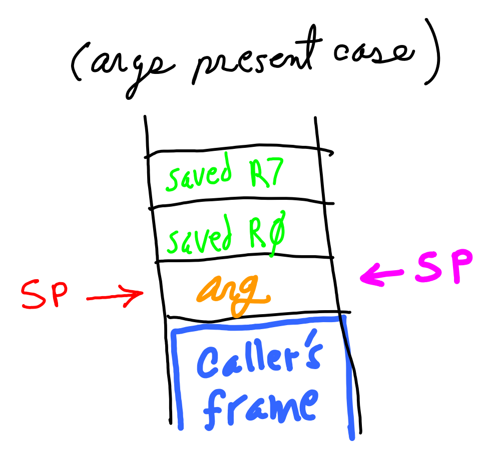


Convention on EXIT

SP → RET VAL

(just above caller's frame)

(args present case)



lcc compatibility

for User code loaded to x3000, run as-is: Use the .obj → .bin

→ Provided OS is loaded at x0200

for OS code, adjust: edit a.asm { ".Orig x3000" → ".Orig x0200" ; "xEFFF" → "x2FFF" }

lc3as a.asm → a.obj → os.bin

	lc3as	
GETC	→	TRAP x20
OUT	→	TRAP x21
PUTS	→	TRAP x22
IN	→	TRAP x23
PUTSP	→	TRAP x24
HALT	→	TRAP x25

Don't use these reserved words in your code.

These are used by lcc's C/ASM code.
Case (upper/lower) is ignored: halt == HALT

Our lc3pre definitions are not the same, e.g.,

```
trap(x20)__    ===>  push__( R7)
                     TRAP x20
                     pop__( R7 )

getc__         ===>  push__( R7)
                     TRAP x20
                     pop__( R7 )
```

Even so,
C conventions are still ok, push/pop is ok.
But,
not if you violate C's conventions (stack usage).

All is ok, as long as,
--- convention is not violated
--- trap routines do what is expected
--- trap vectors are the same

For OS code development,
Can use C+ASM as above, but
don't use C that calls a Trap, until your OS trap is implemented.

→ temporary fix: init all trap vectors to x0200 or to code to halt LC3 (LD STOP, STR MCR)

What is a function's name?

It's a pointer variable

foo:

lc3_foo:

Body of foo

```
void foo (void) {...}
```

You can pass it as an argument

```
do_something( foo );
```

How can you use it? An callee def'n

```
void   do_something( void (*argf)(void) ) {...}
```

formal parameter, a variable

that can be de-referenced
- a pointer variable to
- get the address of a function that
- returns void
- has void args

Use it?

```
(*argf)( );
```

→ calls the function, How?

1. – get address of varible's memory

2. Read Memory

R∅ ← Mem[x1234]

3. jump   JSSR R∅