LC3 Assembly

Assembly Language Programming Tools

src/lc3tools_v12.zip

* get most up-to-date version of src

Tools:
--- lc3as: .asm, assembly language text(ascii) ==> .obj, machine code
--- PennSim.jar, LC3 simulator (debugger)
--- lc3convert: .bin, machine language text(ascii) ==> .obj, machine code

Files:
--- .asm: text, assembly language code in ASCII
--- .obj, LC3 "load module": machine code plus header in BITS
--- .bin: text, binary rep. of machine code in ASCII

Building/using Tools

See documents in LC3-trunk

src/README

src/lc3tools/Readme

src/Makefile (learn "make")

run/README

Notes:

1. lc3tools comes as a zipped file. It must be unpacked. src/Makefile has commands for this.

2. src/Makefile is set up to compile lc3as and put it into "../bin/". Other executables land there also. To have access to these, set your PATH variable.

3. Read Makefile to see what it does. Makefile serves as documentation on how to get things done; so, it isn't necessarily that you always use "make", you might do these things by hand instead.

4. Things are set up under the assumption that temporary files (.obj, .bin, .v, .out ...) go to run/ and are used there. No need to add these temporaries to your branch.

5. PennSim.jar reads .obj files. We also will be loading our .obj code into our LC3's memory in our testbench. For that, the .obj must be translated BACK to ascii because verilog can only read ascii files. The tool that does this is "obj2bin" and the result is a .bin file.

Building lc3tools (lc3as, lc3convert, ...):

```
$> cd trunk/src
$> make lc3tools
```

Compiling lc3tools depends on having these:
-- unzip
-- flex
-- gcc
-- tcl/tk

You can find these on cygwin, XCode, and MacPorts.

Set your shell's PATH variable!

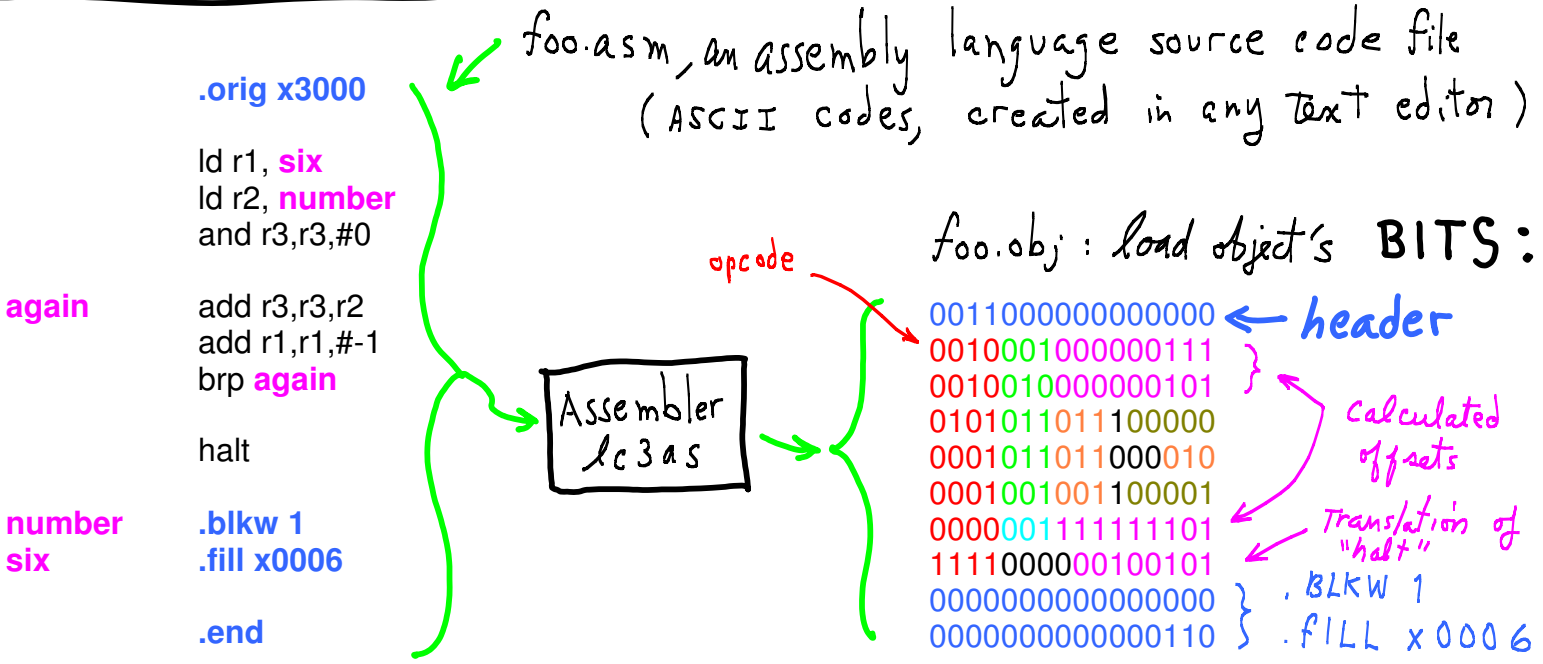
```
$> cd
$> vi .bash_profile
        PATH="<path to LC3-trunk/bin>:${PATH}"
$> source .bash_profile
```

Don't forget the ":". The "<path to LC3-trunk/bin>" is the full path to your branch's bin/. You can find it this way: "cd" to your LC3-trunk/bin, and then,

```
$> pwd
```

Assembly Language

P&P, Figure 7.1



Assembler (lc3as) **Directives** (to control the assembly process):

.orig: puts a load address into the .obj load-object file's header.
.end: tells assembler, this is the end of source code.
.blkw: tells assembler, create *n* blank words (all zeroes).
.fill: tells assembler, put these bits into a word.

The assembler produces machine code words:
--- ONE PER LINE expressing an LC3 instruction
--- ONE PER LINE where there is a .fill directive
--- n PER LINE where there is a .blkw directive

The assembler also calculates offsets for us using **symbols**. Symbols stand for memory addresses (starting for the .orig address). Offsets are calculated by subtraction. Symbols refer to the next instruction's location.

--- foo.obj contains BITS, not ascii codes for "1" and "0".

--- Use this to "see" the bits (you can't unless they are translated and printed):

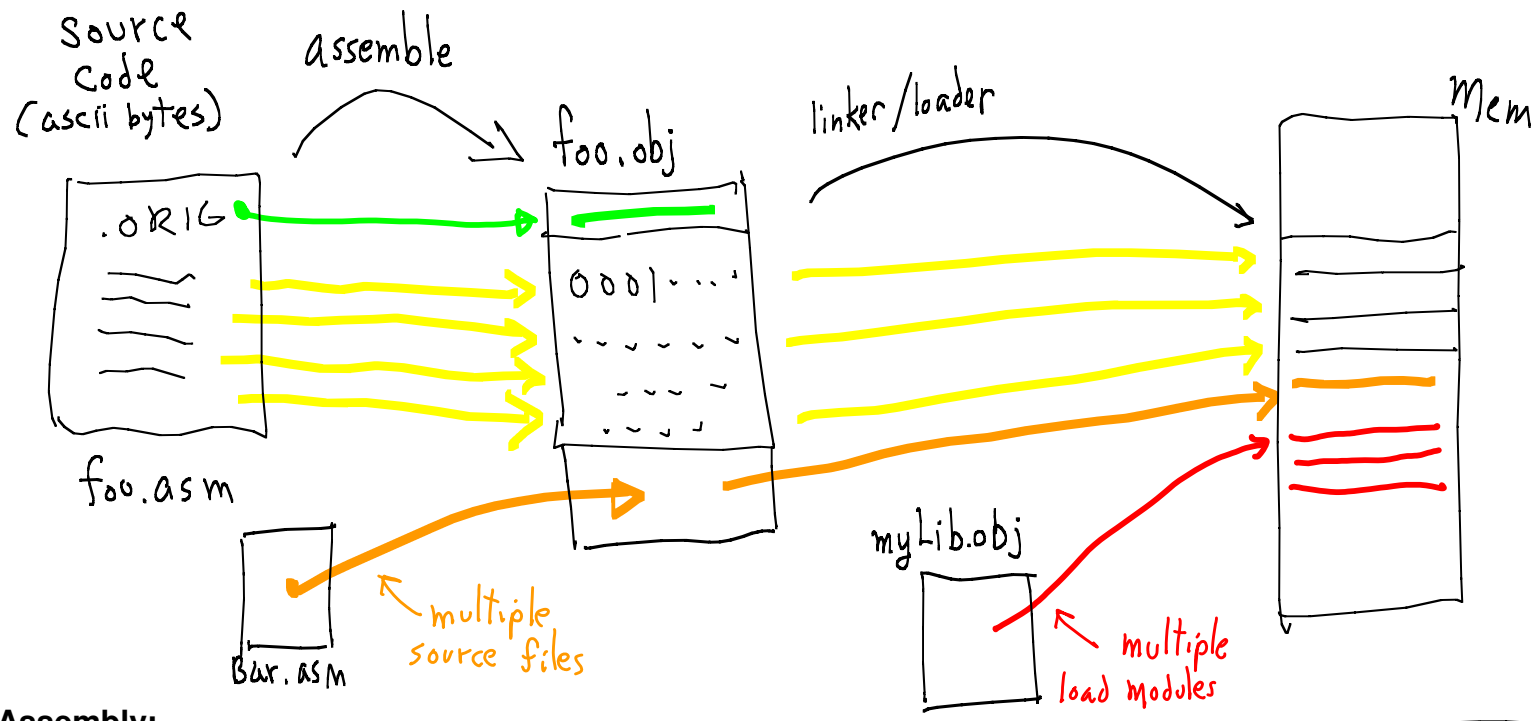
$> od -t x1 foo.obj

displays file's contents, expressed as 1-byte integers in hex notation (**x1**), first byte in file to last:

```
0000000 30 00 22 07 24 05 56 e0 16 c2 12 7f 03 fd f0 25
0000020 00 00 00 06
0000024
```

Yellow are addresses (offsets) into the file (in hex). **Blue** is the .obj file header.
Notice the **big-endian** order. This is most likely an artifact of the way the lc3tools' simulator was written. It does not reflect the LC3 micro-architecture because LC3's memory is **not byte-addressable**.

Assembler actions



Assembly:

strings (instructions) ==> machine words (LC3 instructions)
names ==> offsets in instruction words
directives (storage) ==> machine words (either 0s or some n)

There could be more in .obj:

names (symbols) ==> name/offset pairs (Symbol Table)
names referring to other files ==> translation of "external" source code

File Formats:

Standards define location and representation of information in .obj files.

Linker/Loader:

--- Combine separate load object modules

--- Fix offsets (references)

--- Copy to memory

LC3 Assembly Language | See P&P APPEND. A

Source Code:

```
AND R3, R3, #0    ; This line of code needs explanation.
```

Operation name: maps 1-1 to opcode

NB--there are some special translations, eg., "Halt".

designate an immediate value
(# = base 10 representation)
(x = base 16)
(b = base 2)

Comments: everything on the line after a ";" is ignored. Also, all white space is ignored.

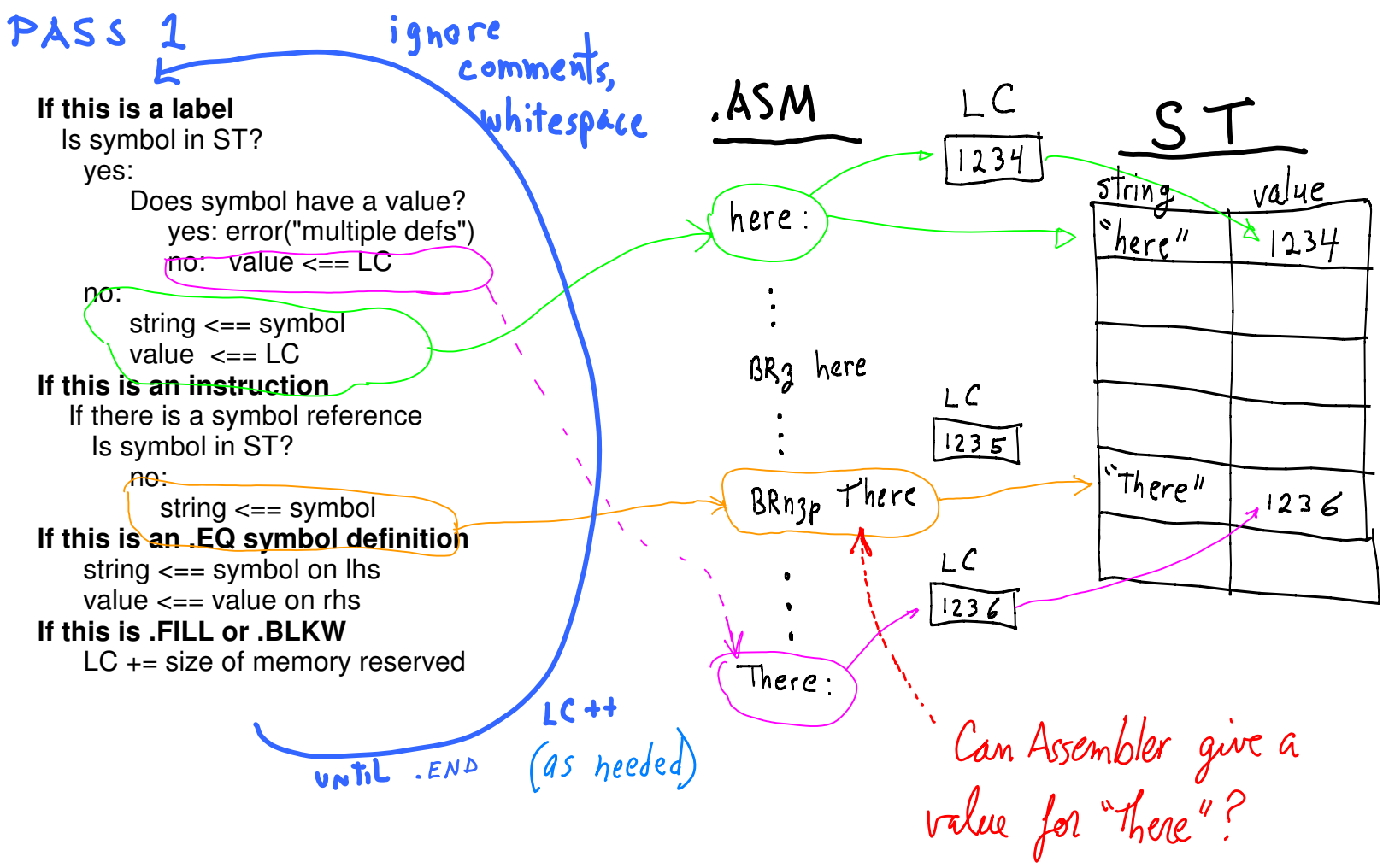
Three ways to write the same instruction:

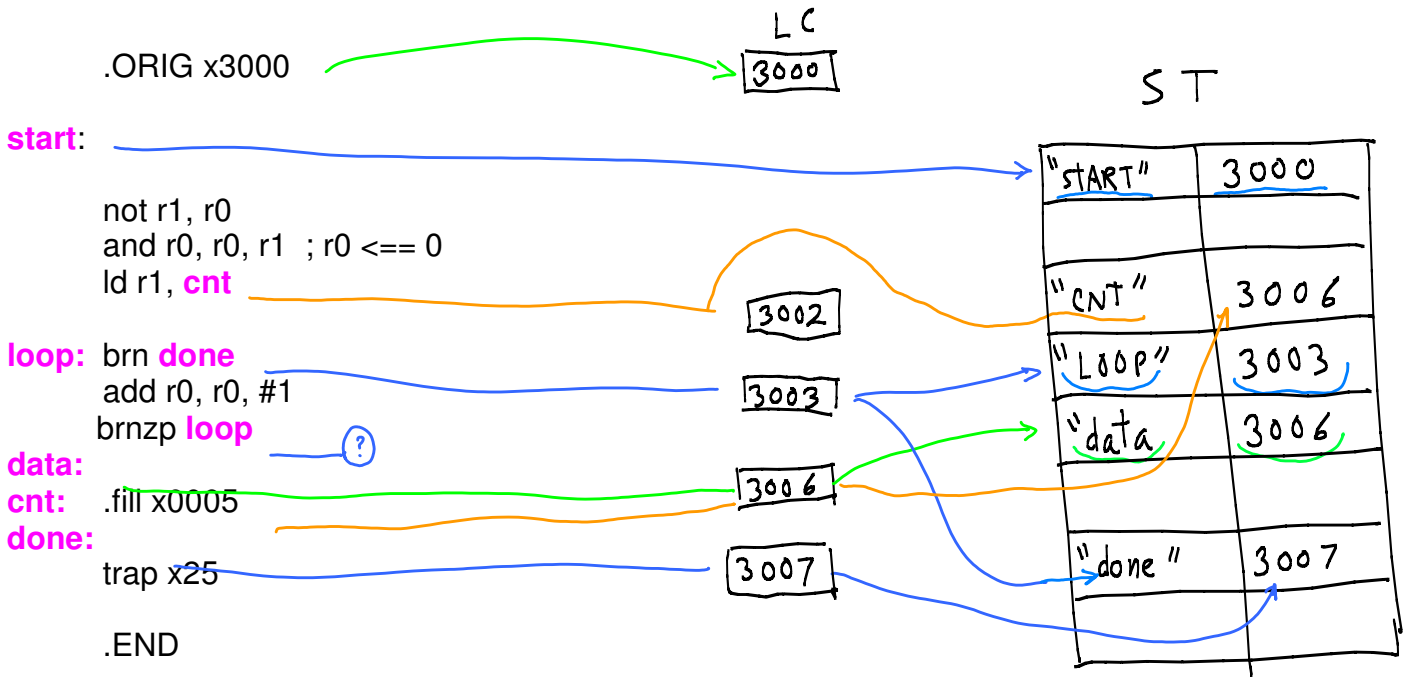
and r3, r3, b10101 and r3, r3, x15 and r3, r3, #21

Assembly Process

PASS 1: Find all symbols and record offsets: Built **S**ymbol **T**able.
initialize: LC <== value in .ORIG declaration

PASS 2: replaced all symbol references w/ offsets,
translate instructions to machine code.





what does assembler produce for "start"?
What address (LC value) does "not r1, r0" have?

PASS 2

*.asm

1. Calculate offsets

Q. Bug in code?

```
        .ORIG x3000

start:

        not r1, r0
        and r0, r0, r1
        ld r1, cnt

loop:   brz done
        add r0, r0, #1
        brnzp loop

data:
cnt:    .fill x0005
done:

        trap x25

        .END
```

```
x3000
not r1, r0
and r0, r0, r1
ld r1, ( cnt - LC - 1)
brz ( done - LC - 1)
add r0, r0, x01
brnzp ( loop - LC - 1)
x0005
trap x25
```

ST

String	Value
"start"	3000
"cnt"	3006
"loop"	3003
"data"	3006
"done"	3007

```
x3000
not r1, r0
and r0, r0, r1
ld r1, (3006 - 3002 - 1) = +3
brz (3007 - 3003 - 1) = +3
add r0, r0, x01
brnzp (3003 - 3005 - 1) = -3
x0005
trap x25
```

.obj

```
0011 0000 0000 0000   } header
1001 001 000 111111
0101 000 000 0 00 001
0010 001 000000011    +3
0000 010 000000011    +3
0001 000 000 1 00001
0000 111 111111101    -3
0000 0000 0000 0101
1111 0000 0010 0101
```

2. generate machine code,

3. If not END, go to 1.

Offset calculation and instruction translation to machine code are done a line at a time, looping until input is exhausted.

Simulator work flow

1. **Create** an assembly language source file using any text editor ===> **f.asm**

2. **Assemble** source code to load object:
   lc3as f.asm ===> **f.obj**

3. **Load** machine code into PennSim.jar for testing, eg.

   PennSim.File.Open_OBJ_file

4. **Load** via verilog, simulate your LC3
   a. convert f.obj to **f.bin** (use obj2bin, see src/Makefile)
   b. write a verilog testbench that loads f.bin into LC3's memory (see test.jelib)

More on assembly language directives

.STRINGZ "abcd"

Same as

```
.FILL x0061   'a'
.FILL x0062   'b'
.FILL x0063   'c'
.FILL x0064   'd'
.FILL x0000   NUL
```

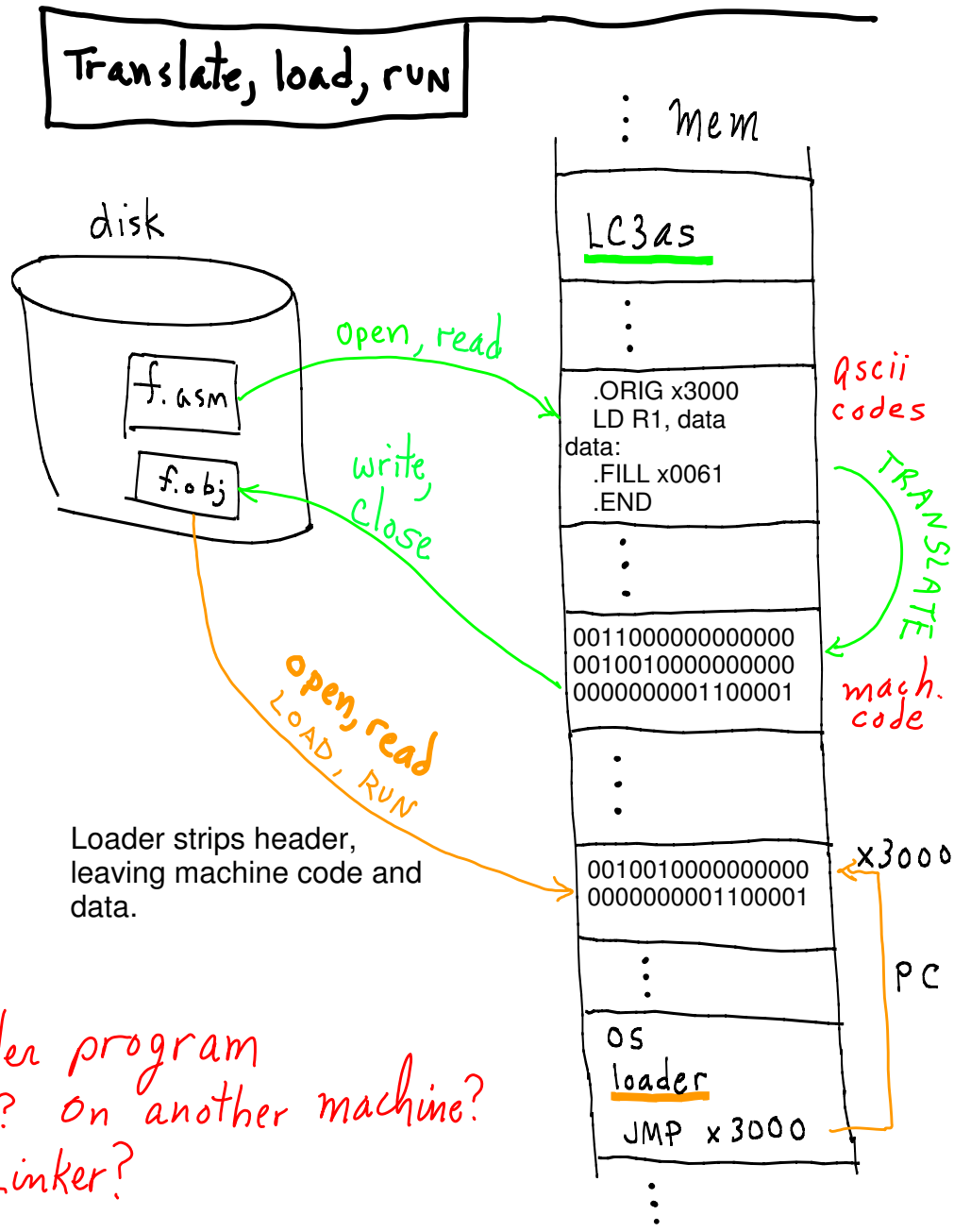
.BLKW 3

Same as

```
.FILL x0000
.FILL x0000
.FILL x0000
```

.FILL <data word>
<data word> ===> .obj file.

NB--Loader could do part of job of assembler: leave ".BLKW 3" in header, fill memory at load time.



Where does loader program execute? On LC3? On another machine? Translater (LC3as)? Linker?

Loader knows where to jump?

Could loader "relocate" mach. code to another memory location?

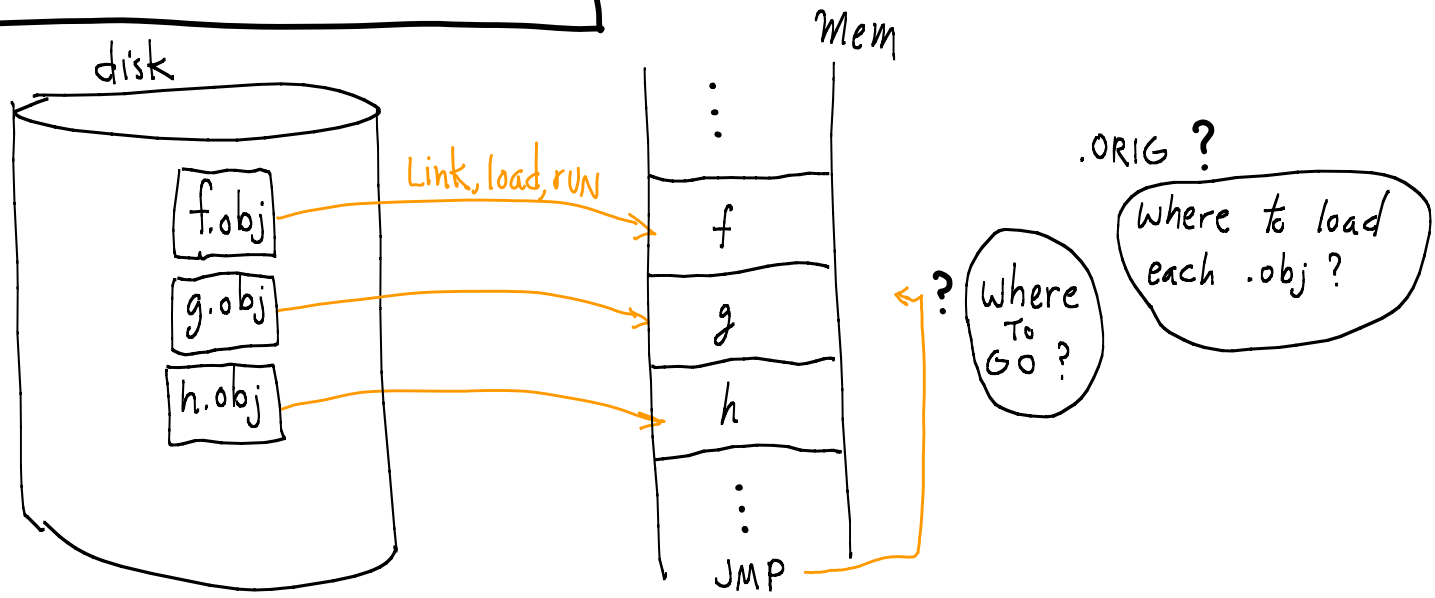
Cross-platform development? How? Communication?

What if LC3 is only simulated? Cross-platform development + communication?

```
;-------------------------------------------------------------------------------
;  parityFSM.asm
; The parity finite-state machine.
;-------------------------------------------------------------------------------
    .ORIG x3000

;---- start up ----
    lea r1, Input          ;-- r1 points to input tape/memory area.
    add r1, r1, x-1        ;-- (minus 1 so states initially compute correct read location.)
    lea r2, Output         ;-- r2 points to output area.
    add r2, r2, x-1        ;-- (minus 1 as above.)

;---- state 0 -----
State_0:
    add r1, r1, x1         ;-- r1++ (move Read head R, towards larger addresses)
    ldr r3, r1, #0         ;-- r3 <== *r1 (dereference pointer r1 to read input)
    and r3, r3, r3         ;-- r3 <== r3  (is r3 == 0?)
    brz   State_0          ;--    yes: stay in state 0.
    brnzp State_1          ;--     no: go to state 1.

;---- state 1 -----
State_1:
    add r1, r1, x1         ;-- move Read head R (towards larger addresses)
    ldr r3, r1, #0         ;-- r3 <== *r1 (read)
    and r3, r3, r3         ;-- r3 <== r3  (is r3 == 0?)
    brz   State_1          ;--    yes: stay in state 1.
    brnzp State_0          ;--     no: go to state 0.

;----------Tape Area----------
Input:
    .FILL x1
    .FILL x0
    .FILL x1
    .FILL x1
    .FILL x0
    .FILL x0
    .FILL x1
    .FILL x1
Output:
    .BLKW #8
    .END
```

Note: Head only moves R. Head moving R is serial input to FSM.

BEYOND LC3as Assembler | Separate compilation, Libraries



Separate compilation/assembly

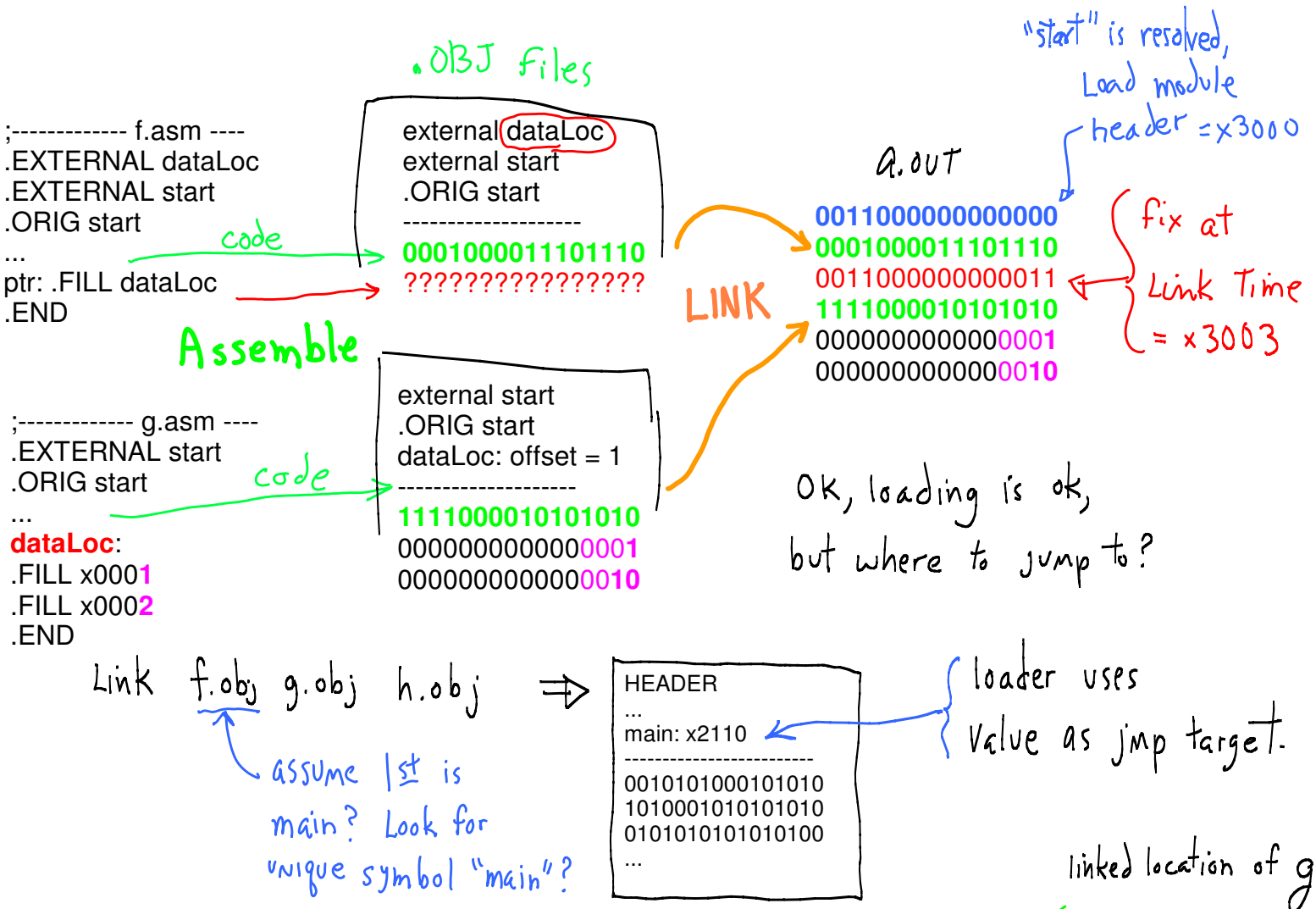
--- Create libraries of pre-translated code, never translate again.

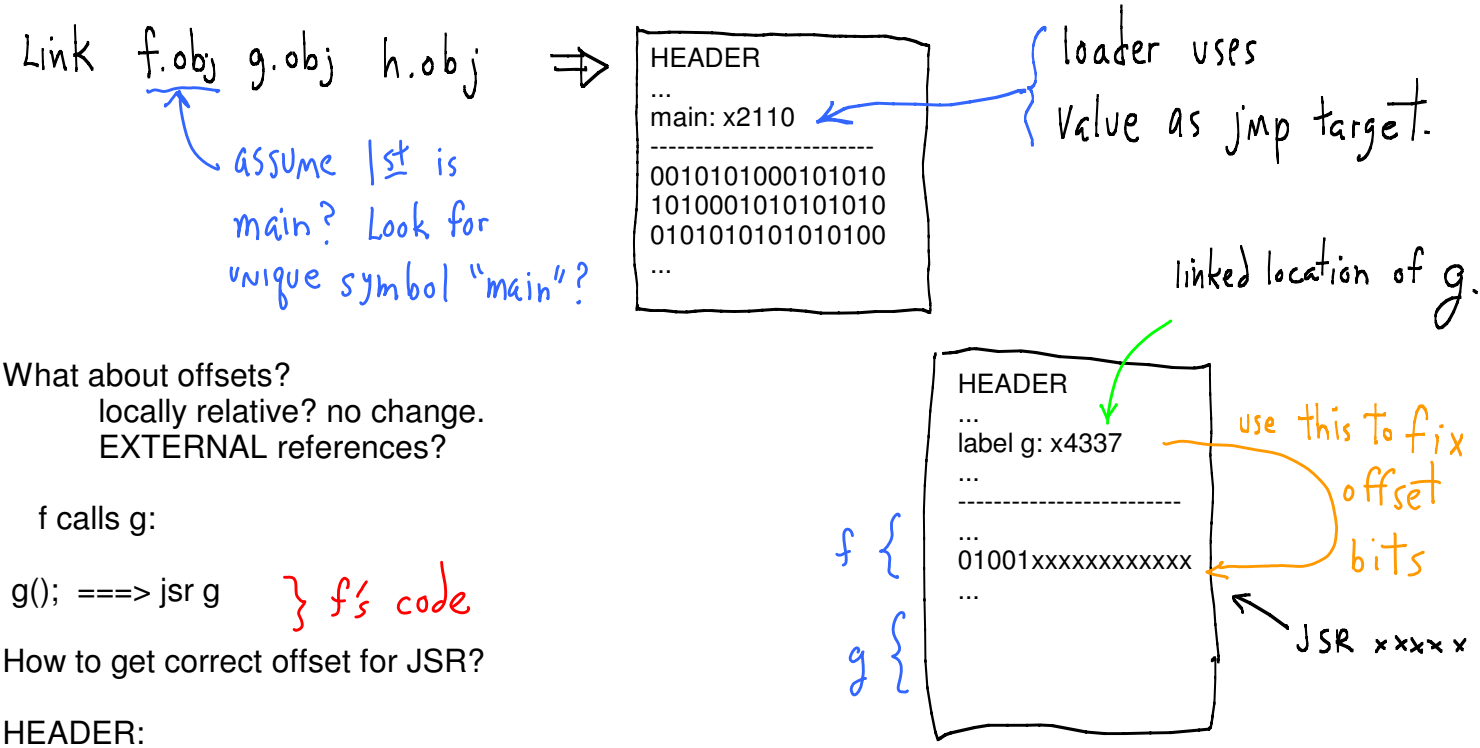
--- Use library routines: use name of function in code.

--- Library pieces are loaded as needed.

--- Each has its own .ORIG, but not exactly relevant as-is:

- --- decide order of layout
- --- some offsets and all fixed addresses need to be adjusted





What about offsets?
locally relative? no change.
EXTERNAL references?

f calls g:

g(); ===> jsr g } f's code

How to get correct offset for JSR?

HEADER:
list of references and where.

Linker calculates offsets and edits machine code.

Quiz:

After linking/loading:

--- f's code located at x3000.

--- g's code located at x4337.

--- What is the address of the last word of f?

--- f has how many words?