Syntax of LC-3

- One instruction, declaration per line
- Comments are anything on a line following ";"
- Comments may not span lines

LC-3 has 2 basic data types
  - Integer
  - Character

Both take 16-bits of space (a word) though a character is only 8-bits in size.
**Directives** give information to the assembler. All directives start with ‘.’ (period)

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ORIG</td>
<td>Where to start in placing things in memory</td>
</tr>
<tr>
<td>.FILL</td>
<td>Declare a memory location</td>
</tr>
<tr>
<td>.BLKW</td>
<td>Reserve a group of memory locations</td>
</tr>
<tr>
<td>.STRINGZ</td>
<td>Declare a group of characters in memory</td>
</tr>
<tr>
<td>.END</td>
<td>Tells assembly where your program source ends</td>
</tr>
</tbody>
</table>
We are writing source code. We need to translate that to binary so it can be run on the LC-3.

This is the job of the assembler

\texttt{LEA 20, Hello}

\texttt{\rightarrow XE002}
Memory

- Our program needs to be stored in our memory and it is placed there by the assembler.
- Each line of our code causes the assembler to store data at memory locations.
- Data is placed sequentially by instruction
  - Instructions themselves are encoded as 16-bit binary numbers.
.ORIG

• Tells simulator where to put your code in memory
• Does not use memory itself
• Only one allowed per program
• We start at this address
.ORIG in Memory

- .ORIG x3000
- ADD R1,R3,R7
- NOT R1,R1

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Memory (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td>ADD R1,R3,R7</td>
<td>x12C7</td>
</tr>
<tr>
<td>0x3001</td>
<td>NOT R1,R1</td>
<td>x927F</td>
</tr>
<tr>
<td>0x3002</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

.ORIG x 4000 < main < x 3000
“a typed language”

```c

type varname;

```

```

type is

  int    (integer)
  char   (character)
  float  (floating point)

```

“LC-3”

```c

varname(really label) .FILL value

```

```

value is required  – the initial value

```
.FILL

flag     .FILL     x0001
counter  .FILL     x2
letter   .FILL     x0041 ; A
letters  .FILL     -436

• One declaration per line
• Always declaring 16-bits, the word size of LC-3
• Don’t mix in with your code, will be treated like an instruction
Program

\[ \sim 90\% \]
.FILL in Memory

- .ORIG x3000
- ADD R1,R3,R7
- NOT R1,R1
- .FILL x0001
- .FILL x2
- .FILL -436

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Memory (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td>ADD R1,R3,R7</td>
<td>x12C7</td>
</tr>
<tr>
<td>0x3001</td>
<td>NOT R1,R1</td>
<td>x927F</td>
</tr>
<tr>
<td>0x3002</td>
<td>No Instruction</td>
<td>x0001</td>
</tr>
<tr>
<td>0x3003</td>
<td>NOP</td>
<td>x0002</td>
</tr>
<tr>
<td>0x3004</td>
<td>NOP</td>
<td>xFE4C</td>
</tr>
<tr>
<td>0x3005</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x3006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x3007</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x3008</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
.BLKW

- Tells assembler to set aside some number of sequential memory locations
- Useful for arrays
- Can be initialized
Examples of .BLKW:

; set aside 3 locations
.BLKW 3

; set aside 1 location and label it.
Bob .BLKW 1

; set aside 1 location, label and initialize to x4.
Num .BLKW 1 x4

; set aside 10 locations, label and initialize to 37.
Num .BLKW 10 37
.BLKW in Memory

- .ORIG x3000
- ADD R1, R3, R7
- NOT R1, R1
- .BLKW 2
- .BLKW 4 6

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Memory (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td>ADD R1, R3, R7</td>
<td>x12C7</td>
</tr>
<tr>
<td>0x3001</td>
<td>NOT R1, R1</td>
<td>x927F</td>
</tr>
<tr>
<td>0x3002</td>
<td>NOP</td>
<td>x0000</td>
</tr>
<tr>
<td>0x3003</td>
<td>NOP</td>
<td>x0000</td>
</tr>
<tr>
<td>0x3004</td>
<td>NOP</td>
<td>x0006</td>
</tr>
<tr>
<td>0x3005</td>
<td>NOP</td>
<td>x0006</td>
</tr>
<tr>
<td>0x3006</td>
<td>NOP</td>
<td>x0006</td>
</tr>
<tr>
<td>0x3007</td>
<td>NOP</td>
<td>x0006</td>
</tr>
<tr>
<td>0x3008</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
.STRINGZ

- Used to declare a string of characters
- Is terminated by x0000
- One character per memory location

Example:

hello .STRINGZ “Hello World!”
.STRINGZ in Memory

- .ORIG x3000
- ADD R1,R3,R7
- NOT R1,R1
- .STRINGZ “FooBar\n”

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Memory (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td>ADD R1,R3,R7</td>
<td>x12C7</td>
</tr>
<tr>
<td>0x3001</td>
<td>NOT R1,R1</td>
<td>x927F</td>
</tr>
<tr>
<td>0x3002</td>
<td>NOP</td>
<td>x0046</td>
</tr>
<tr>
<td>0x3003</td>
<td>NOP</td>
<td>x006F</td>
</tr>
<tr>
<td>0x3004</td>
<td>NOP</td>
<td>x006F</td>
</tr>
<tr>
<td>0x3005</td>
<td>NOP</td>
<td>x0042</td>
</tr>
<tr>
<td>0x3006</td>
<td>NOP</td>
<td>x0061</td>
</tr>
<tr>
<td>0x3007</td>
<td>NOP</td>
<td>x0072</td>
</tr>
<tr>
<td>0x3008</td>
<td>NOP</td>
<td>x000A</td>
</tr>
<tr>
<td>0x3009</td>
<td>NOP</td>
<td>x0000</td>
</tr>
</tbody>
</table>
.END

- Tells the assembler where your program ends
- Only one allowed in your program
Simple LC-3 program

```
.ORIG x3000
LD R2, Zero
LD R0, M0
LD R1, M1
BRz Done
ADD R2, R2, R0
ADD R1, R1, -1
BR Loop
ST R2, Result
HALT ; same as a "TRAP x25"
.FILL x0000
.FILL x0000
.FILL x0004
.FILL x0002
.END
```

- What does this program do?
- What is in "Result" at the end?
HLL – if/else statements...

```
if condition
  statement
else
  statement
```
"Generic"  if count < 0
  count = count + 1

"LC-3"
  LD R0, count
  BRpz greatzero
  ADD R0, R0, #1
  ST R0, count

  greatzero ; next instruction goes here
Loops can be built out of IF’s – WHILE:

“Generic”

while count > 0
    a = a + count
    count = count - 1
“LC-3”

LD R1, a
LD R0, count
while BR nz
  ADD R1, R1, R0
  ADD R0, R0, #-1
  while
    BR
    ST R1, a
    ST R0, count
  endwhile
endwhile
For loops

“general”

\[
\text{for } l = 3 \text{ to } 8 \\
a = a + l
\]
"LC-3"

; R0=a, R1=l, R2=temp

LD    R0, a
AND   R1, R1, #0  ; init l to zero
ADD   R1, R1, #3  ; now make 3

for
ADD   R2, R1, #-8
BRp   endfor
ADD   R0, R0, R1  ; a=a+l
ADD   R1, R1, #1  ; l++
BR     for

endfor

; same as BRnzp
System Calls (TRAPS) and Subroutines
System Calls

Certain operations require specialized knowledge and protection:
- specific knowledge of I/O device registers and the sequence of operations needed to use them
- I/O resources shared among multiple users/programs; a mistake could affect lots of other users!

Not every programmer knows (or wants to know) this level of detail

Provide service routines or system calls (part of operating system) to safely and conveniently perform low-level, privileged operations

OUT
System Call
(service routines)

1. User program invokes system call.
2. Operating system code performs operation.
3. Returns control to user program.

In LC-3, this is done through the TRAP mechanism.
LC-3 TRAP Mechanism

1. A set of service routines.
   - part of operating system -- routines start at arbitrary addresses
   - System code by convention is typically below address x3000
   - up to 256 routines

2. Table of starting addresses.
   - stored at x0000 through x00FF in memory
   - called “System Control Block” or “Vector Table” in some architectures

3. TRAP instruction.
   - used by user program to transfer control to operating system
   - 8-bit trap vector names one of the 256 service routines

4. A linkage back to the user program.
   - want execution to resume immediately after the TRAP instruction
TRAP Instruction

• Trap vector (trapvect8)
  – identifies which system call to invoke
  – 8-bit index into table of service routine addresses
    • in LC-3, this table is stored in memory at 0x0000 – 0x00FF
    • 8-bit trap vector is zero-extended into 16-bit memory address

• Where to go
  – lookup starting address from table; goes to that location

• How to get back
  – saves address of next instruction (current PC) in R7 before changing PC
How do we transfer control back to instruction following the TRAP?

- Save old position in R7.
  - JMP R7 gets us back to the user program at the right spot.
  - LC-3 assembly language lets us use RET (return) in place of “JMP R7”.

- Must make sure that service routine does not change R7, or it won’t know where to return.
TRAP Mechanism Operation

1. Lookup starting address.
2. Transfer to service routine.
3. Return (JMP R7).
Example: Using the TRAP Instruction

; This code just takes upper case characters and converts
; to lower case and prints them. Terminates with a "7"

.ORIG x3000

LD R2, TERM ; Load negative ASCII '7'
LD R3, ASCII ; Load ASCII difference
AGAIN
    TRAP x23 ; input character
    ADD R1, R2, R0 ; Test for terminate: =7?
    BRz EXIT ; Exit if done
    ADD R0, R0, R3 ; Change to lowercase
    TRAP x21 ; Output to monitor...
    BRnzp AGAIN ; ... again and again...
TERM .FILL xFFC9 ; −'7’in 2SC
ASCII .FILL x0020 ; lowercase offset
EXIT TRAP x25 ; halt
.END
The OUT Service Routine

.ORIG x0430 ; syscall address
ST R7, SaveR7 ; save R7 & R1
ST R1, SaveR1

; ----- Write character
TryWrite LDI R1, CRTSR ; get status
BRzp TryWrite ; look for bit [15] on
WriteIt STI R0, CRTDR ; write char

; ----- Return from TRAP
Return LD R1, SaveR1 ; restore R1 & R7
LD R7, SaveR7 ; back to user
RET

CRTSR .FILL xF3FC
CRTDR .FILL xF3FF
SaveR1 .FILL 0
SaveR7 .FILL 0
.END

stored in table, location x21

Maxwell James Dunne
## TRAP Routines and their Assembler Names

<table>
<thead>
<tr>
<th>vector</th>
<th>symbol</th>
<th>routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>x20</td>
<td>GETC</td>
<td>read a single character (no echo)</td>
</tr>
<tr>
<td>x21</td>
<td>OUT</td>
<td>output a character to the monitor</td>
</tr>
<tr>
<td>x22</td>
<td>PUTS</td>
<td>write a string to the console</td>
</tr>
<tr>
<td>x23</td>
<td>IN</td>
<td>print prompt to console, read and echo character from keyboard</td>
</tr>
<tr>
<td>x25</td>
<td>HALT</td>
<td>halt the program</td>
</tr>
</tbody>
</table>
Saving and Restoring Registers

Must save the value of a register if:
- Its value will be destroyed by service routine, and
- We will need to use the value after that action.

Who saves?
- caller of service routine?
  - knows what it needs later, but may not know what gets altered by called routine
- called service routine?
  - knows what it alters, but does not know what will be needed later by calling routine
Saving and Restoring Registers

Called routine -- "callee-save"
- Before start, save any registers that will be altered (unless altered value is desired by calling program!)
- Before return, restore those same registers

Calling routine -- "caller-save"
- Save registers destroyed by own instructions or by called routines (if known), if values needed later
  - save R7 before TRAP
  - save R0 before TRAP x23 (input character)
- Or avoid using those registers altogether

Values are saved by storing them in memory.
Question

Can a service routine call another service routine?  YES

If so, is there anything special the calling service routine must do?  R7
What about User Code?

Service routines provide three main functions:
1. Shield programmers from system-specific details.
2. Write frequently-used code just once.
3. Protect system resources from malicious/clumsy programmers.

Are there any reasons to provide the same functions for non-system (user) code?
Subroutines

A subroutine is a program fragment that:
- lives in user space
- performs a well-defined task
- is invoked (called) by another user program
- returns control to the calling program when finished

Like a service routine, but not part of the OS
- not concerned with protecting hardware resources
- no special privilege required

Reasons for subroutines:
- reuse useful (and debugged!) code without having to keep typing it in
- divide task among multiple programmers
- use vendor-supplied library of useful routines
JSR Instruction

Jumps to a location (like a branch but unconditional), and saves address (or label) of next instruction in R7.

- saving the return address is called “linking”
JSRR Instruction

Just like JSR, except Register addressing mode.
- target address is Base Register

What important feature does JSRR provide that JSR does not?
Returning from a Subroutine

RET (JMP R7) gets us back to the calling routine.

– just like TRAP does it
Example: Negate the value in R0

\[ \text{2sComp} \quad | \quad \text{NOT} \quad R0, \quad R0 \quad ; \quad \text{flip bits} \]
\[ \quad \text{ADD} \quad R0, \quad R0, \quad \#1 \quad ; \quad \text{add one} \]
\[ \quad \text{RET} \quad ; \quad \text{return to caller} \]

; need to compute \( R4 = R1 - R3 \)
\[ \quad \text{ADD} \quad R0, \quad R3, \quad \#0 \quad ; \quad \text{copy R3 to R0} \]
\[ \quad \text{JSR} \quad \text{2sComp} \quad ; \quad \text{negate} \]
\[ \quad \text{ADD} \quad R4, \quad R1, \quad R0 \quad ; \quad \text{add to R1} \]
\[ \ldots \]

Note: Caller should save R0 if we’ll need it later!
Passing Information to/from Subroutines

Arguments

- A value passed in to a subroutine is called an argument.
- This is a value needed by the subroutine to do its job.
- Examples:
  - In 2sComp routine, R0 is the number to be negated.
  - In OUT service routine, R0 is the character to be printed.
  - In PUTS routine, R0 is address of string to be printed.

Return Values

- A value passed out of a subroutine is called a return value.
- This is the value that you called the subroutine to compute.
- Examples:
  - In 2sComp routine, negated value is returned in R0.
  - In GETC service routine, character read from the keyboard is returned in R0.
Using Subroutines

In order to use a subroutine, a programmer must know:

- its address (or at least a label that will be bound to its address)
- its function (what does it do?)
  
  **NOTE:** The programmer does not need to know how the subroutine works, but what changes are visible in the machine’s state after the routine has run.

- its arguments (where to pass data in, if any)
- its return values (where to get computed data, if any)
Saving and Restore Registers

Since subroutines are just like service routines, we also need to save and restore registers, if needed.

Generally use “callee-save” strategy, except for return values.

- Save anything that the subroutine will alter internally that shouldn’t be visible when the subroutine returns.
- It’s good practice to restore incoming arguments to their original values (unless overwritten by return value).

*Remember*: You MUST save R7 if you call any other subroutine or service routine (TRAP).

- Otherwise, you won’t be able to return to caller.
Subtraction (Sub)

\[ x = a - b \]

\[ \text{R0} \quad \text{R1} \quad \text{R2} \]

**Sub** → ST R2, SubSaveR2

NOT R2, R2

ADD R2, R2, 1

ADD R0, R1, R2

LD R2, SubSaveR2

RET

SubSaveR2 .FILL 0

JSR Sub
Example

(1) Write a subroutine **FirstChar** to:
- find the **first** occurrence of a particular **character** (in **R0**)
in a **string** (pointed to by **R1**);
return **pointer** to character or to end of string (NULL) in **R2**.

(2) Use **FirstChar** to write **CountChar**, which:
- counts the **number** of occurrences of a particular **character** (in **R0**)
in a **string** (pointed to by **R1**);
return **count** in **R2**.

Can write the second subroutine first, without knowing the implementation of **FirstChar**!
CountChar Algorithm (using FirstChar)

1. save regs
2. call FirstChar
3. \( R3 \leftarrow M[R2] \)
4. If \( R3 \neq 0 \):
   - no
   - go back to \( R3 \leftarrow M[R2] \)
5. If \( R3 = 0 \):
   - yes
   - \( R1 \leftarrow R2 + 1 \)
   - save \( R7 \), since we're using JSR
   - restore regs
   - return
CountChar Implementation

; CountChar: subroutine to count occurrences of a char

CountChar

ST R3, CCR3 ; save registers
ST R4, CCR4
ST R7, CCR7
ST R1, CCR1
AND R4, R4, #0 ; use for count

FirstChar

JSR FirstChar ; JSR alters R7
LDR R3, R2, #0 ; save original string ptr
BRz CC2 ; initialize count to zero
ADD R4, R4, #1 ; find next occurrence (ptr in R2)
ADD R1, R2, #1 ; see if char or null
BRnzp CC1 ; if null, no more chars

CC1
ADD R2, R4, #0 ; increment count
LD R3, CCR3 ; point to next char in string
LD R4, CCR4
LD R1, CCR1
LD R7, CCR7 ; move return val (count) to R2

CC2 ; restore regs

RET ; and return
FirstChar Algorithm

R0 - char
R1 - address of string
R2 - address of 1st char

save regs

R2 <- R1

R3 <- M[R2]

R3=0

R3=R0

yes

no

R2 <- R2 + 1

yes

restore regs

return
FirstChar Implementation

; FirstChar: subroutine to find first occurrence of a char

FirstChar

ST    R3, FCR3 ; save registers
ST    R4, FCR4 ; save original char
NOT   R4, R0  ; negate R0 for comparisons
ADD   R4, R4, #1
ADD   R2, R1, #0 ; initialize ptr to beginning of string
LDR   R3, R2, #0 ; read character
BRz   FC2 ; if null, we’re done
ADD   R3, R3, R4 ; see if matches input char
BRz   FC2 ; if yes, we’re done
ADD   R2, R2, #1 ; increment pointer
BRnzp FC1

FC1

FC2

LD    R3, FCR3 ; restore registers
LD    R4, FCR4
RET ; and return
Library Routines

Vendor may provide object files containing useful subroutines
- don’t want to provide source code -- intellectual property
- assembler/linker must support EXTERNAL symbols (or starting address of routine must be supplied to user)

```assembly
... .EXTERNAL SQRT ...
LD R2, SQAddr ; load SQRT addr
JSRR R2 ...
SQAddr .FILL SQRT ...
```

Using JSRR, because we don’t know whether SQRT is within 1024 instructions.