LC-3
Assembly Language
(Ch7)
LC-3 is a load/store RISC architecture

• Has 8 general registers
• Has a flat 16-bit addressing range
• Has a 16-bit word size
• Load variables from memory to register
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.
Labels

- Symbolic names that are used to identify memory locations
- Location for target of a branch or jump
- Location for a variable for loading and storing
- Can be 1-20 characters 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>
.ORIG

• Tells simulator where to put your code in memory
• Only one allowed per program
• PC gets set to this address at start up
• Similar to the “main” in “C”
“C”

- type
- varname;

- type is
  - int (integer)
  - char (character)
  - float (floating point)

“LC-3”

- varname .FILL value

- value is required – the initial value
LC-3 Syntax

```
.add R1, R2, R3

flag .FILL   x0001
counter .FILL x0002
letter  .FILL x0041 ; A
letters .FILL x4241 ; BA
```

- 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
.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
Bob .FILL 1
;set aside 1 location, label and initialize to 4.
Num .BLKW 1  #4
.STRINGZ

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

Example:

```
hello .STRINGZ "Hello World!"
```
.END

- Tells the assembler where your program ends
- Only one allowed in your program
<table>
<thead>
<tr>
<th>&quot;LC-3&quot;</th>
<th>&quot;C&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, X</td>
<td></td>
</tr>
<tr>
<td>LD R2, Y</td>
<td></td>
</tr>
<tr>
<td>ADD R3, R2, #0</td>
<td>Z = Y</td>
</tr>
<tr>
<td>ADD R3, R1, R2</td>
<td>Z = X + Y</td>
</tr>
<tr>
<td>??</td>
<td>Z = X - Y</td>
</tr>
<tr>
<td>??</td>
<td>Z = X * Y</td>
</tr>
<tr>
<td>??</td>
<td>Z = X / Y</td>
</tr>
<tr>
<td>ST R3, Z</td>
<td></td>
</tr>
</tbody>
</table>

An immediate is a value specified in an instruction, not by a .FILL declaration.
Simple LC-3 program

R2 = 0
R0 = 4
R1 = 2
R2 = 0 + 4 = 4
R1 = R1 - 1 = 1
R2 = 8
R1 = 0

• What does this program do?
• What is in “Result” at the end?
Program Execution

• Assembler translates to executable – machine language
• Linker combines multiple LC-3 files – if any
• Loader puts executable into memory and makes the CPU jump to first instruction, .ORIG.  \[ PC = 0x2000 \]
• Executes
• When executing is done returns control to OS
  • Or simulator or monitor
• Load again to run again with different data
  • In this case, assemble again, too, since data is in program.

HLL – if/else statements...

if (condition)
    statement;
else
    statement;
“C” if (count < 0)
    count = count + 1;

“LC-3”

\[
\begin{align*}
\text{ADD} & \quad R0, R0, 0 \\
\text{LD} & \quad R0, \text{count} \\
\text{BRpz} & \quad \text{greatzero} \\
\text{ADD} & \quad R0, R0, \#1 \\
\end{align*}
\]

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

“C”

while (count > 0)
{
    a = a + count;
    count--;
}
"LC-3"

while BRnz

LD
LD
ADD
ADD
BR
ST
ST

R1, a
R0, count
R1, R1, R0
R0, R0, #-1
while
R1, a
R0, count
Procedure Calls

Simple procedure calls require 2 instructions:

“JSR” or “JSRR” Jump Service Routine
- Saves the return address into R7

“RET” Jump Return
- Be careful with registers!!
- Cannot nest unless R7 is saved elsewhere
- Cannot be recursive without a stack

R7 = PC
Example

JSR Sub ; calls procedure

... ; calculate R2 = R0-R1
Sub NOT
ADD
ADD RET

R2, R1
R2, R2, #1
R2, R2, R0
; returns to line after
; JSR Sub
Repeat loops

“C”

/* do statement while expression is TRUE */
/* when expression is FALSE, exit loop */
do {
    if (a < b)
        a++;
    if (a > b)
        a--;
} while (a != b)
while

while (1 == 0)
foo();

does not guarantee the loop to run

do-while

do
foo();
while (1 == 0);

1.5
"LC-3"

```
LD   RO, a
LD   R1, b
JSR  Sub ; R2 = R0-R1
repeat
BRpz
ADD  R0, R0, #1
secondif
BRnz
JSR  Sub
until
ADD  R0, R0, #-1
until
JSR  Sub
BRnp
repeat
```
For loops

"C"

```c
for ( l = 3; l <= 8; l++ )
    { a = a+l; }
```

3 4 5 6 7 8
"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
ADD R2, R1, #-8
BRp endfor
ADD R0, R0, R1 ; a=a+l
ADD R1, R1, #1 ; l++
BR for ; same as BRnzp
endfor

3 - 8 = -5
4 - 8 = -4
8 - 8 = 0
9 - 8 = 1

R2 = R1 - 8
TRAP
(System Calls)

• Very tedious and dangerous for a programmer to deal with IO at the OS level.

• Need an instruction though to get the attention of the OS.

Use the “TRAP” instruction and a “trap vector”.
<table>
<thead>
<tr>
<th>Trap Vector</th>
<th>Assembler Name</th>
<th>Usage &amp; Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>GETC</td>
<td>Read a character from console into R0, not echoed.</td>
</tr>
<tr>
<td>0x21</td>
<td>OUT</td>
<td>Write character in R0 to console.</td>
</tr>
<tr>
<td>0x22</td>
<td>PUTS</td>
<td>Write string of characters to console. Start with character at address contained in R0. Stops when 0x0000 is encountered.</td>
</tr>
<tr>
<td>0x23</td>
<td>IN</td>
<td>Print a prompt to console and read in a single character into R0. Character is echoed.</td>
</tr>
<tr>
<td>0x24</td>
<td>PUTSP</td>
<td>Write a string of characters to console, 2 characters per address location. Start with characters at address in R0. First [7:0] and then [15:0]. Stops when 0x0000 is encountered.</td>
</tr>
<tr>
<td>0x25</td>
<td>HALT</td>
<td>Halt execution and print message to console.</td>
</tr>
</tbody>
</table>

**Example:**

```
trap 0x20
g etc
"Hello world"
```
To print a character
    ; the char must be in R0.
    TRAP x21

    or
    OUT 'a'

To read in a character
    ; will go into R0, no echo.
    TRAP x20

    or
    GETC
To end your program:

TRAP x25
or
HALT
Midterm in One Week
Review Next Class (should take most of class time)
Will go over most of the problem types expected

No homework this week due to Midterm

ALU Lab Extended until the 19th

Solutions for Homeworks and Quizzes should go up tomorrow
System Calls (TRAPS) and Subroutines

Ch. 9
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
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; place in PC

- How to get back
  - saves address of next instruction (current PC) in R7 before changing PC
NOTE: PC has already been incremented during instruction fetch stage.
RET (JMP R7)

How do we transfer control back to instruction following the TRAP?

• Save old PC 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?
BRzEXIT ; 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, x1452, 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
RET ; back to user

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

stored in table, location x21
## 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
Example

```
LEA R3, Binary
LD R6, ASCII ; char->digit template
LD R7, COUNT ; initialize to 10
AGAIN
TRAP x23 ; Get char
ADD R0, R0, R6 ; convert to number
STR R0, R3, #0 ; store number
ADD R3, R3, #1 ; incr pointer
ADD R7, R7, -1 ; decr counter
BRp AGAIN ; more?
BRnzp NEXT
```

ASCII
.FILL xFFD0
COUNT
.FILL #10
Binary
.BLKW 10

What's wrong with this code? What happens to R7?
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?
  – Sure, PUTS calls OUT

If so, is there anything special the calling service routine must do?
  – Better save 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 current PC (addr of next instruction) in R7.

- saving the return address is called “linking”
- target address is PC-relative ($PC + \text{Sext}(IR[10:0])$)
- bit 11 specifies addressing mode
  - if =1, PC-relative: target address = $PC + \text{Sext}(IR[10:0])$
  - if =0, register: target address = contents of register IR[8:6]
NOTE: PC has already been incremented during instruction fetch stage.
JSRR Instruction

Just like JSR, except Register addressing mode.
- target address is Base Register
- bit 11 specifies addressing mode

What important feature does JSRR provide that JSR does not?
NOTE: PC has already been incremented during instruction fetch stage.
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

2sComp
   NOT R0, R0 ; flip bits
   ADD R0, R0, #1 ; add one
   RET ; return to caller

To call from a program (within 1024 instructions):

; need to compute R4 = R1 - R3
   ADD R0, R3, #0 ; copy R3 to R0
   JSR 2sComp ; negate
   ADD R4, R1, R0 ; add to R1
   ...

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)

\[ \text{int foo}(a, b) \]
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.
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 ← M[R2]
4. R1 ← R2 + 1
5. If R3 = 0, yes, go to the next step. If R3 = 0, no, go back to call FirstChar.
6. save R7, since we're using JSR
7. restore regs
8. return
CountChar Implementation

; CountChar: subroutine to count occurrences of a char

CountChar

ST R3, CCR3 ; save registers
ST R4, CCR4 ; use for count
ST R7, CCR7 ; JSR alters R7
ST R1, CCR1 ; save original string ptr
AND R4, R4, #0 ; initialize count to zero

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

CC2 ADD R2, R4, #0 ; move return val (count) to R2
LD R3, CCR3 ; restore regs
LD R4, CCR4
LD R1, CCR1
LD R7, CCR7
RET ; and return
FirstChar Algorithm

save regs

R2 <- R1

R3 <- M[R2]

R3=0

R3=R0

R2 <- R2 + 1

restore regs

return

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

FC1
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

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.