System Calls (TRAPS) and Subroutines

Ch. 9
Midterm Solutions today
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. \[ \text{putc} \]
2. Operating system code performs operation.
3. Returns control to user program.

In LC-3, this is done through the \text{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
TRAP

NOTE: PC has already been incremented during instruction fetch stage.
8-bit +
16-bit +

... 00000000
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?
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

Maxwell James Dunne – Spring 2016
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
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
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 -- “calleesave”
- 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 -- “callersave”
- 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.
<table>
<thead>
<tr>
<th>Food Ordered</th>
<th>Food Delivered</th>
<th>Saved</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 \((\text{PC} + \text{Sext}([10:0]))\)
- bit 11 specifies addressing mode
  - if =1, PC-relative: target address = \(\text{PC} + \text{Sext}([10:0])\)
  - if =0, register: target address = contents of register \([8:6]\)
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?

16-bit space
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

\[
\begin{align*}
2\text{sComp} & : \quad \text{NOT R0, R0} \quad ; \quad \text{flip bits} \\
\text{ADD} & : \quad \text{R0, R0, #1} \quad ; \quad \text{add one} \\
\text{RET} & : \quad \text{return to caller}
\end{align*}
\]

To call from a program (within 1024 instructions):

\[
\begin{align*}
; \text{need to compute } R4 = R1 - R3 \\
\text{ADD} & : \quad \text{R0, R3, #0} \quad ; \quad \text{copy R3 to R0} \\
\text{JSR} & : \quad 2\text{sComp} \quad ; \quad \text{negate} \\
\text{ADD} & : \quad \text{R4, R1, R0} \quad ; \quad \text{add to R1} \\
\ldots
\end{align*}
\]

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)

\[ x \times 8000 \]
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 \texttt{FirstChar} to:

\texttt{Print( )}

- find the \texttt{first} occurrence of a particular \texttt{character} (in R0) in a \texttt{string} (pointed to by R1);
- return \texttt{pointer} to character or to end of string (NULL) in R2.

(2) Use \texttt{FirstChar} to write \texttt{CountChar}, which:

- counts the \texttt{number} of occurrences of a particular \texttt{character} (in R0) in a \texttt{string} (pointed to by R1);
- return \texttt{count} in R2.

Can write the second subroutine first, without knowing the implementation of \texttt{FirstChar}!
CountChar Algorithm (using FirstChar)

\[
R0 = '1'
\]

- \text{save regs}

- \text{call FirstChar}

- \text{R3} \leftarrow M[R2]

- 'hello world'

- \text{R3=0}
  - no
  - \text{restore regs}
  - return

- yes

- \text{R1} \leftarrow R2 + 1

- save R7
  - since we're using JSR

\text{restore regs}

\text{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

cc1
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
ADD  R2, R4, #0  ; increment count
ADD  R1, R2, #1  ; point to next char in string
BRnzp  CC1
ADD  R3, R2, #0  ; move return val (count) to R2
ADD  R4, R4, #0
ADD  R1, R2, #0
ADD  R7, R2, #0  ; restore regs
RET  ; and return
FirstChar Algorithm

save regs

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

R2 <- R1

R3 <- M[R2]

R3=0

R3=R0

yes

no

R2 <- R2 + 1

restore regs

return

The algorithm saves the registers, then loads the address of the first character from the second register. It then loads the character from memory, and checks if it's zero. If it is, it returns. If not, it increments the second register and repeats the process. The third register is set to the fourth register, and the flow continues.
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.
```
integer division

\[
\frac{33}{5} = \frac{28}{5} = 23
\]
R3 = -s
R4 = 33

R4 < s

NO

R4 = R4 - s
R5 = R5 + 1

YES

done
43 / 4

Div

Uro 32

14
23
36
Memory and Data Structures

Arrays, Stacks, Queues

(Ch 10 & 16)
Memory

- This is the “RAM” in a system
- We have seen labels and addresses point to pieces of memory storing:
  - words
  - bytes
  - strings
  - numbers
- Memory is just a collection of bits
  - We could use it to represent integers
  - Or as an arbitrary set of bits
Treat memory as a giant array

- Compiler or programmer decides what use to make of it.
- The element numbering starts at 0
- The element number is an address
- In “C” to allocate some memory:

```c
char m[size_of_array];
```
Storage of Data

- LC-3 architecture is “word addressable” meaning that all addresses are “word” addresses.
- This means the smallest unit of memory we can allocate is 16-bits, a word.
- Use ‘LD’ (load) and ‘ST’ (store) to access this unit (or LDR & STR).
Example

mychar
newline

... 

LD R1, newline
GETC
ST R0, mychar
JSR Sub ; R2=R1-R0
BRz found_newline

... 

found_newline ...
The data is placed in memory like this at start up (assuming data section starts at address 1). The “mychar” variable will change to the value of the character entered by the user once stored.
Pointers and Arrays

We've seen examples of both of these in our LC-3 programs, let's see how these work in "C"

**Pointer**
- Address of a variable in memory
- Allows us to **indirectly** access variables
  - in other words, we can talk about its *address* rather than its *value*

**Array**
- A list of values arranged sequentially in memory
- Example: a list of telephone numbers
- Expression `a[4]` refers to the 5th element of the array `a`
Arrays

Array implementation is very important

- Most assembly languages have only basic concept of arrays (BLKW)
- From an array, any other data structure we might want can be built
Arrays

Properties of arrays:

- Each element is the same size
- Elements are stored contiguously
- First element at the smallest memory address

In assembly language we must

- Allocate correct amount of space for an array
- Map array addresses to memory addresses
Arrays

LC-3 declarations of arrays within memory

To allocate a portion of memory (more than a single variable’s worth)

```
variableName .BLKW numElements
```

numElements is just that, numbering starts at 0 (as in C)
Array of Integers

Calculating the address of an array element

```c
int myarray[7] /* C */

.BLLKW 7 ; LC-3
```

- If base address of "myarray" is 25

```
0 1 2 3 4 5 6
```

- Which is base address + distance from the first element

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>2A</td>
<td>2B</td>
</tr>
</tbody>
</table>
```

Element index

Address

Maxwell James Dunne – Spring 2016
How do you get the address of `myarray`?

- Use the “load effective address” instruction, “LEA”
- Keep clear the difference between an address and the contents of an address.
Addressing Byte Arrays

To get address of `myarray[4]` in LC-3, write the code...

```assembly
LEA R0, myarray
ADD R1, R0, 4
```

Now, if we wanted to increment element number 5 by 1...

```assembly
LDR R4, R1, 0
ADD R4, R4, 1
STR R4, R1, 0
```
Address vs. Value

Sometimes we want to deal with the **address** of a memory location, rather than the **value** it contains.

Recall example from Chapter 6: adding a column of numbers.
- **R2** contains address of first location.
- Read value, add to sum, and increment **R2** until all numbers have been processed.

**R2** is a pointer -- it contains the address of data we’re interested in.
2-Dimensional Arrays

2-Dimensional arrays are more complicated in assembly

- Memory is a 1-D array
- Must map 2-D array to 1-D array
- Arrays have rows and columns
  - $r \times c$ array
  - $r =$ rows
  - $c =$ columns
2-Dimensional Arrays

\[[N] = \text{start} \rightarrow N\]

Two sensible ways to map 2-D to 1-D

Row major form: (rows are all together)

<table>
<thead>
<tr>
<th>0,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
</tr>
<tr>
<td>1,0</td>
</tr>
<tr>
<td>1,1</td>
</tr>
<tr>
<td>2,0</td>
</tr>
<tr>
<td>2,1</td>
</tr>
<tr>
<td>3,0</td>
</tr>
<tr>
<td>3,1</td>
</tr>
</tbody>
</table>

Column major form: (columns are all together)

<table>
<thead>
<tr>
<th>0,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
</tr>
<tr>
<td>2,0</td>
</tr>
<tr>
<td>3,0</td>
</tr>
<tr>
<td>0,1</td>
</tr>
<tr>
<td>1,1</td>
</tr>
<tr>
<td>2,1</td>
</tr>
<tr>
<td>3,1</td>
</tr>
</tbody>
</table>
2-Dimensional Arrays

\[
\begin{bmatrix}
0 & 1 & 2 & 3 \\
1 & 4 & 5 & 6 \\
2 & 7 & 8 & 9 \\
\end{bmatrix}
\]

How do you calculate addresses in a 2-D array?

- **Row Major:**
  
  Address \((r_i, c_i) =\)
  
  Base Address + (((\(r_i\) \(\times\) Number of Cols) + \(c_i\)) \(\times\) Element size)

- **Column Major:**
  
  Address \((r_i, c_i) =\)
  
  Base Address + (((\(c_i\) \(\times\) Number of Rows) + \(r_i\)) \(\times\) Element size)
Fill x 5000
Fill x 4000
Fill x 2000
Summary of 2D arrays

• Row/Column major (storage order)
• Base address
• Size of elements
• Dimensions of the array

How about 3-D arrays?
Bounds Checking

- Many HLL’s have bounds checking (not C!!!)
- Assembly languages have no implied bounds checking
- Your program is in total control of memory
- With a 5 x 3 array, what does the following address?

```
array  .BLKW 15

LEA    R1, array
ADD    R1, R1, 15
LDR    R0, R1, 0

6th
```

- Bounds checking is often a good idea!!
- Most C development environments include optional bounds checking.
Heartbleed

Open SSL

\texttt{gimme(56)}

\uparrow

\texttt{gimme(1024)}
Stacks

A LIFO (last-in first-out) storage structure.
- The first thing you put in is the last thing you take out.
- The last thing you put in is the first thing you take out.

This means of access is what defines a stack, not the specific implementation.

Two main operations:
- **PUSH**: add an item to the stack
- **POP**: remove an item from the stack
A Physical Stack

Coin rest in the arm of an automobile

Initial State | After One Push | After Three More Pushes | After One Pop

First quarter out is the last quarter in.
A Hardware Implementation

Data items move between registers

Initial State

Empty: Yes

After One Push

Empty: No

After Three More Pushes

Empty: No

After Two Pops
A Software Implementation

Data items don't move in memory, just our idea about where the TOP of the stack is.

By convention, R6 holds the Top of Stack (TOS) pointer.
Delete Secret.txt
Basic Push and Pop Code

For our implementation, stack grows downward (when item added, TOS moves closer to 0)

**Push**
- ADD R6, R6, #-1 ; decrement stack ptr
- STR R0, R6, #0 ; store data (R0)

**Pop**
- LDR R0, R6, #0 ; load data from TOS
- ADD R6, R6, #1 ; increment stack ptr
Pop with Underflow Detection

If we try to pop too many items off the stack, an underflow condition occurs.

- Check for underflow by checking TOS before removing data.
- Return status code in R5 (0 for success, 1 for underflow)

```
[ POP ] LD R1, EMPTY ; EMPTY = -x4000
ADD R2, R6, R1 ; Compare stack pointer
BRz FAIL ; to x4000 to see if empty
LDR R0, R6, #0
ADD R6, R6, #1
AND R5, R5, #0 ; SUCCESS: R5 = 0
RET
FAIL AND R5, R5, #0 ; FAIL: R5 = 1
ADD R5, R5, #1
RET
EMPTY .FILL xC000 ; 2SC rep of -x4000
```
Push with Overflow Detection

If we try to push too many items onto the stack, an overflow condition occurs. This example assumes stack has room for 5 items.

- Check for overflow by checking TOS before adding data.
- Return status code in R5 (0 for success, 1 for overflow)

```
PUSH) LD R1, MAX ; MAX = -x3FFB
ADD R2, R6, R1 ; Compare stack pointer
BRz FAIL ; top address to see if full
ADD R6, R6, #-1
STR R0, R6, #0
AND R5, R5, #0 ; SUCCESS: R5 = 0
RET
FAIL AND R5, R5, #0 ; FAIL: R5 = 1
ADD R5, R5, #1
RET
MAX .FILL xC005 ; 2SC of -x3FFB
```
Stack Example

- Printing out a positive integer, character by character
- Push LSB to MSB
- Pop MSB to LSB (LIFO)

integer = 1024

if integer == 0 then
  push '0'
else
  while integer != 0
    digit ← integer mod base
    char ← digit + 48
    push char onto stack
  end while
  integer ← integer div base
end if

while stack is not empty
  pop char
  print char
Arithmetic Using a Stack

Instead of registers, some ISA’s use a stack for source and destination operations: a zero-address machine.

- Example:
  ADD instruction pops two numbers from the stack, adds them, and pushes the result to the stack.

Evaluating \((A+B) \cdot (C+D)\) using a stack:

1. push A
2. push B
3. ADD
4. push C
5. push D
6. ADD
7. MULTIPLY
8. pop result

Why use a stack?
- Limited registers.
- Convenient calling convention for subroutines.
- Algorithm naturally expressed using LIFO data structure.
Example: OpAdd

POP two values, ADD, then PUSH result.
Example: OpAdd

OpAdd

JSR POP ; Get first operand.
ADD R5,R5,#0 ; Check for POP success.
BRp Exit ; If error, bail.
ADD R1,R0,#0 ; Make room for second.
JSR POP ; Get second operand.
ADD R5,R5,#0 ; Check for POP success.
BRp Restore1 ; If err, restore & bail.
ADD R0,R0,R1 ; Compute sum.
JSR RangeCheck ; Check size.
BRp Restore2 ; If err, restore & bail.
JSR PUSH ; Push sum onto stack.
RET

Restore2 ADD R6,R6,#-1 ; Decr stack ptr (undo POP)

Exit RET

Resto1 ADD R6,R6,#-1 ; Decr stack ptr
Queues

A queue is a FIFO (First In, First Out).
- The classic analogy of a queue is a line.
  - Person gets on the end of the line (the Tail),
  - Waits,
  - Gets off at the front of the line (the Head).
- Getting into the queue is an operation called enqueue.
- Taking something off the queue is an operation called dequeue.
- It takes 2 pointers to keep track of the data structure,
  - Head (let’s use R5)
  - Tail always points to empty element (R6)
Initial state:

```
  ▲  Head (R5), and Tail (R6)
```

After 1 enqueue operation:

```
  X
  ▲  Head (R5)  ▲  Tail (R6)
```

After another enqueue operation:

```
  X  Y
  ▲  Head (R5)  ▲  Tail (R6)
```
After a dequeue operation:

```
X Y
```

Head (R5) ——— Tail (R6)

Like stacks, when an item is removed from the data structure, it is physically still present, but correct use of the structure cannot access it.
Implementation of a queue

Storage:

queue .BLKW infinity ; assume infinite for now
LEA R5, queue ; head
LEA R6, queue ; tail

Enqueue (item):

STR R0, R6, #0 ; R0 has data to store
ADD R6, R6, #1

Dequeue (item):

JSR SUB ; R0 = R5-R6
BRz queue_empty
LDR R1, R5, #0 ; put data in R1
ADD R5, R5, #1
Circular Queues

- To avoid infinite array, wrap around from end to beginning.
- Head == Tail means empty
- Head points to first item (for next dequeue)
- Tail point to empty location (for next enqueue)

Example of an 8 element circular queue
Circular Queues

After "enqueue'ing" one element

After "enqueue'ing" another element
After “dequeue’ing” an element
Circular Queues

Storage and initialization:

```
queue .BLKW queue_size
queue_end .BLKW 1
LEA R5, queue ; head
LEA R6, queue ; tail
```

Enqueue (item)

```
STR R0, R6, #0 ; data to enqueue is in R0
ADD R6, R6, #1
LEA R1, queue_end
JSR SUB ; R1 = R1 - R6
BRp continue1
LEA R6, queue ; wrap around
continue1
```
Dequeue (item):

JSR SUB ; R1 = R5 - R6
BRz queue_empty
LDR R0, R5, #0
ADD R5, R5, #1
LEA R1, queue_end
JSR SUB ; R1 = R5 - R1
BRn continue2
LEA R5, queue ; wrap around
continue2
Summary of data structures

- All data structures are based on the simple array.
- 2D Arrays, Stacks, Queues.
- It is all about the implementation.
- Bounds checking is important.
- If not documented can become confusing.