Input and Output

How things get into and out of the CPU
Computer System
I/O: Connecting to Outside World

• So far, we’ve learned how to:
  – compute with values in registers
  – load data from memory to registers
  – store data from registers to memory

• But where does data in memory come from?

• And how does data get out of the system so that humans can use it?
Types of I/O devices characterized by:

- **behavior**: input, output, storage
  
  - input: keyboard, motion detector, network interface
  - output: monitor, printer, network interface
  - storage: disk, CD-ROM

- **data rate**: how fast can data be transferred?
  
  - keyboard: 100 bytes/sec
  - Spinning disk: 30 MB/s or more
  - SSD: over 200 MB/s to Samsung 960 claims 3,500 MB/s
  - network: 1 Mb/s - 1 Gb/s
Hospital
 Year
 1 month
 3+ears

brain

7 info
durations
angles

90 wpm

Enter
I/O Devices

Keyboard

- User presses ‘A’ key -> ‘a’
- ASCII code is 0x61
- Keyboard sends this on wires
- 1 for start, 8-bits of data, 0 for stop
- ‘a’ is: 1011000010
- Buffer at computer catches these bits
Displays

- Character display works with the reverse process (sort of)
- Most displays today are “bit mapped”

Printers

- Just like a display but now being “printed” to paper, not a screen.
- Again, most printers are now “bit mapped” verses character.
Hard Disk

- A spinning disk (4600, 5200, 7200, 10000+ RPM)
- 20 – 8000 GB and growing FAST
- Magnetic and read/write (like tape)
- Both sides
- Usually a stack of platters
- Disk access

<table>
<thead>
<tr>
<th>Queuing</th>
<th>Seek</th>
<th>Rotation</th>
<th>Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depends</td>
<td>10ms</td>
<td>10ms</td>
<td>1ms</td>
</tr>
</tbody>
</table>

- Electronic speeds are in the nanoseconds (10-9 sec)
- Disk speeds are in the milliseconds (10-3 sec)
- Why use a disk? **cheap**
- Replaced by SSD’s for most non-bulk storage
Questions

- How does CPU ask for a char to be printed?
- Which printer?
- Which display? Who’s?
- When is printer ready for the next char?
- When does keyboard have the next char?
- What about the million times slower?
LC-3 I/O

Uses the “TRAP” instruction to invoke the Operating System, which in turns talks to the hardware.

User specifies the type of operation desired by giving a code to the OS.

Ex. “TRAP x20” gets a character from the keyboard.
• Don’t use “JSR” because
  • OS doesn’t trust user to provide the correct address
  • Want to switch into OS mode, where more things are allowed
• CPU sees the “TRAP” instruction and uses the trap vector to determine where to go in the OS code.
• OS will not allow (or should not)
  • Users to read each other’s keyboards
  • Users to access all memory or disk locations
I/O Controller

- **Control/Status Registers**
  - CPU tells device what to do -- write to control register
  - CPU checks whether task is done -- read status register

**Data Registers**
- CPU transfers data to/from device

---

**Diagram**

- **CPU**
  - Control/Status
  - Output Data
  - **Graphics Controller**
    - **Electronics**
  - **display**

---

**Device electronics**
- performs actual operation
  - pixels to screen, bits to/from disk, characters from keyboard

Maxwell James Dunne
Programming Interface

How are device registers identified?
  - Memory-mapped vs. special instructions

How is timing of transfer managed?
  - Asynchronous vs. synchronous

Who controls transfer?
  - CPU (polling) vs. device (interrupts)
Memory-Mapped vs. I/O Instructions

**Special Instructions**
- designate opcode(s) for I/O
- register and operation encoded in instruction

**Memory-mapped**
- assign a memory address to each device register
- use data movement instructions (LD/ST) for control and data transfer
Which is better?

864 + 5 256 16

- What is the problem with having special instructions for IO?
- What happens if a new device is created?

257th

Memory mapped is much more flexible and expandable.

3 - 4K
Memory Mapped IO

- Idea is to place devices other than RAM chips at physical address locations.
- This way to access IO devices you use the same load and store instructions.
Design hardware and software to recognize certain addresses

Real Memory - RAM

0x00000000

From keyboard

0xffffffff0000

To display

0xffffffff0008

0xffffffff0010
• Devices on bus watch for their address
• But is there a new char to read?
• But is the display done with the last char?
Linux = file
Need I/O device status to coordinate

Real Memory - RAM

0x00000000

0xffffffff

0xffffffff0008
DATA from keyboard

0xffffffff000c
STATUS from keyboard

0xffffffff0010
DATA to Display

0xffffffff0014
STATUS from Display
Transfer Timing

I/O events generally happen much slower than CPU cycles.

**Synchronous**

- data supplied at a fixed, predictable rate
- CPU reads/writes every X cycles

**Asynchronous**

- data rate less predictable
- CPU must synchronize with device, so that it doesn’t miss data or write too quickly
Transfer Control

Who determines when the next data transfer occurs?

Polling
- CPU keeps checking status register until new data arrives OR device ready for next data
- “Are we there yet? Are we there yet? Are we there yet?”

Interrupts
- Device sends a special signal to CPU when new data arrives OR device ready for next data
- CPU can be performing other tasks instead of polling device.
- “Wake me when we get there.”
## LC-3

### Memory-mapped I/O (Table A.3)

<table>
<thead>
<tr>
<th>Location</th>
<th>I/O Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xFE00</td>
<td>Keyboard Status Reg (KBSR)</td>
<td>Bit [15] is one when keyboard has received a new character.</td>
</tr>
<tr>
<td>xFE02</td>
<td>Keyboard Data Reg (KBDR)</td>
<td>Bits [7:0] contain the last character typed on keyboard.</td>
</tr>
<tr>
<td>xFE04</td>
<td>Display Status Register (DSR)</td>
<td>Bit [15] is one when device ready to display another char on screen.</td>
</tr>
<tr>
<td>xFE06</td>
<td>Display Data Register (DDR)</td>
<td>Character written to bits [7:0] will be displayed on screen.</td>
</tr>
</tbody>
</table>

Asynchronous devices
- synchronized through status registers

Polling and Interrupts
- the details of interrupts is in chapter 10
Input from Keyboard

When a character is typed:

- its ASCII code is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
- the “ready bit” (KBSR[15]) is set to one
- keyboard is disabled -- any typed characters will be ignored

When KBDR is read:

- KBSR[15] is set to zero
- keyboard is enabled
Basic Input Routine

Polling

new char?

[...]

- POLL  LDI  R0,  KBSRPtr
- BRzp  POLL
- LDI  R0,  KBDRPtr

[...]

KBSRPtr  .FILL  xFE00
KBDRPtr  .FILL  xFE02

Spinning beachball
Output to Monitor

When Monitor is ready to display another character:
- the “ready bit” (DSR[15]) is set to one

When data is written to Display Data Register:
- DSR[15] is set to zero
- character in DDR[7:0] is displayed
- any other character data written to DDR is ignored (while DSR[15] is zero)
Basic Output Routine

- **Polling**
- **screen ready?**
  - NO
  - YES

**write character**

- Poll LDI R1, DSRPtr
- BRzp POLL
- STI R0, DDRPtr

- DSRPtr .FILL xFE04
- DDRPtr .FILL xFE06
40 MHz

UART+ ⏞ 115200 bits
Keyboard Echo Routine

Usually, input character is also printed to screen.
- User gets feedback on character typed and knows its ok to type the next character.

```
POLL1  LDI  R0, KBSRPtr
       BRzp POLL1
   LDI  R0, KBDRPtr
POLL2  LDI  R1, DSRPtr
       BRzp POLL2
       STI  R0, DDPtr
       ...
KBSRPtr  .FILL  xFE00
KBDRPtr  .FILL  xFE02
DSRPtr  .FILL  xFE04
DDRPtr  .FILL  xFE06
```
Polling

• How much time is spent spinning?
• A **PUTC** or **GETC** is less than 10 instructions, or ~10ns on a modern processor
• Mechanical devices take milliseconds
• Almost all time is spent spinning
  • Would be nice to do useful work while waiting
  • Periodically poll devices and send characters when ready
Polling I/O

- The OS must check regularly (poll) for ready devices
  - Perhaps once a millisecond
- If ready, then OS services device
  - Keyboard: transfer character and put on queue
  - Display: transmit character to the graphics HW
Polling I/O

Problems:
- How often to poll?
- How does the OS code get run?
- What happens to the user program?

Is there a better solution?
Exceptions and the OS

STOP!!! Do this!!!
Interrupts

“External condition related to IO devices”

- Have device tell OS/CPU it is ready
- Requires hardware to support.
- OS/CPU can then interrupt what it is doing to:
  - Determine what device wants service
  - Determine what service it wants
  - Perform or start the service
  - Go back to what OS/CPU was doing
mult

Keyboard
Examples of Interrupts

- Disk drive at sector/track position (old days)
- Mouse moved
- Keyboard key pressed
- Printer needs data
- Video card wants memory access
- Network sending or receiving
- USB scanner has data
Interrupt Properties

- They arrive asynchronously
- Can’t communicate with a running program (no args or return values)
- They are associated with various priorities
- You want to handle them soon (interrupt latency)
- (usually) want to resume program
Trap

“Internal conditions related to instruction stream”

Trap Examples:

→ “TRAP” instruction
  • Illegal instruction
  • Arithmetic overflow
  • Divide by zero
  • “LD” from illegal/protected memory address

3/0
$\text{TRAP} \rightarrow \text{RESET}$
Trap Properties

- They arrive synchronously
- Get trap in same place if you re-run program
- They are associated with various priorities
- Must handle immediately
- Want to resume program (usually)
Exceptions

“Mechanism used to handle both interrupts and traps”

• HW handles initial reaction
• Then invokes SW called an “Exception Handler” to take care of the interrupt/trap
LC-3 Exceptions

- LC-3 only has the concept of IO related exceptions, interrupts.
- Does not have any mechanisms to deal with illegal instructions, or arithmetic overflow.
- Remember the LC-3 is just a learning aid, not real hardware.
LC-3 Interrupt-Driven I/O

External device can:

1. Force currently executing program to stop;
2. Have the processor satisfy the device’s needs; and
3. Resume the stopped program as if nothing happened.

Why do interrupts?

- Polling consumes a lot of cycles, especially for rare events – these cycles can be used for more computation.
- Example: Process previous input while collecting current input. (See Example 8.1 in text.)
To implement an interrupt mechanism, we need:

- A way for the I/O device to **signal** the CPU that an interesting event has occurred.
- A way for the CPU to **test** whether the **interrupt signal is set** and whether its **priority is higher** than the current program.

**Generating Signal**

- Software sets "**interrupt enable**" bit in device register.
- When ready bit is set and IE bit is set, interrupt is signaled.
Priority

Every instruction executes at a stated level of urgency.

**LC-3: 8 priority levels (PLO-PL7)**

- Example:
  - Payroll program runs at PL0.
  - Nuclear power correction program runs at PL7.

- It’s OK for PL6 device to interrupt PL0 program, but not the other way around.

**Priority encoder** selects highest-priority device, compares to current processor priority level, and generates interrupt signal if appropriate.
Testing for Interrupt Signal

- CPU looks at signal between STORE and FETCH phases.
- If not set, continues with next instruction.
- If set, transfers control to interrupt service routine.
Operating Systems

• The “program” that launches the user programs and deals with exceptions.
• How does the operating system deal with such things as simultaneous exceptions?
• What happens on machines that have many users “on” at once?

Time sharing
Multiple Exceptions

Problem: How about multiple simultaneous exceptions?

Solution: Have priorities in HW / SW

- Handle highest-priority exception first
- Equal priority exceptions handled arbitrarily
- Give higher priority
  - more serious (e.g., power failing)
  - can’t wait long (e.g., rotating disk)
How about exceptions during exception handling?

Option #1: Make it wait until done with first exception
  - May be a bad idea for higher priority exception

Option #2: Make exception handler re-entrant
  - Make able to handle multiple active calls
  - Allow higher-priority exceptions to bypass lower-priority exception currently being serviced
Memory and Data Structures

Arrays, Stacks, Queues

(Ch 10 & 16)
Midterm Overview

• May 11th, 2017 1:30PM-3:05PM
  – Scheduled for whole class
• No books, notes or calculators
• Show your work.
  – No credit if we don’t know how you did something
  – Lots of partial credit
• Practice midterm on canvas
• Seating chart
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:

  ```
  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 .BLKW 1
newline .FILL xA

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
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

α[5] → x3 008A
LC-3 declarations of arrays within memory

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

`variable name .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 */
```

- If base address of “myarray” is 25

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

- Which is base address + distance from the first elem
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.
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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3107</td>
<td>x3100</td>
</tr>
<tr>
<td>x2819</td>
<td>x3101</td>
</tr>
<tr>
<td>x0110</td>
<td>x3102</td>
</tr>
<tr>
<td>x0310</td>
<td>x3103</td>
</tr>
<tr>
<td>x0100</td>
<td>x3104</td>
</tr>
<tr>
<td>x1110</td>
<td>x3105</td>
</tr>
<tr>
<td>x11B1</td>
<td>x3106</td>
</tr>
<tr>
<td>x0019</td>
<td>x3107</td>
</tr>
</tbody>
</table>

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

Maxwell James Dunne
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
  - \( r = \text{rows} \)
  - \( c = \text{columns} \)
Two sensible ways to map 2-D to 1-D

Row major form: (rows are all together)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1</td>
</tr>
<tr>
<td>0,1</td>
<td>2</td>
</tr>
<tr>
<td>1,0</td>
<td>3</td>
</tr>
<tr>
<td>1,1</td>
<td>4</td>
</tr>
<tr>
<td>2,0</td>
<td>5</td>
</tr>
<tr>
<td>2,1</td>
<td>6</td>
</tr>
<tr>
<td>3,0</td>
<td>7</td>
</tr>
<tr>
<td>3,1</td>
<td>8</td>
</tr>
</tbody>
</table>

Column major form: (columns are all together)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1</td>
</tr>
<tr>
<td>1,0</td>
<td>3</td>
</tr>
<tr>
<td>2,0</td>
<td>5</td>
</tr>
<tr>
<td>3,0</td>
<td>7</td>
</tr>
<tr>
<td>0,1</td>
<td>2</td>
</tr>
<tr>
<td>1,1</td>
<td>4</td>
</tr>
<tr>
<td>2,1</td>
<td>6</td>
</tr>
<tr>
<td>3,1</td>
<td>8</td>
</tr>
</tbody>
</table>

4x2 example
3 \times 2 + 1 = 7

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

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

- **Column Major:**
  
  $\text{Address (} r_i, c_i \text{)} = \text{Base Address} + (((c_i \times \text{Number of Rows}) + r_i) \times \text{Element size})$
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
```

```
<table>
<thead>
<tr>
<th>LEA</th>
<th>R1, array</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R1, R1, 15</td>
</tr>
<tr>
<td>LDR</td>
<td>R0, R1, 0</td>
</tr>
</tbody>
</table>
```

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

\[ S00 > "A \]
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

After One Push

After Three More Pushes

After Two Pops
A Software Implementation

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

Initial State  | After One Push  | After Three More Pushes  | After Two Pops
\hspace{1cm} x4000 R6 \hspace{1cm} x3FFF R6 \hspace{1cm} x3FFC R6 \hspace{1cm} x3FFE R6

By convention, R6 holds the Top of Stack (TOS) pointer.
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)

```python
integer = 1024

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR POP</td>
<td>Get first operand.</td>
</tr>
<tr>
<td>ADD R5,R5,#0</td>
<td>Check for POP success.</td>
</tr>
<tr>
<td>BRp Exit</td>
<td>If error, bail.</td>
</tr>
<tr>
<td>ADD R1,R0,#0</td>
<td>Make room for second.</td>
</tr>
<tr>
<td>JSR POP</td>
<td>Get second operand.</td>
</tr>
<tr>
<td>ADD R5,R5,#0</td>
<td>Check for POP success.</td>
</tr>
<tr>
<td>BRp Restore1</td>
<td>If err, restore &amp; bail.</td>
</tr>
<tr>
<td>ADD R0,R0,R1</td>
<td>Compute sum.</td>
</tr>
<tr>
<td>JSR RangeCheck</td>
<td>Check size.</td>
</tr>
<tr>
<td>BRp Restore2</td>
<td>If err, restore &amp; bail.</td>
</tr>
<tr>
<td>JSR PUSH</td>
<td>Push sum onto stack.</td>
</tr>
<tr>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>ADD R6,R6,#-1</td>
<td>Decr stack ptr (undo POP)</td>
</tr>
<tr>
<td>RET</td>
<td>Decr stack ptr</td>
</tr>
<tr>
<td>Restore2</td>
<td></td>
</tr>
<tr>
<td>Restore1</td>
<td></td>
</tr>
</tbody>
</table>
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 a 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
After "enqueue'ing" one element

After "enqueue'ing" another element
After “dequeue’ing” an element
Storage and initialization:

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

Enqueue (item)

```assembly
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.