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).
Storage of bytes

Example

mychar .BLKW 1
newline .FILL xA
...
...
LD R1, newline
GETC
ST R0, mychar
JSR Sub ; R2=R1-R0
BRz found_newline
...
found_newline ...
Storage of bytes

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
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 */
myarray .BLKW 7 ; LC-3
```

- If base address of “myarray” is 25

- Which is base address + distance from the first element
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 \texttt{myarray[4]} in LC-3, write the code...

\begin{verbatim}
LEA    R0, myarray
ADD    R1, R0, 4
\end{verbatim}

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

\begin{verbatim}
LDR    R4, R1, 0
ADD    R4, R4, 1
STRU   R4, R1, 0
\end{verbatim}
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
  - \( r = \text{rows} \)
  - \( c = \text{columns} \)
2-Dimensional Arrays

[[r][c]]

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

Row major form:
(rows are all together)

```
0,0
0,1
1,0
1,1
2,0
2,1
3,0
3,1
```

Column major form:
(columns are all together)

```
0,0
1,0
2,0
3,0
0,1
1,1
2,1
3,1
```
2-Dimensional Arrays

\[
\text{Cr} \left\lbrack \begin{array}{c}
\text{Cn}
\end{array} \right\rbrack
\]

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

- **Row Major:**

  \[
  \text{Address} (r_i, c_i) = \text{Base Address} + (((r_i \times \text{Number of Cols}) + c_i) \times \text{Element size})
  \]

- **Column Major:**

  \[
  \text{Address} (r_i, c_i) = \text{Base Address} + (((c_i \times \text{Number of Rows}) + r_i) \times \text{Element size})
  \]
\[
\begin{align*}
A & \quad \text{RM} \\
0 & \quad \frac{1}{2} \\
3 & \quad 4 \quad 5 \\
6 & \quad 7 \quad 8
\end{align*}
\]

\[
\begin{align*}
A[0][0] & = (0 \cdot 3 + 0) \cdot 1 - 0 \\
& = 0 \\
A[0][2] & = (0 \cdot 3 + 2) = 2 \\
A[2][1] & = (2 \cdot 3 + 1) = 7
\end{align*}
\]
Chart: \[ [8] = 8 \]
Short: \[ [8] = 16 \]
Int: \[ [8] = 32 \]
2-Dimensional Arrays

\[ r \cdot \text{colwidth} + c \cdot \text{2width} + z \]

Summary of 2D arrays

\[ r \cdot \text{colwidth} + c \]

- 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
```

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

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
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)

```plaintext
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
    endwhile

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)
Restore1 ADD R6,R6,#-1 ; Decr stack ptr
Exit RET
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)
Queues

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:

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
Circular Queues

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
```
Circular Queues

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.