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

```assembly
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`
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
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
• Address of myarray[4] = 25 + 4 = 29

![Diagram of array with indices and addresses]

• 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 `myarray[4]` in LC-3, write the code...

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

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

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

Row major form: (rows are all together)

Column major form: (columns are all together)

4x2 example
\[ A[0] = "CAT" \]
\[ A[1] = "DOG" \]

row major

CAT
AT
DOG

Column major

C
D
A
O
T
G
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:**
  
  \[
  \text{Address } (r_i, c_i) = \text{Base Address} + ((c_i \times \text{Number of Rows}) + r_i) \times \text{Element size}
  \]
Summary of 2D arrays

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

Powers of 2

How about 3-D arrays?

\[ x, y, z, t \]
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.
HeartBleed
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 ; 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
    integer ← integer div base
  end while
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)
RET

Restore1

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:

Head (R5) ← Tail (R6)

After another enqueue operation:

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:

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

Enqueue (item):

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

Dequeue (item):

```assembly
JSR SUB
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, the mod operation is applied to the current element in the circular list.
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
- BRz
- LDR
- ADD
- LEA
- JSR
- BRn
- LEA

```
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.
Midterm Overview

• No books, notes or calculators
• Show your work.
  – No credit if we don’t know how you did something
  – Lots of partial credit
• Bring your ID
CMPE12 Midterm Rules

• Put your backpacks on the sides of the classroom

• Closed books, closed notes
  – Nothing on your desk except writing equipment
    • Possible a snack and/or drink

• If you need to use the restroom during the exam please bring your exam to the front of the class along with your ID and phone.
  – We will hold your exam for you
## Transistor and Gates

### Truth Table to Gates

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logic Elements to Gates

- Draw the gate level diagram of a 2-4 decoder
ADD R0, R1, R2
LD R0, F00

F00 + PC

RO 000

REEF
# LC-3 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>DR</th>
<th>SR1</th>
<th>SR2</th>
<th>Imm5</th>
<th>BaseR</th>
<th>Offset6</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>111111</td>
</tr>
<tr>
<td>BR</td>
<td>0000</td>
<td>n</td>
<td>z</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td>PCoffset9</td>
</tr>
<tr>
<td>JMP</td>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BaseR</td>
<td></td>
<td>000000</td>
</tr>
<tr>
<td>JSR</td>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PCoffset11</td>
<td></td>
</tr>
<tr>
<td>JSRR</td>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BaseR</td>
<td></td>
<td>000000</td>
</tr>
<tr>
<td>RET</td>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td>111</td>
<td></td>
<td></td>
<td>000000</td>
</tr>
<tr>
<td>LDI</td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PCoffset9</td>
</tr>
<tr>
<td>LDR</td>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BaseR</td>
<td></td>
<td>Offset6</td>
</tr>
<tr>
<td>LEA</td>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PCoffset9</td>
</tr>
<tr>
<td>ST</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BaseR</td>
<td></td>
<td>Offset9</td>
</tr>
<tr>
<td>STI</td>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BaseR</td>
<td></td>
<td>PCoffset9</td>
</tr>
<tr>
<td>STR</td>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BaseR</td>
<td></td>
<td>Offset6</td>
</tr>
<tr>
<td>TRAP</td>
<td>1111</td>
<td></td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>trapvect8</td>
</tr>
<tr>
<td>RTI</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>000000000000</td>
</tr>
<tr>
<td>reserved</td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Base Conversion Table 8 bits

<table>
<thead>
<tr>
<th>Decimal</th>
<th>2’s Complement</th>
<th>Hexadecimal</th>
<th>Octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>-35</td>
<td>0110 0001</td>
<td>61</td>
<td>141</td>
</tr>
<tr>
<td>1001 1101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Arbitrary Base Conversion

- $1210_3$ in base 10
Binary Arithmetic

- Unsigned
- 2’s Complement
Binary Multiply (25c)

\[
\begin{array}{c}
\phantom{0}0110 \\
\times 01011 \\
\hline
0110 \\
\phantom{0}0000000 \\
\phantom{0}0110000 \\
\phantom{0}000000000 \\
\hline
000011110
\end{array}
\]
Fetch

\[ m[pc] \rightarrow IR \]

\[ pc \leftarrow pc+1 \]

HW 3 is up

solutions
NAND

0 1 1 0 1 0 1
1 0 0 1 0 1 0

AND RO, R1, R2
NOT RO, RO
; R0 output  R1, R2, input

SUB

ST R1, SUBSAVER1
ST R2, SUBSAVER2

NOT R2, R2
ADD R2, R2, #1
ADD R0, R1, R2

LD R1, 5
LD R2, 5

RET

SUBSAVER1 .fill