Arrays, Stacks and the LC-3

(Textbook chapter 10 – 10.2)
Summary

1. The array data structure
2. The stack data structure
3. \texttt{PUSH} and \texttt{POP} operations/routines
4. Using stack to store registers
5. Interrupts and the stack
Arrays

Definition: A list of values arranged sequentially in memory and grouped under a single name

Example: a list of telephone numbers

In C, the expression \( a[4] \) refers to the 5th element of the array \( a \).

Most assembly languages have only a basic concept of, or structure for, an array (.BLKW LC-3)
Properties of Arrays

Properties of arrays:
- Each element is the same size (i.e. same type)
- Elements are stored contiguously
- First element is located at the lowest memory address

To build an array in assembly language we must:
- Allocate correct amount of space for an array
- Map array addresses to memory addresses
- Ex:

  myArray  .BLKW   5
Pointer

Address of a variable in memory
Allows us to indirectly access variables
Base of an array = pointer to the first element

LEA R2, myArray
LDR R0, R2, #0
STR R0, R2, #2

myArray .BLKW 5
Memory is one-dimensional

<table>
<thead>
<tr>
<th>x3300</th>
<th>Array[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3301</td>
<td>Array[1]</td>
</tr>
<tr>
<td>x3302</td>
<td>Array[2]</td>
</tr>
<tr>
<td>x3303</td>
<td>Array[3]</td>
</tr>
<tr>
<td>x3304</td>
<td>Array[4]</td>
</tr>
<tr>
<td>x3305</td>
<td>Array[5]</td>
</tr>
<tr>
<td>x3306</td>
<td>Array[6]</td>
</tr>
<tr>
<td>x3307</td>
<td>Array[7]</td>
</tr>
<tr>
<td>x3308</td>
<td>Array[8]</td>
</tr>
<tr>
<td>x3309</td>
<td>Array[9]</td>
</tr>
</tbody>
</table>

But the concept of arrays is multi-dimensional.

So we need to map n dimensions onto a single dimension. How?
Multi-dimensional arrays

2 “major” memory mappings: column-major (where column elements are all adjacent) and row-major.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A_{00}</td>
<td>A_{01}</td>
<td>A_{02}</td>
<td>A_{03}</td>
</tr>
<tr>
<td>1</td>
<td>A_{10}</td>
<td>A_{11}</td>
<td>A_{12}</td>
<td>A_{13}</td>
</tr>
</tbody>
</table>

Column-major
Multi-dimensional arrays

2nd “major” memory mapping: row-major, where all row elements are adjacent in memory

4 columns

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A00</td>
<td>A01</td>
<td>A02</td>
<td>A03</td>
</tr>
<tr>
<td>1</td>
<td>A10</td>
<td>A11</td>
<td>A12</td>
<td>A13</td>
</tr>
</tbody>
</table>

Row-major
Multi-dimensional arrays

2 rows

<table>
<thead>
<tr>
<th></th>
<th>A00</th>
<th>A01</th>
<th>A02</th>
<th>A03</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A00</td>
<td>A01</td>
<td>A02</td>
<td>A03</td>
</tr>
<tr>
<td>1</td>
<td>A10</td>
<td>A11</td>
<td>A12</td>
<td>A13</td>
</tr>
</tbody>
</table>

4 columns

x3100

<table>
<thead>
<tr>
<th>A00</th>
<th>A00</th>
</tr>
</thead>
</table>

x3101

<table>
<thead>
<tr>
<th>A01</th>
<th>A10</th>
</tr>
</thead>
</table>

x3102

<table>
<thead>
<tr>
<th>A02</th>
<th>A01</th>
</tr>
</thead>
</table>

x3103

<table>
<thead>
<tr>
<th>A03</th>
<th>A11</th>
</tr>
</thead>
</table>

x3104

<table>
<thead>
<tr>
<th>A10</th>
<th>A02</th>
</tr>
</thead>
</table>

x3105

<table>
<thead>
<tr>
<th>A11</th>
<th>A12</th>
</tr>
</thead>
</table>

x3106

<table>
<thead>
<tr>
<th>A12</th>
<th>A03</th>
</tr>
</thead>
</table>

x3107

| A13 | A13 |

column-major: (fortran, matlab)

row-major: (almost everyone else)
Stack and Queue Data Structures

• Abstract Data Structures (more abstract than arrays)
  – defined by the rules for inserting and extracting data
• **LIFO** (Last In-First Out)    **FIFO** (First In-First Out)

```
A, B, C
C, B, A
```

```
A, B, C
```

```
A, B, C
```

“stack”

“queue”
Stack

- More fundamental to how programs work than queue
- Physical Analog: A coin holder as a stack
Stack Data Structure

• Like Physical Stack, Last In First Out
• Unlike physical stack
  – Elements do not move when you push or pop data
  – Top of Stack (TOS) moves instead
Stack Data Structure

• Data structures:
  – Top Of Stack (TOS): first location on stack
  – Stack Pointer: Register that holds the address of the TOS
  – Stack body: array

• Operations (LIFO):
  – **PUSH** (enter item at top of stack)
  – **POP** (“remove” item from top of stack)
  – Data in stack does not move; TOS moves during PUSH and POP
LC 3 stack

• LC 3 conventions
  – Stack “grows” towards address 0000
  – TOS holds the next value to be “POPed”

• Stack Pointer
  – R6 is the stack register, holds address of Top Of Stack
  – Don’t use R6 in your user program
Stack overflow and underflow

Error conditions:

- Overflow (trying to push onto full stack)
- Underflow (trying to pop from empty stack)

Programmer is responsible for preventing these errors
Implementing a Stack in LC-3

1. Declare an array to be your stack
2. Initialize R6 to point to the TOS
3. Implement **PUSH** and **POP**
Initializing Stack

.ORIG x3000
  LD     R6, Base_of_Satck

(more code)

Base_of_Stack  .FILL x3FFF

(more variables)

.END
Push and Pop in LC-3

• Assume data to push is in a specific register (R0)
• Decrement R6 (our stack grows downward)
• Then write data in R0 to mem[R6]

```
PUSH     ADD     R6, R6, #-1
        STR     R0, R6, #0
```

• Assume data to pop will go to a specific register (R0)
• Read data at mem[SP] into R0
• Then increment SP

```
POP      LDR     R0, R6, #0
        ADD     R6, R6, #1
```
When is the stack empty or full?

Remember: TOS always points to top valid element

Stack Full:
TOS = x3FFB
and PUSH

Stack Empty:
TOS = x4000
and POP
Checking for overflow

• Before pushing, we have to test to see if stack is full (overflow)

```
PUSH     ST R1  PUSH_SAVE_R1
         ST R2  PUSH_SAVE_R2
         LD    R1, NEG_MAX
         ADD   R2, R6, R1
         BRz   STACK_FAILURE
         ADD   R6, R6, #-1
         STR   R0, R6, #0
         ST R1  PUSH_SAVE_R1
         ST R2  PUSH_SAVE_R2
         RET

.FILL NEG_MAX xC005  ;NEG_MAX = -3FFB
.FILL PUSH_SAVE_R1
.FILL PUSH_SAVE_R2
```

Black instructions for original PUSH
Checking for underflow

• Before pushing, we have to test to see if stack if full (overflow)

```
POP      ST  R1  POP_SAVE_R1
         ST  R2  POP_SAVE_R2
LD       R1,  NEG_EMPTY
ADD      R2,  R6,  R1
BRz      STACK_FAILURE
LDR      R0,  R6,  #0
ADD      R6,  R6,  #1
ST       R1  POP_SAVE_R1
         ST  R2  POP_SAVE_R2
RET
.FILL    NEG_MAX  xC005  ;NEG_MAX = -3FFB
         FILL  POP_SAVE_R1
         FILL  POP_SAVE_R2
```

Black instructions for original POP
When is the stack empty or full?

Different convention:
TOS always points to next available location to **PUSH**

Stack Overflow:
TOS = 3FFA

Stack Underflow:
TOS = 3FFE
Another Stack protocol on LC-3

- **PUSH** pushes R0, returns success in R5
- **POP** pops into R0, returns success in R5
- Stack pointer is R6
- All other used registers need to be callee-saved
- The TOS points to the top valid element
- The stack goes from **x3FFF** to **x3FFB**
Stack protocol in LC-3: **POP**  (1/3)

;R5 holds success or failure and thus the value of R5 is changed in this subroutine.

POP  ST    R2, Save2 ; save, needed by POP
     ST    R1, Save1 ; save, needed by POP
     LD    R1, nBASE ; nBASE contains -x3FFF
     ADD   R1, R1, #~1 ; R1 now has -x4000
     ADD   R2, R6, R1 ; compare SP to BASE
     BRz   fail_exit ; branch if stack is empty
     LDR   R0, R6, #0 ; the actual ‘pop’
     ADD   R6, R6, #1 ; adjust stack pointer
     BRnzp success_exit
Stack protocol in LC-3: **PUSH**  

; R5 holds success or failure and thus the value of R5 ; is changed in this subroutine.

PUSH  
ST  R2, Save2 ; needed by PUSH
ST  R1, Save1 ; needed by PUSH
LD  R1, nMAX ; nMAX has -x3FFB
ADD R2, R6, R1 ; compare SP to x3FFB
BRz fail_exit ; branch is stack is full
ADD R6, R6, #-1 ; adjust Stack Pointer
STR R0, R6, #0 ; the actual ‘push’
Stack prot. in LC-3: Return values
(3/3)

<table>
<thead>
<tr>
<th></th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>success_exit</td>
<td>LD  R1, Save1 ; restore registers</td>
</tr>
<tr>
<td></td>
<td>LD  R2, Save2</td>
</tr>
<tr>
<td></td>
<td>AND R5, R5, #0 ; R5 &lt;--- success</td>
</tr>
<tr>
<td></td>
<td>RET</td>
</tr>
<tr>
<td>fail_exit</td>
<td>LD  R1, Save1 ; restore registers</td>
</tr>
<tr>
<td></td>
<td>LD  R2, Save2</td>
</tr>
<tr>
<td></td>
<td>AND R5, R5, #0</td>
</tr>
<tr>
<td></td>
<td>ADD R5, R5, #1 ; R5 &lt;--- fail</td>
</tr>
<tr>
<td></td>
<td>RET</td>
</tr>
<tr>
<td>nBASE</td>
<td>.FILL  xC001 ; nBASE has -x3FFF</td>
</tr>
<tr>
<td>nMAX</td>
<td>.FILL  xC005 ; nMAX has -x3FFB</td>
</tr>
<tr>
<td>Save1</td>
<td>.FILL  x0000</td>
</tr>
<tr>
<td>Save2</td>
<td>.FILL  x0000</td>
</tr>
</tbody>
</table>
Stack as an alternative to Registers

• Three-address vs zero-address
  – The LC-3 explicitly specifies the location of each operand: it is a three-address machine
    • e.g. \texttt{ADD R0, R1, R2}
  – Some machines use a stack data structure for all temporary data storage: these are zero-address machines
    • the instruction \texttt{ADD} would simply pop the top two values from the stack, add them, and push the result back on the stack
    • some calculators use a stack to do arithmetic, most general purpose microprocessors use a register bank
Interesting Routines in Book

• 10.4.2 shows how to convert ASCII to Binary with a stack.
• 10.4.3 shows how to convert from Binary to ASCII with a stack.
Temporary storage in a subroutine

SubroutineX

ST R0, Temp1
ST R1, Temp2
ST R7, Temp3
...
JSR SubroutineY
...
LD R0, Temp1
LD R1, Temp2
LD R7, Temp3
RET

Works, but what if SubroutineY is SubroutineX?

Temp1 .FILL
Temp2 .FILL
Temp3 .FILL

Where will RET return to?
Stack as temporary storage

Sub_X

ADD R6, R6, #-3
STR R0, R6, #2
STR R1, R6, #1
STR R7, R6, #0
...
JSR Sub_X
...
LDR R0, R6, #2
LDR R1, R6, #1
LDR R7, R6, #0
ADD R6, R6, #3
RET

Stack at beginning of Sub_X

x78F0 -------
x78F1 -------
x78F2 -------
x78F3 -------
x78F4 -------
x78F5 -------
x78F6 -------
x78F7 -------
x78F8 data

R6
Stack to create space for local variables

Sub_X ADD R6, R6, #-10
    ...
    JSR Sub_X
    ...
ADD R6, R6, #10
RET

Creates 10 local variables that can be accessed as mem[R6] to mem[R6]+9
Interrupts (part 2)

• Initiate the interrupt
  – Signal computer to “interrupt” current program
  – Save the state of the original program
  – Take control of the CPU

• Service the interrupt
  – Execute interrupt routine

• Return from the interrupt
  – Return the CPU to it’s “pre-interrupt” state
  – Give the original program control of the CPU
Saving the original program’s state

- There is a second stack: supervisor stack (R6)
- Interrupt must save current state on S-stack
  - Program Status Register (PSR) = Privilege, Priority, CCR
  

\[
\begin{array}{cccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\end{array}
\]

- Program Counter

- The remaining registers are pushed onto the S-stack by the interrupt routine
Interrupt Service Routine

- Runs in Privileged mode and uses the Supervisor Stack through the Supervisor R6.
- End of interrupt routine restores R0-R7 (except R6) then executes RTI
  - RTI restores original PSR and PC values to the CPU by popping them from the S-stack
Interrupt example

Program A

x3000
x3020
x30F0

Interrupt routine B

x6010
x6223
x6555

RTI

Interrupt routine B

x7000
x70F0

RTI

x6224
PSR(B)
x3021
PSR(A)

-----

-----
saved.R6
Exercises to turn in Friday November 16

• Ex 10.3, 10.4, 10.6, 10.10
  also
• Ex 9.10 (by semantics the question means how would the trap actually work and still have space for 16 instructions per trap.)