10.1
The defining characteristic of a stack is the unique specification of how it is to be accessed. Stack is a LIFO (Last in First Out) structure. This means that the last thing that is put in the stack will be the first one to get out from the stack.

10.2
The entries in the model in Figure 10.2 actually move when other entries are pushed and popped, while they do not in the model of Figure 10.3. If the stack is implemented in memory, it makes more sense to access the one entry alone, plus the stack pointer, rather than access all entries on the stack. If the stack is implemented as a piece of tailored logic, it is faster to physically move the actual entries. provided that the power required to do so can handle it. But that is a subject for a later course.

10.4
This routine copies the value of the first element on stack into R0. If underflow occurs, R5 is set to 1 (failure) else R5 remains 0 (success). Overflow error checking is not necessary because we are not adding anything to the stack.

```
PEEK
AND R5, R5, #0 ; initialize R5
LEA R0, StackBase
NOT R0, R0
ADD R0, R0, #-1 ;R0 = -(addr of stackbase + 1)
ADD R0, R0, R6 ;R6 - stack pointer
BRZ Underflow
LDR R0, R6, #0 ;put the first ;element in R0
RET
Underflow
ADD R5, R5, #1 ;failure
RET
```

```
StackMax .BLKW 10, x0000
StackBase .FILL x0000
```

10.6
This routine pushes the values in R0 and R1 onto the stack. R5 is set to 0 if operation is successful. If overflow occurs R5 is set to 1. This routine works with a stack consisting of memory locations x3FFF(BASE) to x3FFA(MAX).

```
PUSHST
R2, SaveR2
LD R2, MAX
NOT R2, R2
ADD R2, R2, #1 ; R2 = -addr of stackmax
ADD R2, R6, R2
BRz Failure
STR R0, R6, #-1
STR R1, R6, #-2
ADD R6, R6, #-2
AND R5, R5, #0
```
LD R2, SaveR2
RET

Failure  AND R5, R5, #0
         ADD R5, R5, #1
         LD R2, SaveR2
         RET

MAX       .FILL x3FFA
SaveR2    .FILL x0000

This routine pops the top two elements of the stack into R0 and R1. R5 is set to 0 if the operation is successful. If under_ow occurs R5 is set to 1. This routine works with a stack consisting of memory locations x3FF(BASE) to x3FFA(MAX).

POP       ST R2, SaveR2
         LD R2, EMPTY ; EMPTY <-- -x4000
         ADD R2, R6, R2
         BRz Failure ; Underflow
         LDR R1, R6, #0 ; Pop the first value
         LDR R0, R6, #1 ; Pop the second value
         ADD R6, R6, #2
         AND R5, R5, #0
         RET

Failure  AND R5, R5, #0
         ADD R5, R5, #1
         RET
         EMPTY .FILL xC000
SaveR2   .FILL x0000

10.8
(a) The stack looks like the following after each operation. Top of the stack is the rightmost element.
PUSH A : A
PUSH B : A B
POP : A
PUSH C : A C
PUSH D : A C D
POP : A C
PUSH E : A C E
POP : A C
POP : A
PUSH F : A F
So the stack contains A F after the PUSH F operation.
(b) The stack looks like the following after each operation. Top of the stack is the rightmost element.
PUSH G : A F G
PUSH H : A F G H
PUSH I : A F G H I
PUSH J : A F G H I J
POP : A F G H I
PUSH K : A F G H I K
The stack contains the most elements after PUSH J and PUSH K operations.

(c) Now the stack contains A F M (M is at the top of stack).

10.11
Correction, The question should have read:
In the example of Section 10.2.3, what are the contents of locations 0x01F1 and 0x01F2?
They are part of a larger structure. Provide a name for that structure.
x01F1 - 0x6200
x01F2 - 0x6300
They are part of the Interrupt Vector Table.

10.17
The Multiply step works by adding the multiplicand a number of times to an accumulator. The
number of times to add is determined by the multiplier. The number of instructions executed
to perform the Multiply step = 3 + 3*n, where n is the value of the multiplier. We will in
general do better if we replace the core of the Multiply routine (lines 17 through 19 of Figure
10.14) with the following, doing the Multiply as a series of shifts and adds:

```
AND R0, R0, #0
ADD R4, R0, #1 ;R4 contains the bit mask (x0001)
```

Again
```
AND R5, R2, R4 ;Is corresponding
BRz BitZero ;bit of multiplier=1
ADD R0, R0, R1 ;Multiplier bit=1
 ;---> add
 ;shifted multiplicand
BRn Restore2 ;Product has already
 ;exceeded range
```

BitZero
```
ADD R1, R1, R1 ;Shift the
 ;multiplicand bits
BRn Check ;Mcan too big
 ;---> check if any
 ;higher mpy bits = 1
ADD R4, R4, R4 ;Set multiplier bit to
 ;next bit position
BRn DoRangeCheck ;We have shifted mpy
BRnzp Again ;bit into bit 15
 ;--->done.
```

Check
```
AND R5, R2, R4
BRp Restore2
ADD R4, R4, R4
BRp Check
```

DoRangeCheck
10.20
ASCIItoBinary   AND R0, R0, #0 ; R0 will be used for our result
                 LEA R5, ASCIIBUFF
                 LD R4, NegASCIIOffset ; R4 gets xFFD0, i.e., -x0030
                 ADD R1, R1, #0 ; Test number of digits.
LOOP
                 BRz DoneAtoB ;
                 LD R2, CNT
                 ADD R3, R0, #0
                 MUL10 ADD R0, R0, R3
                 ADD R2, R2, #-1
                 BRp MUL10
                 LDR R3, R5, #0 ; Strip off the ASCII template
                 ADD R3, R4, R3 ; Add digits contribution
                 ADD R5, R5, #1
                 ADD R1, R1, #-1
                 BRnzp LOOP
DoneAtoB
                 RET
NegASCIIOffset .FILL xFFD0
CNT .FILL #9
ASCIIBUFF .BLKW #10

10.23
This program reverses the input string. For example, given an input of .Howdy., the output
is .ydwoH..  

Written Questions

2)  
Base Address  = 0x40FF, Width = 30, Height = 24
Row Major     = 0x40FF + ( y * Width + x )
Column Major  = 0x40FF + ( x * Height + y )

<table>
<thead>
<tr>
<th>Index (y, x)</th>
<th>Row Major</th>
<th>Column Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0x40FF</td>
<td>0x40FF</td>
</tr>
<tr>
<td>5,5</td>
<td>0x419A</td>
<td>0x417C</td>
</tr>
<tr>
<td>4,6</td>
<td>0x417D</td>
<td>0x4193</td>
</tr>
<tr>
<td>6,4</td>
<td>0x41B7</td>
<td>0x4165</td>
</tr>
<tr>
<td>23,29</td>
<td>0x43CE</td>
<td>0x43CE</td>
</tr>
</tbody>
</table>

3)
A stack is a LIFO structure, while a queue is a FIFO structure which would not serve the program flow of the LC-3 machine.

4)
Caller saves are prior to execution of the subroutine, while callee saves are done by (inside) the subroutine. Caller savings are a waste if the called routine doesn’t use the registers, and callee savings are a waste if the calling program wasn’t using the registers.