F.10 Chapter 10 Solutions

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.

10.3 (a) PUSH R1

(b) POP R0

(c) PUSH R3

(d) POP R7

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
```

10.5 One way to check for overflow and underflow conditions is to keep track of a pointer that tracks the bottom of the stack. This pointer can be compared with the address of the first and last addresses of the space allocated for the stack.

```
;
; Subroutines for carrying out the PUSH and POP functions. This
; program works with a stack consisting of memory locations x3FFF
; (BASE) through x3FFB (MAX). R6 is the bottom of the stack.
```

```
;

POP             ST  R1, Save1 ; are needed by POP.
                ST  R2, Save2
                ST  R3, Save3
                LD  R1, NBASE   ; BASE contains -x3FFF.
                ADD R1, R1, #-1 ; R1 contains -x4000.
                ADD R2, R6, R1  ; Compare bottom of stack to x4000

                BRz fail_exit   ; Branch if stack is empty.

                LD  R1, BASE    ;Iterate from the top of
                                ;the stack
                LDI R0, BASE    ;Load the value from the
                NOT R3, R6      ;top of stack
                ADD R3, R3, #1  ;Generate the
                                ;negative of the
                                ;bottom-of-stack pointer
                ADD R6, R6, #1  ;Increment the
                                ;bottom-of-stack
                                ;pointer

pop_loop        ADD   R2, R1, R3  ;Compare iterating
                                  ;pointer to
                                  ;bottom-of-stack pointer
                BRz   success_exit;Branch if no more
                                  ;entries to shift
                LDR   R2, R1, #-1  ;Load the entry to shift
                STR   R2, R1, #0  ;Shift the entry
                ADD   R1, R1, #-1  ;Increment the
                                  ;iterating pointer
                BRnzp pop_loop

PUSH            ST  R1, Save1 ; Save registers that
                ST  R2, Save2 ; are needed by PUSH.
                ST  R3, Save3
                LD  R1, MAX   ; MAX contains -x3FFB
                ADD R2, R6, R1 ; Compare stack pointer to -x3FFB
                BRz fail_exit ; Branch if stack is full.

                ADD R1, R6, #0 ;Iterate from the bottom
                                ;of stack
                LD  R3, NBASE    ;NBASE contains
                                 ;-x3FFF
                ADD R3, R3, #-1   ; R3 = -x4000
```

```
push_loop         ADD   R2, R1, R3  ;Compare iterating
                                    ;pointer to
                                    ;bottom-of-stack pointer
                  BRz   push_entry  ;Branch if no more
                                    ;entries to shift
                  LDR   R2, R1, #0  ;Load the entry to shift
                  STR   R2, R1, #-1  ;Shift the entry
                  ADD   R1, R1, #1 ;Decrement the
                                    ;iterating pointer
                  BRnzp push_loop

push_entry        ADD   R6, R6, #-1  ;Increment the
                                    ;bottom-of-stack pointer
                  STI   R0, BASE    ;Push a value onto stack
                  BRnzp success_exit

success_exit      LD    R1, Save1   ;Restore original
                  LD    R2, Save2   ;register values
                  LD    R3, Save3
                  AND   R5, R5, #0  ;R5 <--- success
                  RET

fail_exit         LD    R1, Save1   ;Restore original
                  LD    R2, Save2   ;register values
                  LD    R3, Save3
                  AND   R5, R5, #0
                  ADD   R5, R5, #1  ;R5 <--- failure
                  RET

BASE              .FILL x3FFF
NBASE             .FILL xC001 ; NBASE contains -x3FFF.
MAX               .FILL xC005

Save1             .FILL x0000
Save2             .FILL x0000
Save3             .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).

```
PUSH              ST      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 underflow occurs R5 is set to 1. This routine works with a stack consisting of memory locations x3FFF(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
```

10.7
```
; Subroutines for carrying out the PUSH and POP functions. This
; program works with a stack consisting of memory locations x3FFF
; (BASE) through x3FFB (MAX). R6 is the stack pointer. R3 contains
; the size of the stack element. R4 is a pointer specifying the
; location of the element to PUSH from or the space to POP to
;
```

```
POP           ST     R2, Save2 ; are needed by POP.
              ST     R1, Save1
              ST     R0, Save0
              LD     R1, BASE ; BASE contains -x3FFF.
              ADD    R1, R1, #-1 ; R1 contains -x4000.
              ADD    R2, R6, R1 ; Compare stack pointer to x4000
              BRz    fail_exit ; Branch if stack is empty.
              ADD    R0, R4, #0
              ADD    R1, R3, #0
              ADD    R5, R6, R3
              ADD    R5, R5, #-1
              ADD    R6, R6, R3

pop_loop      LDR    R2, R5, #0
              STR    R2, R0, #0
              ADD    R0, R0, #1
              ADD    R5, R5, #-1
              ADD    R1, R1, #-1
              BRp    pop_loop
              BRnzp  success_exit

PUSH          ST     R2, Save2 ; Save registers that
              ST     R1, Save1 ; are needed by PUSH.
              ST     R0, Save0
              LD     R1,MAX ; MAX contains -x3FFB
              ADD    R2,R6,R1 ; Compare stack pointer to -x3FFB
              BRz    fail_exit ; Branch if stack is full.
              ADD    R0, R4, #0
              ADD    R1, R3, #0
              ADD    R5, R6, #-1

              NOT    R2, R3
              ADD    R2, R2, #1
              ADD    R6, R6, R2

push_loop     LDR    R2, R0, #0
              STR    R2, R5, #0
              ADD    R0, R0, #1
              ADD    R5, R5, #-1
              ADD    R1, R1, #-1
              BRp    push_loop

success_exit  LD     R0, Save0
              LD     R1, Save1 ; Restore original
              LD     R2, Save2 ; register values.
```

```
                    AND     R5, R5, #0 ; R5 <-- success.
                    RET
fail_exit           LD      R0, Save0
                    LD      R1, Save1 ; Restore original
                    LD      R2, Save2 ; register values.

                    AND     R5, R5, #0
                    ADD     R5, R5, #1 ; R5 <-- failure.
                    RET
BASE                .FILL   xC001 ; BASE contains -x3FFF.
MAX                 .FILL   xC005
Save0               .FILL   x0000
Save1               .FILL   x0000
Save2               .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
POP    : A F G H I
POP    : A F G H
POP    : A F G
PUSH L : A F G L
POP    : A F G
POP    : A F
PUSH M : A F M
```

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).

10.9 (a) BDECJKIHLG

(b)
```
Push Z
Push Y
Pop Y
Push X
Pop X
Push W
Push V
Pop V
Push U
Pop U
Pop W
Pop Z
Push T
Push S
Pop S
Push R
Pop R
Pop T
```

(c) 14 different output streams.

10.10 For example, lets take the following program which adds 10 numbers starting at memory location x4000 and stores the result at x5000

```
.ORIG  x3000
LD    R1, PTR
AND   R0, R0, #0
LD    R2, COUNT
LOOP LDR   R3, R1, #0
ADD   R0, R0, R3
ADD   R2, R2, #-1
BRp   LOOP
STI   R0, PRES
HALT
PTR .FILL  x4000
PRES .FILL x5000
COUNT .FILL  #10
```

If the condition codes were not saved as part of initiation of the interrupt service routine, we could end up with incorrect results. In this program, take the case when an interrupt occurred during the processing of the intruction at location x3005 and condition codes were not saved. Let R2 = 5 and hence the condition codes be P=1, N=0, Z=0 before servicing the interrupt. When control is returned to the instruction at location x3006, the BR instruction, the condition codes depend on the processing within the interrupt service routing. If they are P=0, N=0, Z=1, then the Br is not taken. This means that result stored is just the sum of the first five values and not all ten.

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.

10.12 (a) PC = x3006

Stack:
___
___

xxxxx - Saved SSP

(b) PC = x6200

Stack:
___
___

PSR of Program A - R6
x3007
xxxxx

(c) PC = x6300

Stack:
___
___

PSR for device B - R6
x6203
PSR of Program A
x3007
xxxxx

(d) PC = x6400

Stack:
___
___

PSR for device C - R6
x6311
PSR for device B
x6203
PSR of Program A
x3007
xxxxx

(e) PC = x6311

Stack:
——
——

PSR for device C
x6311
PSR for device B - R6
x6203
PSR of Program A
x3007
xxxxx

(f) PC = x6203

Stack:
——
——

PSR for device C
x6311
PSR for device B
x6203
PSR of Program A - R6
x3007
xxxxx

(g) PC = x3007

Stack:
——
——

PSR for device C
x6311
PSR for device B
x6203

PSR of Program A
x3007
xxxxx - Saved.SSP

10.13 (a) PC = x3006

Stack:

____

____

xxxxx - Saved SSP

(b) PC = x6200

Stack:

____

____

PSR of Program A - R6
x3007
xxxxx

(c) PC = x6300

Stack:

____

____

PSR for device B - R6
x6203
PSR of Program A
x3007
xxxxx

(d) PC = x6203

Stack:

____

____

PSR for device B
x6203
PSR of Program A - R6
x3007
xxxxx

(e) PC = x6400

Stack:
___
___

PSR for device B - R6
x6204
PSR of Program A
x3007
xxxxx

(f) PC = x6204

Stack:
___
___

PSR for device B
x6204
PSR of Program A - R6
x3007
xxxxx

(g) PC = x3007

Stack:
___
___

PSR for device B
x6204
PSR of Program A
x3007
xxxxx - Saved.SSP

10.14 Correction - If the buffer is full, a character has been stored in 0x40FE.

```
        LDI     R0, KBDR
        LDI     R1, PENDBF
        LD      R2, NEGEND
        ADD     R2, R1, R2
        BRz     ERR         ; Buffer is full
        STR     R0, R1, #0  ; Store the character
        ADD     R1, R1, #1
        STI     R1, PENDBF  ; Update next available empty
```

```
                                  ; buffer location pointer
         BRnzp   DONE
ERR      LEA     R0, MSG
PUTS
DONE     RTI
KBDR     .FILL   xFE02
PBUF     .FILL   x4000
PENDBF   .FILL   x40FF
NEGEND   .FILL   xBF01 ; xBF01 = -(x40FF)
MSG      .STRINGZ  "Character cannot be accepted; input buffer full."
```

10.15 Note: This problem introduces the concept of a data structure called a queue. A queue has a First-In-First-Out(FIFO) property - Data is removed in the order as it is inserted. By having the pointer to the next available empty location wrap around to the beginning of the buffer in this problem, the queue becomes a circular queue. A circular queue is space efficient as it makes use of entries which have been removed by the consuming program. These concepts will be covered in detail in a data structure or algorithms course.

```
         LDI     R0, KBDR
         LDI     R1, PNUMCH
         LD      R2, NMAXCH
         ADD     R2, R1, R2
         BRz     ERR         ; Buffer is full

         ADD     R1, R1, #1
         STI     R1, PNUMCH ; update char count

         LDI     R1, PENDBF
         STR     R0, R1, #0 ; Store the character

         LD      R2, NEGEND
         ADD     R2, R1, R2 ; Compare the next available empty location
                            ; pointer with x40FC.
         BRz     RESETPTR   ; If same, wrap around so that the next available
                            ; empty location is x4000
         ADD     R1, R1, #1
         BRnzp   STPTR
RESETPTR LD      R1, PBUF

STPTR    STI     R1, PENDBF ; Update next available empty buffer
                            ; location pointer
         BRnzp   DONE
ERR      LEA     R0, MSG
         PUTS
DONE     RTI
```

```
KBDR    .FILL   xFE02
PBUF    .FILL   x4000
PNUMCH  .FILL   x40FD
PENDBF  .FILL   x40FF
NEGEND  .FILL   xBF04 ; xBF04 = -(x40FC)
NMAXCH  .FILL   xFF03 ; xFF03 = -(xFD)
MSG     .STRINGZ  "Character cannot be accepted; input buffer full."
```

10.16 Correction - Consider the modified interrupt handler of Exercise 10.15.

The variable "number of characters in the buffer" is shared between both the interrupt handler which is adding numbers to the buffer and the program that is removing characters. So now if the program has just loaded the number of characters in the buffers value into a register when an interrupt occurs, the value in the register is going to be stale after the interrupt is serviced. Hence when the program writes this value back to x40FD, it is writing a wrong value.

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         ;Mcand 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
```

10.18 To add the two numbers, convert the two characters from their ASCII representations to 2's complement integers. After performing the addition, convert the result back to ASCII representation.

```
                  LD        R2, NEGASCII
                  TRAP      x23
                  ADD       R1, R0, R2  ;Remove ASCII template
                  TRAP      x23
                  ADD       R0, R0, R2  ;Remove ASCII template
                  ADD       R0, R1, R0
                  LD        R2, ASCII
                  ADD       R0, R0, R2  ;Add the ASCII template
                  TRAP      x21
                  TRAP      x25
NEGASCII          .FILL     x-0030
ASCII             .FILL     x0030
```

10.19 This program assumes that hex digits are all capitalized.

```
                  LD        R3, NEGASCII
                  LD        R5, NEGHEX
                  TRAP      x23
                  ADD       R1, R0, R3   ;Remove ASCII template
                  LD        R4, HEXTEST  ;Check if digit is hex
                  ADD       R0, R1, R4
                  BRnz      NEXT1
                  ADD       R1, R1, R5   ;Remove extra
                                         ;offset for hex
NEXT1             TRAP      x23
                  ADD       R0, R0, R3   ;Remove ASCII template
                  ADD       R2, R0, R4   ;Check if digit is hex
                  BRnz      NEXT2
                  ADD       R0, R0, R5   ;Remove extra
                                         ;offset for hex

NEXT2             ADD       R0, R1, R0   ;Add the numbers
                  ADD       R1, R0, R4   ;Check if digit > 9
                  BRnz      NEXT3
                  LD        R2, HEX
                  ADD       R0, R0, R2   ;Add offset for hex digits
```

```
NEXT3            LD       R2, ASCII
                 ADD      R0, R0, R2   ;Add the ASCII template

DONE             TRAP     x21
                 TRAP     x25

ASCII            .FILL    x0030
NEGASCII         .FILL    x-0030
HEXTEST          .FILL    #-9
HEX              .FILL    x0007
NEGHEX           .FILL    x-7
```

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 ;
                 ADD     R3, R4, R3 ; Strip off the ASCII template
                 ADD     R0, R0, 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
```

10.21

```
;
; R1 contains the number of digits including 'x'. Hex
; digits must be in CAPS.

ASCIItoBinary    AND R0, R0, #0 ; R0 will be used for our result
                 ADD  R1, R1, #0 ; Test number of digits.
                 BRz  DoneAtoB   ; There are no digits
;
```

```
                LD  R3, NegASCIIOffset ; R3 gets xFFD0, i.e., -x0030
                LEA  R2, ASCIIBUFF
                LD R6, NegXCheck
                LDR R4, R2, #0
                ADD R6, R4, R6
                BRz DoHexToBin

                ADD  R2, R2,R1
                ADD   R2, R2, #-1 ; R2 now points to "ones" digit
;
                LDR  R4, R2, #0 ; R4 <-- "ones" digit
                ADD  R4, R4, R3 ; Strip off the ASCII template
                ADD  R0, R0, R4 ; Add ones contribution
;
                ADD  R1, R1, #-1
                BRz  DoneAtoB ; The original number had one digit
                ADD  R2, R2, #-1 ; R2 now points to "tens" digit
;
                LDR  R4, R2, #0 ; R4 <-- "tens" digit
                ADD  R4, R4, R3 ; Strip off ASCII template
                LEA  R5, LookUp10 ; LookUp10 is BASE of tens values
                ADD  R5, R5, R4 ; R5 points to the right tens value
                LDR  R4, R5, #0
                ADD  R0, R0, R4 ; Add tens contribution to total
;
                ADD  R1, R1, #-1
                BRz  DoneAtoB ; The original number had two digits
                ADD  R2, R2, #-1 ; R2 now points to "hundreds" digit
;
                LDR  R4, R2, #0 ; R4 <-- "hundreds" digit
                ADD  R4, R4, R3 ; Strip off ASCII template
                LEA R5, LookUp100 ; LookUp100 is hundreds BASE
                ADD  R5, R5, R4 ; R5 points to hundreds value
                LDR  R4, R5, #0
                ADD  R0, R0, R4 ; Add hundreds contribution to total
                RET

DoHexToBin      ; R3 = NegASCIIOffset
                ; R2 = Buffer Pointer
                ; R1 = Num of digits + x
                ;
                ST R7, SaveR7
                LD R6, NumCheck
                ADD R1, R1, #-1
```

```
                ADD  R2, R2,R1
;
                LDR  R4, R2, #0 ; R4 <-- "ones" digit
                ADD  R4, R4, R3 ; Strip off the ASCII template
                ADD R7, R4, R6
                BRnz Cont1
                LD R7, NHexDiff
                ADD R4, R4, R7
Cont1           ADD  R0, R0, R4 ; Add ones contribution

;
                ADD  R1, R1, #-1
                BRz  DoneAtoB ; The original number had one digit
                ADD  R2, R2, #-1 ; R2 now points to "tens" digit
;
                LDR  R4, R2, #0 ; R4 <-- "tens" digit
                ADD  R4, R4, R3 ; Strip off ASCII template
                ADD R7, R4, R6
                BRnz Cont2
                LD R7, NHexDiff
                ADD R4, R4, R7

Cont2           LEA  R5, LookUp16
                ADD  R5, R5, R4
                LDR  R4, R5, #0
                ADD  R0, R0, R4
;
                ADD  R1, R1, #-1
                BRz  DoneAtoB ; The original number had two digits
                ADD  R2, R2, #-1 ; R2 now points to "hundreds" digit
;
                LDR  R4, R2, #0
                ADD  R4, R4, R3 ; Strip off ASCII template
                ADD R7, R4, R6
                BRnz Cont3
                LD R7, NHexDiff
                ADD R4, R4, R7

Cont3           LEA R5, LookUp256
                ADD  R5, R5, R4
                LDR  R4, R5, #0
                ADD  R0, R0, R4

;
DoneAtoB        LD R7, SaveR7
```

```
                RET

NegASCIIOffset  .FILL  xFFD0
NumCheck        .FILL  #-9
NHexDiff        .FILL #-7
NegXCheck       .FILL  xFF88
SaveR7          .FILL  x0000

ASCIIBUFF       .BLKW 4
LookUp10        .FILL #0
                .FILL #10
                .FILL #20
                .FILL #30
                .FILL #40
                .FILL #50
                .FILL #60
                .FILL #70
                .FILL #80
                .FILL #90
;
LookUp100       .FILL #0
                .FILL #100
                .FILL #200
                .FILL #300
                .FILL #400
                .FILL #500
                .FILL #600
                .FILL #700
                .FILL #800
                .FILL #900
LookUp16        .FILL     #0
                .FILL     #16
                .FILL     #32
                .FILL     #48
                .FILL     #64
                .FILL     #80
                .FILL     #96
                .FILL     #112
                .FILL     #128
                .FILL     #144
                .FILL     #160
                .FILL     #176
                .FILL     #192
                .FILL     #208
                .FILL     #224
```

```
                .FILL    #240

LookUp256       .FILL    #0
                .FILL    #256
                .FILL    #512
                .FILL    #768
                .FILL    #1024
                .FILL    #1280
                .FILL    #1536
                .FILL    #1792
                .FILL    #2048
                .FILL    #2304
                .FILL    #2560
                .FILL    #2816
                .FILL    #3072
                .FILL    #3328
                .FILL    #3584
                .FILL    #3840
```

10.22

```
;
BinarytoASCII   AND     R4, R4, #0
                LD      R5, ASCIIoffset
                LEA     R1, ASCIIBUFF ; R1 points to string being generated
                ADD     R0, R0, #0 ; R0 contains the binary value
                BRn     NegSign ;
                BRnzp   Begin100
NegSign         LD      R2, ASCIIminus ; First store ASCII minus sign
                STR     R2, R1, #0
                NOT     R0, R0 ; Convert the number to absolute
                ADD     R0, R0, #1 ; value; it is easier to work with.
                ADD     R1, R1, #1
;
Begin100        AND     R2, R2, #0 ; Prepare for "hundreds" digit
;
                LD      R3, Neg100 ; Determine the hundreds digit
Loop100         ADD     R0, R0, R3
                BRn     End100
                ADD     R2, R2, #1
                BRnzp   Loop100
;
End100          ADD     R2, R2, #0
                BRz     Post100
                ADD     R2, R2, R5
                STR     R2, R1, #0 ; Store ASCII code for hundreds digit
```

```
                ADD     R1, R1, #1
                ADD     R4, R4, #1 ; Stored a digit
Post100         LD      R3, Pos100
                ADD     R0, R0, R3 ; Correct R0 for one-too-many subtracts
;
                AND     R2, R2, #0 ; Prepare for "tens" digit
;
Begin10         LD      R3, Neg10 ; Determine the tens digit
Loop10          ADD     R0, R0, R3
                BRn     End10
                ADD     R2, R2, #1
                BRnzp   Loop10
;
End10           ADD     R4, R4, #0
                BRp     Store10
                ADD     R2, R2, #0
                Brz     Post10
Store10         ADD     R2, R2, R5
                STR     R2, R1, #0 ; Store ASCII code for tens digit
                ADD     R1, R1, #1
Post10          ADD     R0, R0, #10 ; Correct R0 for one-too-many subtracts
Begin1          LD      R2, ASCIIoffset ; Prepare for "ones" digit
                ADD     R2, R2, R0
                STR     R2, R1, #0
                RET
;
ASCIIplus       .FILL x002B
ASCIIminus      .FILL x002D
ASCIIoffset     .FILL x0030
Neg100          .FILL xFF9C
Pos100          .FILL x0064
Neg10          .FILL xFFF6
ASCIIBUFF       .BLKW 4
```

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

10.24 Please correct the problem to read:

Suppose the keyboard interrupt vector is x34 and the keyboard interrupt service routine starts at location x1000. What can you infer about the contents of any memory location from the above statement?

Solution:

Memory location x0134 contains the address x1000.

9.7 Note: This problem belongs in chapter 10.

The three errors that arose in the first student's program are:

1. The stack is left unbalanced.

2. The privilege mode and condition codes are not restored.

3. Since the value in R7 is used for the return address instead of the value that was saved on the stack, the program will most likely not return to the correct place.