1. Explain why the R4 register is pushed and popped in the strfindn function.

```
-------------------------------------------------------------------------------------------
// int32_t strFindN(const char str[], char c, uint32_t n)
// address of str[0] in R0, character value in R1, instance in R2 // return index in R0 or -1 if not found
strFindN:
   PUSH   {R4}              // push value of R4 on stack
   MOV    R3, R0             // R3 = &str[0]
   MOV    R0, #0             // R0 = i = 0
strFindN_loop:
   LDRSB  R4, [R3, R0]        // R4 = str[i]
   CMP    R4, R1             // is str[i] == c?
   BNE    strFindN_next       // no: look for next char SUBS R2, R2, #1 // decrement instance
   BEQ    strFindN_end        // if instance==0, exit loop
strFindN_next:
   ADD    R0, R0, #1          // i++
   CMP    R4, #0             // is str[i] == NULL?
   BNE    strFindN_loop       // no: loop back
   MOV    R0, #-1            // yes: end of string found - no match
strFindN_end:
   POP    {R4}              // pop value of R4 from stack BX LR
   BX LR
-------------------------------------------------------------------------------------------
```

Answer:

R0 holds result

R1 holds char to find

R2 holds Instance Counter

R3 holds Address of str

R4 holds char we read from memory

Since we are using more than the four registers R0-R3 that we can freely use, based on the C programming API, we must store the value of any additional registers we use. We must also restore those values prior to returning from our function.

In this function we only need one more register, R4. We store the value of R4 with PUSH, and we restore the contents to R4 with POP.

```
.global isStrEqual                  @a
.global strConcatenate              @b
.global leftString                  @c
.global decimalStringToInt16        @d
.global hexStringToUint8            @e

.text

@ a. bool isStrEqual(const char str1[], const char str2[])
isStrEqual:
   LDRB    R2, [R0], #1       @ load next byte of str1
   LDRB    R3, [R1], #1       @ load next byte of str2
   CMP     R2, R3            @ check that both match
   MOVNE   R0, #0             @ No match, set result to ZERO
   BNE     isStrEqual_exit     @ No match, branch to exit
   CMP     R2, #0            @ did we reach end of 1st string?
   MOVEQ   R0, #1             @ reached end of 1st string and still match, set Result to ONE
   BEQ     isStrEqual_exit     @ everything Matched & at end of strings, branch to exit
   B       isStrEqual         @ branch to loop start (program start)
isStrEqual_exit:
   BX  LR

@ b. void strConcatenate(char strTo[], const char strFrom[])
strConcatenate:
   LDRB    R2, [R0], #1          @ load char of TO
   CMP     R2, #0               @ check for null of TO
   BNE     strConcatenate         @ not null, get next char
   SUB     R0, R0, #1            @ 'backup' address to the null
strCat_copy_loop:
   LDRB    R2, [R1], #1          @ load char of FROM & incr address
   STRB    R2, [R0], #1          @ store byte FROM in TO & incr address
   CMP     R2, #0               @ check for null of FROM
   BNE     strCat_copy_loop       @ null not found, copy next char
   BX  LR

@ c. void leftString(char * strOut, const char * strIn, uint32_t length)
leftString:
   CMP     R2, #0               @ check how many left to copy
   BEQ     leftString_exit        @ no more chars, then exit
   LDRB    R3, [R1], #1          @ load char from strIN & incr address
   STRB    R3, [R0], #1          @ store char to strOut & incr address
   SUB     R2, R2, #1            @ decr number of chars to copy
   CMP     R3, #0               @ check if we found null
   BNE     leftString            @ not null, loop
leftString_exit:
   BX  L
```

```
@ d. int16_t decimalStringToInt16(const char * str)
decimalStringToInt16:
   PUSH   {R4}              @ save R4 contents
   MOV    R1, R0            @ copy our pointer out of return register
byte
   MOV    R0, #0            @ set return value to zero
   MOV    R4, #10           @ our Base multiplier
   MOV    R3, #1            @ set the identity multiplier
   LDRB   R2, [R1], #1      @ load 1st char & post-increment
   CMP    R2, #'-'          @ is it minus sign?
   BNE    decimalString_loop    @ not '-', continue to processing
   MOV    R3, #-1           @ it was '-' so we need change identity multiplier
   LDRB   R2, [R1], #1      @ load next char only if minus sign
decimalString_loop:
   CMP    R2, #0            @ check if string end found
   BEQ    decimalString_end     @ go to the end
   SUB    R2, R2, #'0'      @ subtract 0x30 or 48
   CMP    R2, #0            @ check lower bound
   MOVMI  R0, #0            @ if R2 < 0 set return to zero
   BMI    decimalString_end     @ if R2 < 0 go to exit
   CMP    R2, #9            @ check if > 9 (upper bound)
   MOVGT  R0, #0            @ if R2 > 9, set return to zero
   BGT    decimalString_end     @ if R2 > 9 go to exit
   MUL    R0, R0, R4
   ADD    R0, R0, R2        @ add new digit to ones place
   LDRB   R2, [R1], #1      @ load the next digit
   B      decimalString_loop    @ now loop
decimalString_end:
   MUL    R0, R0, R3        @ multiply by our idenity value
   POP    {R4}              @ restore R4 contents
   BX LR

@ e. uint8_t hexStringToUint8(const char * str)
hexStringToUint8:
   MOV    R1, R0            @ copy our pointer out of return register
   MOV    R0, #0            @ set return value to zero
hexStringToUint8_loop:
   LDRB   R2, [R1], #1      @ load char & post-increment
   CMP    R2, #0            @ check if string end found
   BEQ    hexStringToUint8_end   @ go to the end
   CMP    R2, #48           @ compare char  "0"
   BLO    invalid_char_exit     @ if < "0" branch to to invalid_char_exit
   CMP    R2, #57           @ if > "9"
   BHI    hexString_hexDigit    @ > 9 means we arent 0-9
we process a 0-9 digit
   SUB    R2, R2, #48       @   subtract #48
   LSL    R0, R0, #4        @ mutliply current total by 2^4
   ADD    R0, R2            @ add the new digit
   B      hexStringToUint8_loop  @ loop to next char
hexString_hexDigit:
   CMP    R2, #65           @ compare char to "A" #65
   BLO    invalid_char_exit     @ if char <  "A" branch to invalid_char_exit
   CMP    R2, #70           @ compare char to "F" #70
   BHI    invalid_char_exit     @ if char >  "F" jump to invalid_char_exit
process A-F digit
   SUB    R2, #55           @ subtract #55
   LSL    R0, R0, #4        @ multiply current total by 2^4
   ADD    R0, R2            @ add_digit
   B      hexStringToUint8_loop  @ loop to next char
invalid_char_exit:
   MOV    R0, #0            @ set result to 0
hexStringToUint8_end:
   CMP    R0, #0xFF         @ check if > 255
   MOVHI  R0, #0            @ set return to 0 if > 255
   BX LR
```