Question 1:

The instructions used for accessing constant data or addresses within a program. The instruction `ldr r1, [pc, #n]` is an example of using a global variable or constants defined in the program in a high level language.

`ldr r1, [r2, r3]` can be used for accessing elements in arrays or dynamic structures, where `r2` could be the base address of the array or structure, and `r3` be the offset.

`ldr r1, [r2, #n]` can be used for accessing specific fields within data structures or arrays, kind of like OOP in which attributes and methods can be referenced with the constant offset - since objects wouldn't be scattered around in memory in general/

`ldr r1, [sp, #n]` would commonly be used for accessing local variables or function parameters that are stored on the stack.

Question 2

The `ldrh` instruction loads a 16-bit unsigned halfword from memory into the lower 16-bits of a 32-bit register. The `ldrsb` also loads a 16-bit quantity from memory into a 32-bit register, bit it sign-extends the halfword in order to preserve the sign of the original 16-bit value.

The difference between these two is that they treat the stored 16-bit numbers as either unsigned or signed bit-integer.

As for storing values, the instruction `strh` is sufficient since storing values doesn't require any subtleties to be known whether the bit-string was encoded signed or unsigned relative to the program.

Question 3

A potential solution is to try manually calculating the address of the variables and then use a register to load from that address:

```
add r3, sp, #offset @ Calculate the exact address where the variable is stored
ldr r1, [r3]        @ Load the variable using its address
```

`#offset` refers to the total offset from `sp` to the desired variable.

Question 4

We can try manually adjusting the stack pointer and use store and load instructions to simulate the two missing operations.

We can do the following:

```
sub sp, sp, #12      @ Allocate 12 bytes of stack space (3 registers * 4 bytes
each)
str r4, [sp, #0]     @ Store r4 at the start of the newly allocated space
str r5, [sp, #4]     @ Store r5 4 bytes into the allocated space
str lr, [sp, #8]     @ Store lr 8 bytes into the allocated space
```

This will have to be run before the subroutine in order to configure the stack pointer to be at the correct place. This will essentially do the same thing as `push`.

To simulate a `pop` instruction, we can follow the same logic of manually changing our stack pointer:

```
ldr lr, [sp, #8]     @ Load lr from the stack
ldr r5, [sp, #4]     @ Load r5 from the stack
ldr r4, [sp, #0]     @ Load r4 from the stack
add sp, sp, #12      @ Deallocate 12 bytes of stack space
```

Question 5

```
    ldr r0, =i         @ Load address of global variable i
    ldr r0, [r0]       @ Load value of i into r0

    add r1, sp, #60    @ Calculate address of j on the stack
    ldr r1, [r1]       @ Load value of j into r1

    add r2, r0, r1     @ r2 = i + j, the index for a[i+j] and b[i+j]

    lsl r2, r2, #2     @ Multiply index by 4 to get byte offset (since each int is
4 bytes)

    add r3, sp, r2     @ Calculate address of b[i+j] on the stack
    add r3, r3, #4     @ Adjust for b starting at offset 4 from the SP
    ldr r3, [r3]       @ Load b[i+j] into r3

    mul r3, r3, #3     @ Multiply b[i+j] by 3
```

```
    ldr r4, =a          @ Load address of global array a
    add r4, r4, r2      @ Calculate address of a[i+j]
    str r3, [r4]        @ Store 3*b[i+j] into a[i+j]
```

Question 6

Involving a hidden register to hold the first half of the address bits from the first 16-bit word of the `bl` instruction and combining these with the next 16-bit word to form the complete branch target address may work fine in normal execution conditions, however, some problems may arise when interrupts are enabled.

If an interrupt occurs after the first half of the `bl` instruction has been executed, but before the second half is completed, the content of the hidden register could be lost or become redundant - it may even be overwritten. Upon returning from the interrupt handler, the original `bl` instruction execution may be broken.

Although it requires less mechanisms to be built in for this way of executing to work, having to do more work in catering for the case of an interrupt-heavy environment, the alternative of `bl` containing all 32 bits seems like the more preferred choice.

Question 7

```
toupper:
    push {lr}               @ Save link register to return to caller
loop:
    ldrb r1, [r0], #1       @ Load byte from address in r0, post-increment r0
    cmp r1, #0              @ Compare loaded byte to null terminator
    beq end                 @ If byte is null terminator, end loop
    cmp r1, #'a'            @ Compare byte to ASCII value of 'a'
    blt loop                @ If byte is less than 'a', skip conversion
    cmp r1, #'z'            @ Compare byte to ASCII value of 'z'
    bgt loop                @ If byte is greater than 'z', skip conversion
    sub r1, r1, #32         @ Subtract 32 to convert lowercase to uppercase
    strb r1, [r0, #-1]      @ Store the converted byte back, pre-decrement r0 as
we incremented it earlier
    b loop                  @ Continue loop
end:
    pop {pc}                @ Return to caller
```

Question 8

Here's a recursive implementation of the factorial subroutine:

```
factorial_recursive:
    cmp r0, #1              @ Compare n to 1
    ble factorial_end       @ If n <= 1, end recursion
    push {r0, lr}           @ Save n and return address

    sub r0, r0, #1          @ Decrement n
    bl .factorial_recursive @ Recursive call with n-1

    pop {r1, lr}            @ Restore original n and return address
    mul r0, r1, r0          @ Multiply result with n (r1)

    b .factorial_end        @ Return to caller
factorial_end:
    bx lr                   @ Return
```

And here is the iterative alternative implementation of the factorial subroutine:

```
factorial_iterative:
    mov r1, #1          @ Initialize result to 1
loop_start:
    cmp r0, #1          @ Compare n to 1
    ble loop_end        @ If n <= 1, end loop

    mul r1, r1, r0      @ Multiply result with n
    sub r0, r0, #1      @ Decrement n
    b loop_start        @ Repeat loop
loop_end:
    mov r0, r1          @ Move result to r0
    bx lr               @ Return
```

A good example of a problem that can be solved with the use of recursion is the problem of any problem which uses dynamic programming. I'll pick one example of finding the maximum increasing subsequence in a sequence of numbers. Essentially, the optimal solution can be picked and then recursively unravelled in ordered sub-solutions which in-turn solve the whole solution elegantly.

Another problem that can be effectively solved with the use of recursion is the Tower of Hanoi problem, where `n` discs of decreasing order are stacked on tower 1 and need to be moved to tower k (classical Tower of Hanoi problems have k = 3). One rule of the stack moving is that in no way can we have a 'larger' disc on top of a smaller disc. The problem can be solved recursively by moving `n-1` discs from the starting tower to an auxiliary tower. Then, use divide and conquer to recursively solve the auxiliary towers until the whole problem is solved.

Question 9

Here are some macro definitions which I found useful for this question:

- `GPIO_IN` is a macro or constant representing the address of the input register for the General-Purpose Input/Output (GPIO) ports.
- `GET_BIT(x, n)` could be defined to extract bit `n` from variable `x`, typically using a bit-wise AND operation and a shift. A likely definition is `((x) >> (n)) & 1`.
- `BUTTON_A` and `BUTTON_B` are macros representing the bit positions within the GPIO input register that correspond to buttons A and B, respectively.

Seeing the implementation shown, I'm not sure exactly why this is an inefficient implementation (perhaps instead of dealing with the interrupt, it's always actively checking whether there's an interrupt?)

Does this work?

```
if (!((GPIO_IN >> BUTTON_A) & 1) || !((GPIO_IN >> BUTTON_B) & 1)) {
    // A button is pressed
}
```

(Stumped on this question)