Using a subroutine to process a set of Data words.

```
_start:  MOV  R4, #TESTNUM   // R4 has address of data
         MOV  R5, #0         // R5 will hold the result.
main:    LDR  R1, [R4], #4   // R1 gets the next Data word.
         CMP  R1, #0         // done whole list?
         BEQ  END
         BL   ONES           // parameter is passed in R1
         CMP  R5, R0         // result is returned in R0
         MOVLT R5, R0        // new best result?
         B    MAIN
END:     B    END

// subroutine to find longest string of 1's in R1
// Result is returned in R0

ONES:    ... code to shift/AND algorithm
LOOP:    :
         ADD  R0, #1
         B    LOOP
END_ONES: MOV  PC, LR
TESTNUMS: .word  0x103EF00F
          :
          .word  55555555
          .word  00000001
          .word  0
```

Nested subroutine.

If a subroutine, sub-1, needs to call a nested sub-routine Sub-2, then sub-1 must first save its copy of LR before executing BL sub-2. Using the stack for ARM

- Register R13 is dedicated for use as a stack ptr. ("S.P") ARM has two special "stack instructions":

```
push {Rx, Ry, ...., Rz} (*)
pop  {Rx, Ry, ...., Rz} (*)
```

(*) Same as :

```
SUB  R13, #4
STR  Rx, [R13]
  :
SUB  R13, #4
STR  Rz, [R13]
```

(*) Same as :

```
LDR  Rz, [SP]
ADD  SP, #4
  :
LDR  Rx, [SP]
ADD  SP, #4
```

- Nested subroutines

```
main() {                      MAIN: mov SP, #0X40000000 (*)
  :                                 :
```

```
    sub_1( ..... );
      :
}

sub_1( ..... ) {
      :
   sub_2( ... );
      :
}
sub_2( ..... ){
      :
}
```

```
        BL    sub_1
         :
sub_1:  // Assume sub-routine modifies R4, R5
        push {R4, R5, LR}   ⎫
         :                  ⎪
        BL   sub_2          ⎬ (*)
         :                  ⎪
        POP {R4, R5, LR}    ⎭
        MOV  PC, LR

sub_2:  // do stuff
         :
        mov  PC, LR
```

(*) SP gets decremented by 4 before use, So the first store will be at 0x3FFFFFFC which is the addr. of the bottom of our memory.

(*). order of operands inside { ..... } is irrelevant. the ARM processor always pushes larger register indices first (R14, R5, R4), and pop in the opposite order. (R4, R5, R14)

Accessing an I/O device. (prep for part 4)

Recall: ARM uses memory-mapped I/O to read/write I/O ports.

Some addr. in the DE1-SoC computer are:

LEDR(0xFF200000),

HEX(0xFF200020, 0xFF200030).

SW (0xFF200040) KEY (0xFF200050)

Example: endless loop that reads from SW and writes to LEDR

```
        LDR   R0, =0xFFF200000   // pointer to LEDR
        LDR   R1, =0xFF200040    // pointer to SW

MAIN:   LDR   R2, [R1]
        STR   R2, [R0]
        B     MAIN
```