Tuan

Question 1:

`ldr r1, [pc, #n]` – this can be used to load a value of a variable from the memory

`ldr r1, [r2, r3]` – this instruction may be used to access the i-th element of an array; r2 may store the address of the base and r3 might contain 4 times the index

`ldr r1, [r2, #n]` – this may be used when we want to jump to the last element of the array

`ldr r1, [sq, #n]` – after we recursively called a function and want to load from the memory a variable

`add r1, pc, #n`

`add r1, sp, #n` – when we want to move to a register

`add sp, sp, #n` – this may be used when we want to find the sum of an array, and we don't need the element that we have already added up

Question 2:

The difference between `ldrh` and `ldrsh` is that the former loads a halfword from the memory, pads it with zeros until it becomes a 32-bit word and puts in into a register. The second instruction takes a halfword from the memory, sign-extends it to form a 32-bit word and puts it into a register. We need two load instruction because for example the `byte` type can be signed and we want to do arithmetic with it. There is no need for a second store instruction because it only takes the bottom 16 bits and whether the number was signed or unsigned the result would be the same. We can substitute `ldrhs r1, [r2, #n]` by putting `n` into a register because we have encoding for `ldrhs r1, [r2, r3]`.

```
movs r3, #n
ldrhs r1, [r2, r3]
```

Question 3:

If the offset from the stack pointer is too big, the value can be stored into a register, and then use the `ldr` instruction to load the variable from the memory.

```
adds r2, sp, #n
ldr r1, [r2, #0]
```

Question 4:

Let's say that we want to push and pop `r4, r5, r6, r7`.

```
str r4, [sp, #-4]
str r5, [sp, #-8]
```

```
str r6, [sp, #-12]
str r7, [sp, #-16]
sub sp, #16

... @ some code

ldr r7, [sp, #4]
ldr r6, [sp, #8]
ldr r5, [sp, #12]
ldr r4, [sp, #16]
add sp, #16 @restore the previous sp
```

I subtract 4 for each of the registers from the `sp` because each of the registers take 4 bytes of space.

<u>Question 5:</u>

```
baz:
      push {r4-r7, lr}
      sub sp, #64             @ Allocate 64 bytes for array b and j
      ldr r4, =a              @ Assign r4 the address of the base of a
      ldr r5, =i              @ Assign r5 the address of i
      ldr r6, [sp, #60]       @ r6 has the value of j
      ldr r7, [r5]            @ r7 has the value of i
      add r6, r6, r7          @ r6 = i + j
      cmp r6, #9              @ check if i+j is in the array
      bmi i                   @ if N bit is 1, skip next instruction
      mul r6, r6, #4
      ldr r0, [sp, r6]        @ r0 = b(i+j)
      ldr r1, [r4, r6]        @ r1 = a(i+j)
      mul r1, r0, #3
      str r1, [r4, r6]        @ save r1 to the address of a(i+j)
      add sp. #64
      pop {r4-r7, pc}

      .bss
      .align 2
i:
      .space 4
      ...
      .align 2
a:
      .space 40
```

Question 6:
First I will write my Scala code, and then turn it into assembly language.

```
var r = 1;
// Invariant : row[0..k) = (j choose row.length) && 0 <= j < k
//          && 1 <= i <= j
while(r <= n){
  var i = 0
  while(i < r){
    if(i == 0){ row(i) = 1; i += 1 }
    else if(i == r - 1){ row(i) = 1 }
    else{ row(i) += row(i-1); i += 1 }
  }
  r += 1
}
row(k)
```

```
      .thumb_func
foo:
      push{r4-r7, lr}
      ldr r4, =row            @ Set r4 to the base of the array
      movs r3, #1
      str r3, [r4, #0]
      movs r5, #1             @ r5 will be the variable r
outer:
      adds r0, r0, #1
      cmp r5, r0              @ r0 will store n
      movs r6, #0             @ r6 will be the variable i
inner:
      cmp r6, r5              @ Compare i and r
      beq increment
      subs r7, r5, #1
      cmp r6, r7
      beq endofarray2
      cmp r6, #0
      beq endofarray1
      subs r6, r6, #1         @ i = i – 1
      lsls r2, r6, #2         @ r2 has the offset
      ldr r3, [r4, r2]        @ load row(i-1)
      adds r6, r6, #1
      lsls r2, r6, #2        @ r2 has the offset
```

```
      ldr r7, [r4, r2]       @ load row(i)
      adds r3, r3, r7        @ r3 stores the updated row(i)
      adds r6, r6, #1
increment:
      adds r5, r5, #1
      b outer
endofarray1:
      movs r3, #1
      lsls r2, r6, #2         @ Calculate the offset for row(i)
      str r3, [r4, r2]
      b increment
endofarray2:
      movs r3, #1
      adds r6, r6, #1
      lsls r2, r6, #2         @ Calculate the offset for row(i)
      str r3, [r4, r2]
      b increment
done:
      lsls r2, r1, #2
      ldr r0, [r4, r2]
      pop {r4-r7, pc}

      .bss
      .align 2
row:
      .space 1024
```

Question 7:

```
      .global foo
      .thumb_func
foo:
@ Compute Catalan(n) from the defining recurrence
@ ... using a static array and loops
      push {r4-r7, lr}          @ Save registers
@@ r0 = n, r3 = t, r4 = row, r5 = k, r6 = j, r7 = 4 * n
      movs r5, 0                @ k = 0
      ldr r4, =row
      lsls r7, r0, #2
      subs sp, [r4, r7]         @ Allocate n spaces for the array
      movs r1, #1
      str r1, [r4]              @ row[0] = 1
outer:
```

```
      cmp r5, r7               @ while (k < n)
      bge done
      movs r6, r5              @ j = k
      movs r3, #0              @ t = 0
inner:
      bgt indone
      movs r1, r6              @ put row[j] in r2
      ldr r2, [r4, r1]
      subs r1, r5, r6          @ put row[k-j] in r1
      ldr r1, [r4, r1]
      movs r0, #0              @ r7 = 4 * r0, so we can get rid of r0
loop:
      cmp r1, #0               @ Invariant: row(j)*row(k-j) = r1*r2 + r0
      beq continue
      subs r1, r1, #1
      adds r0, r0, r2
      b loop
continue:
      movs r2, r0
      adds r3, r3, r2          @ add to t
      subs r6, r6, #4          @ j -= 4
      cmp r6, #-4              @ while (0 <= j)
      b inner
indone:
      adds r5, r5, #4          @ k += 4
      movs r1, r5              @ row[k] = t
      str r3, [r4, r1]
      b outer
done:
      movs r1, r7             @ return row[n]
      ldr r0, [r4, r1]
      adds sp, [r4, r7]
      pop {r4-r7, pc}          @ restore and return
@ Statically allocate 256 words of storage
      .bss
      .align 2
```

a) Instead of writing `row: .space 1024`, we can subtract 4 * n from the stack pointer in the beginning of the program to allocate space for the array. Then at the end of the function we can add 4 * n to the stack pointer.

b) We can do `movs r6, r5` so the value of j can be k. Then we should compare `r6` to 0. Finally, we can move the test to the end of the loop. (The changes that I made are underlined)

c) Changing the `lsls` with `adds` and `subs`(in the case of `r6`).

d) The changes are the in **loop**.