DIGITAL SYSTEMS HT2018

SHEET 2

GABRIEL MOISE

Question 1

We'll discuss what each instruction does and what use it might have for a higher-level language:

- `ldr r1, [pc, #n]`

This loads in r1 the address obtained by adding the offset of n to the address from pc. It might be useful for storing a 32-bit quantity in a register (as shown in Lecture 6), if then we put the value from the address that is in r1, into another register, for example: `ldr r0, [r1]`

- `ldr r1, [r2,r3]`

This loads in r1 the value from the address that is obtained by adding a value from r3 to an address from r2. We can use this to get an element from an array that has the base address at r2 and the distance from it is r3 (if r3=4*i, we get the value from a[i]).

- `ldr r1, [r2, #n]`

This loads the value that is at the address obtained by adding the address stored in r2 with n, into r1. It can also be useful for arrays, where we know that the base address is r2 and the element we need is at a fixed distance from it.

- `ldr r1, [sp, #n]`

This loads the value from the address obtained by adding the address from sp with n, into r1. This can be used to access some value, knowing the distance we are from the current location of the stack pointer address.

- `add r1, pc, #n`

This stores in r1 the address obtained by adding to the program counter the offset n. This way, we can find the address of the next instruction we want to execute in the program.

- `add r1, sp, #n`

This stores in r1 the address obtained by adding to the stack pointer address the offset n. This way, we can find the address of a value of an array that has the base at the stack pointer location or find a value that is at a known distance from the stack pointer.

- `add sp, sp, #n`

This moves the stack pointer up by an offset of n, which can be useful if we want to have access to a variable that is at a known distance from the position we currently are.

Question 2

The instruction `ldrh r0, [r1, #4]` means: load the lower 16 bits (the halfword) from the 32-bit address obtained by adding 4 to r1. The bits at the positions 16-31 in r0 are set to be 0. This is the operation for unsigned halfwords.

However, the instruction `ldrsh r0, [r1, r2]` means: load the lower 16 bits (the halfword) from the 32-bit address obtained by adding the value from r2 to r1, and the bits at the positions 16-31 in r0 are set to be equal to the sign of the halfword, which is the 15th bit (the suffix sh comes from signed halfword).

The instruction `strh r0, [r1, #4]` stores the first 16 bits from r0 into the address obtained by adding 4 to the address of r1. There is no need for two functions here (for signed/unsigned) because we only store the first 16 bits of the 32-bit quantity that is currently in r0. The sign bit can be the 15th bit of that halfword or not, but we do not care.

As there is no encoding for (typo in the problem sheet, it's not lsrsh) ldrsh r1, [r2, #n], what we can do is store the value n in another register and use the value of that register instead. So, we would do:

```
movs r0, #n

ldrsh r1, [r2, r0]
```

Question 3

The encoding of the SP-relative load allows us to have an offset of 8 bits, so n can be at most 1024, as 2^8=256 and as n is always a multiple of 4, we can say that an offset is represented by its value over 4 in binary. So, n can be up to 1024 bytes (we work with bytes in the stack frame).

The question is, what do we do if we want to address variables that are farther away than that (that have an offset of more than 1024 bytes)?

To do that, we place the address of sp in register r0, and the value on n in register r2, and then we add then and load the value from the address obtained by adding r2 to r0 into r1:

```
mov r0, sp

ldr r2, =n

ldr r1, [r0, r2]
```

Question 4

To replace the push instruction we can manually store every register and lr into the stack. Instead of writing `push {r4-r7, lr}`, we write:

```
sub sp, #20

str r4, [sp, #0]

str r5, [sp, #4]
str r6, [sp, #8]

str r7, [sp, #12]

mov r0, lr

str r0, [sp, #16]
```

So we store each register manually after we have allocated enough space.

To replace the pop instruction we can manually load every register and pc from the stack. Instead of writing `pop {r4-r7, pc}`, we write:

```
ldr r4, [sp, #0]

ldr r5, [sp, #4]

ldr r6, [sp, #8]

ldr r7, [sp, #12]

ldr r0, [sp, #16]

mov pc, r0

add sp, #20
```

Question 5

```
baz:

     push {r4, r5, lr}
     sub sp, #64
     ldr r4, =a        @ r4 = base address of the global array a
     ldr r5, =i        @ r5 = global variable i
     ldr r2, [sp,#60]  @ r2 = j
     adds r2, r2, r5   @ r2 = i + j
     add r1, sp, #4    @ r1 = base address of local array b
     lsls r1, r2, #2   @ r1 = address of b(i+j)
     ldr r0, [r1]      @ r0 = b[i+j]
     movs r3, #3       @ r3 = 3
     muls r0, r3       @ r0 = 3*b[i+j]
     str r0, [r4, r2]  @ store 3*b[i+j] in a[i+j]
     add sp, #64
     pop {r4, r5, pc}

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

Question 6

```
        .syntax unified
        .text

        .global foo
        .thumb_func

      @ r0 = n
      @ r1 = k
      @ r2 = local use addresses
      @ r3 = the jth entry in the current line
      @ r4 = base address of the array row
      @ r5 = i
      @ r6 = j
      @ r7 = local use values from the Pascal triangle
      @ row[i][j] = the jth entry on the ith line of the Pascal
triangle


foo:
      push {r4-r7, lr}      @ Save all registers
      ldr r4, =row          @ Set r4 to base of array
      movs r5, #0                 @ i = 0
@ Invariant I : the ith row of the Pascal triangle is stored in the
stack frame
outer:
      movs r6, #0           @ j = 0
      movs r3, #1                 @ row[i][0] = 1
      str r3, [r4, #0]      @ store row[i][0] at offset 0
      adds r5, #1                 @ i = i + 1 && r3 = row[i-1][j]
      cmp r5, r0            @ if i>n, then the nth row is complete
      bhi done              @ so we are done
@ Invariant J : row[0..j] contains the first (j+1) numbers on the ith
line of the Pascal triangle && r3 = row[i-1][j]
inner:
      adds r6, #1                 @ j = j + 1
      cmp r5, r6            @ when i=j, we are done with inner,
      beq inner_done        @ as we calculated the ith row
      lsls r2, r6, #2       @ r2 = address of row[i-1][j]
      ldr r7, [r4, r2]
@ r7 = row[i-1][j] && r3 = row[i-1][j-1]
      adds r7, r7, r3       @ r7 = row[i][j]
      subs r3, r7, r3       @ r3 = row[i-1][j]
      str r7, [r4, r2]      @ store row[i][j] at the jth entry &&J
      b inner

inner_done:
      movs r3, #1                 @ row[i][j] = row[i][i] = 1
      lsls r2, r5, #2       @ calculate the address of row[i][j]
      str r3, [r4, r2]      @ store row[i][j] && I
      b outer               @ back to outer

done:
      lsls r2, r1, #2       @ return row[n][k]
      ldr r0, [r4, r2]
      pop {r4-r7, pc}       @ Restore regs and return

@ Statically allocate 256 words for the array
      .bss
      .align 2
row:
      .space 1024
```

Question 7

The modifications to the program lab1-asm/catalan.s needed for each subtask are in **bold**.

(a) To allocate space for the 1024-bytes array in the stack frame of the function, we do the following things:

```
foo:
        push {r4-r7, lr}        @ Save registers
        sub sp, #256            @ Allocate locals (1024 bytes)
        sub sp, #256            @ We allocate 4 times 256 bytes
        sub sp, #256            @ as we cannot allocate more than
        sub sp, #256            @ 508 at a time

        movs r5, #0             @ k = 0
        mov r4, sp
        movs r1, #1
        str r1, [r4]            @ row[0] = 1
        ...
done:
        lsls r1, r0, 2          @ return row[n]
        ldr r0, [r4, r1]
        add sp, #256            @ Deallocate locals (1024 bytes)
        add sp, #256
        add sp, #256
        add sp, #256
        pop {r4-r7, pc}         @ Restore and return
```

(b) To rewrite the inner loop to count down from k to 0, we will set j to k and then go down, the conditional branch will be bpl after we subtract 1 from r6 (which stores j) and we only go back in the inner loop if j is non-negative:

```
outer:
        cmp r5, r0              @ while (k < n)
        bge done

        movs r6, r5             @ j = k
        movs r3, #0             @ t = 0

inner:
        lsls r1, r6, #2         @ put row[j] in r2
        ldr r2, [r4, r1]
        subs r1, r5, r6         @ put row[k-j] in r1
        lsls r1, r1, #2
        ldr r1, [r4, r1]
        muls r2, r2, r1         @ multiply
        adds r3, r3, r2         @ add to t
        subs r6, r6, #1         @ j--
        bpl inner               @ if j is non-negative, we go back
indone:
        adds r5, r5, #1         @ k++
        lsls r1, r5, #2         @ row[k] = t
        str r3, [r4, r1]
        b outer
```

(c) To eliminate the left-shift instructions, we make the modifications that are indicated in bold:

```
        push {r4-r7, lr}        @ Save registers
@@ r0 = 4*n, r3 = t, r4 = row, r5 =4*k, r6 =4*j
        adds r0, r0, r0
        adds r0, r0, r0
@ Multiply n by 4 from the start
        movs r5, #0             @ k = 0
        ldr r4, =row
        movs r1, #1
        str r1, [r4]            @ row[0] = 1

outer:
        cmp r5, r0              @ while (k < n)
        bge done

        movs r6, #0             @ j = 0
        movs r3, #0             @ t = 0

inner:
        cmp r6, r5              @ while (j <= k)
        bgt indone
        movs r1, r6
      @ lsls r1, r6, #2         @ put row[j] in r2
        ldr r2, [r4, r1]
        subs r1, r5, r6         @ put row[k-j] in r1
      @ lsls r1, r1, #2
        ldr r1, [r4, r1]
        muls r2, r2, r1         @ multiply
        adds r3, r3, r2         @ add to t
        adds r6, r6, #4         @ j = j + 4
        b inner
```

```
indone:
        adds r5, r5, #4         @ k = k + 4
      @ lsls r1, r5, #2         @ row[k] = t
        str r3, [r4, r5]
        b outer

done:
        lsls r1, r0, 2          @ return row[n]
        ldr r0, [r4, r1]

        pop {r4-r7, pc}         @ Restore and return

@ Statically allocate 256 words of storage for the array
        .bss
        .align 2
row:
        .space 1024
```

(d) In this part, we replace the `muls r2, r2, r1` instruction with the procedure:

```
multiply:
        movs r7, #0             @ r7 = 0
        b test
again:
        lsrs r2, r2, #1         @ r2 = r2/2
        bcc even                @ if r2 was even, skip
        adds r7, r7, r1         @ r7 = r7 + r1
even:
        lsls r1, r1, #1         @ r1 = r1*2
test:
        cmp r2, #0              @ if r2 != 0
        bne again               @ repeat
        movs r2, r7             @ return r2
```

We use r7 because it has no use throughout the program. This is meant for the program to work on chips that don't have a hardware multiplier, too.

Question 8

At this question I will try to intuitively answer the questions:

- for an architecture that supports interrupts, the problem with the bl instruction might be that after it does its first part, of storing the first 11 bits of the address, it might be interrupted by another parallel instruction which can overwrite the 11 bits with other ones, and when we get back to put the remaining 11 bits and jump to the address we obtain we will most likely go in a different location(possibly out of the stack)
- by using a register that was not hidden, like lr, to store these bits, we can have them stored, and, in the case of an interrupt, we know where to get them back from and use them
- the risks associated with the use of lr would be that the lr register might also be used in the eventuality of an interrupt and overwritten, so this method might not work either