ARM

Monday, June 20, 2022 6:36 PM

Consider a natural number sequence in which, arbitrarily chosen the first number of the sequence c_0, the following elements in the sequence are obtained in the following way:

$$c_{i+1} = \begin{cases} \frac{c_i}{2} & if\ c_i\ is\ even \\ 3 * c_i + 1 & if\ c_i\ is\ odd \end{cases}$$

The sequence terminates when the number 1 is obtained.

Example 1. If c_0= 12, the sequence is: 12, 6, 3, 10, 5, 16, 8, 4, 2, 1. The sequence contains 10 elements.
Example 2. If c_0= 19, the sequence is: 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. The sequence contains 21 elements.

Specification 1 (8 points). Write in ARM assembly an iterative subroutine `iterativeCollatz` that receives in input a natural number, through a cycle calculates the sequence of Collatz, and returns the number of elements that compose the sequence.
The subroutine shall comply with the ARM Architecture Procedure Call Standard (AAPCS) standard, in particular with regard to switching the parameter to input/output and saving the registers.

Example 1: If the input parameter is 12, the return value is 10.
Example 2: If the input parameter is 19, the return value is 21.

Note: Since the only splitting operation is to calculate half of a number, `UDIV` and `SDIV` instructions cannot be used.

```
;SPECIFICATION1---------------------------------------------------------------------------
iterativeCollatz PROC
                 PUSH{r4-r11,LR}
                 MOV r4,r0                   ;saving in another register
                 MOV r5,#1                   ;counter for number of elements in the array
nextelement      TST r4,#1                   ;check LSB bit for even or not
                 MOV r6,r4                   ;for multplication by three
                 ITEEE EQ                    ;checking condition on the basis of even or not
                 LSREQ r4,#1                 ;c is even so dividing by 2 for next value
                 LSLNE r4,#1                 ;multiply by 2 in case of even
                 ADDNE r4,r6                 ;actually multiplying by three
                 ADDNE r4,#1                 ;adding one
                 ADD r5,#1                   ;incrementing the array elements counter
                 CMP r4,#1                   ;comparing array elements with one
                 BNE nextelement             ;go bac again untill one comes in the array
                 MOV r0,r5                   ;return with value of number of elements
                 POP{r4-r11,PC}
                 ENDP
```

For even z=1
For odd z=2

We compare value of z with 1

Specification 2 (6 points). Write in assembly ARM a subroutine `recursiveCollatz` that receives in input two parameters:
P1: a natural number corresponding to the c_i element of the sequence
P2: the number i

The subroutine `recursiveCollatz` modifies the two parameters in input performing the following operations:
- increases the second parameter by 1: $P2_{new} = P2 + 1$
- if the first parameter is equal to 1, the subroutine sets $P1_{new} = 1$ and then ends (return to calling program)
- otherwise, the subroutine calculates the c_{i+1} element of the sequence, sets $P1_{new} = c_{i+1}$ and then calls itself up with $P1_{new}$ and $P2_{new}$ as parameters.

Example 1: If the input parameters are P1 = 12 and P2 = 0, the subroutine (after 9 recursive calls) returns to the calling program $P1_{new} = 1$ and $P2_{new} = 10$.
Example 1: If the input parameters are P1 = 19 and P2 = 0, the subroutine (after 20 recursive calls) returns to the calling program $P1_{new} = 1$ and $P2_{new} = 21$.

Note: Since the only splitting operation is to calculate half of a number, `UDIV` and `SDIV` instructions cannot be used.

```
;SPECIFICATION2--------------------------------------------------------------------------
recursiveCollatz PROC
                PUSH{r4-r11,LR}
newiteration    ADD r1,#1                    ;increases the second parameter by 1: P2new = P2 + 1
                CMP r0,#1                    ;comparing first parameter with one
                ITT EQ                       ;if else conditions
                MOVEQ r0,#1                  ;if the first parameter is equal to 1, the subroutine sets P1new = 1
                BEQ returnloop               ;then ends (return to calling program);
                TST r0,#1                    ;check LSB bit for even or not
                MOV r6,r0                    ;for multplication by three
                ITEEE EQ                     ;checking condition on the basis of even or not
                LSREQ r0,#1                  ;c is even so dividing by 2 for next value
                LSLNE r0,#1                  ;multiply by 2 in case of even
                ADDNE r0,r6                  ;actually multiplying by three
                ADDNE r0,#1                  ;adding one
                B newiteration
returnloop
                POP{r4-r11,PC}
                ENDP
```

The IEEE-754 SP standard expresses floating-point numbers in 32 bits:

31	30 … 23	22 … 0
sign	exponent	mantissa

Bit 31 is 0 if the number is positive, 1 if negative.

Write the `addFPpositiveNumbers` subroutine, which receives in input two 32-bit numbers, considers them as IEEE-754 SP floating point numbers, and returns their sum (in the same format). Bit 31 of the two input numbers is always 0 (i.e., the two numbers are positive).

In details, the subroutine implements the following steps:

1. take the mantissa of the two parameters
2. set the 23rd bit of the mantissa to 1
3. compare the two exponents. If they are equal, the exponent of the result is the same. If they are different:
   - the exponent of the result is the highest one
   - shift right the mantissa of the number with the lower exponent by as many position as the difference between the two exponents.
4. sum the two mantissas: this is the mantissa of the result. If the 24st bit of the mantissa of the result is 1:
   - shift right the mantissa of the result by one position
   - increment the exponent of the result by one.
5. set the 23rd bit of the mantissa of the result to 0.
6. combine the mantissa and the exponent to get the final result.

Example

parameter1 = 0100 0010 0100 1011 0000 0000 0000 0000

parameter2 = 0100 0001 1010 0100 0000 0000 0000 0000

1. mantissa1 = 0000 0000 0100 1011 0000 0000 0000 0000
   mantissa2 = 0000 0000 0010 0100 0000 0000 0000 0000
2. mantissa1 = 0000 0000 1100 1011 0000 0000 0000 0000
   mantissa2 = 0000 0000 1010 0100 0000 0000 0000 0000
3. exponent1 = 1000 0100
   exponent2 = 1000 0011
   - exponentResult = 1000 0100
   - mantissa2 = 0000 0000 0101 0010 0000 0000 0000 0000
4. mantissaResult = 0000 0001 0001 1101 0000 0000 0000 0000
   - mantissaResult = 0000 0000 1000 1110 1000 0000 0000 0000
   - exponentResult = 1000 0101
5. mantissaResult = 0000 0000 0000 1110 1000 0000 0000 0000
6. result = 0100 0010 1000 1110 1000 0000 0000 0000

```
                AREA    |.text|, CODE, READONLY
param1 equ 2_01000010010010110000000000000000
param2 equ 2_01000001101001000000000000000000
; Reset Handler

Reset_Handler   PROC
                EXPORT  Reset_Handler              [WEAK]
                ldr r0,=param1
                ldr r1,=param2
                bl addFPposNum
stop b          stop
                ENDP

addFPposNum     proc
                push{r4-r11,lr}

;step1          getting mantissa 1 and mantissa 2
                ldr r6,=0x007fffff ;mask for mantissa
                and r4, r0, r6
                and r5, r1, r6

;step2          setting 23rd bit of mantissa to 1
                ldr r6,=0x00800000 ;mask for setting 23rd bit 1
                orr r4,r6
                orr r5,r6

;step3          compare exponents
                ;getting exponents
                ldr r6,=0x7f800000 ;mask for exponent
                and r7,  r0, r6 ;exp1
                and r8, r1, r6 ;exp2
                lsr r7,#23
                lsr r8,#23
                cmp r7,r8
                beq equal
                bgt firstIsHigher
                blt secondIsHigher

firstIsHigher   sub r11,r7,r8
                lsr r5, r11
                mov r9,r7
                b   cont

secondIsHigher  sub r11,r8,r7
                lsr r4, r11
                mov r9,r8 ;r9 has exponent of result
                b   cont

equal           mov r9,r7

cont            add r4,r5 ;sum of mantissas
                mov r5,r4 ;copy of sum
                and r5,#0x01000000 ;checking 24th bit of sum
                cmp r5,#0x01000000
                bne skip
                lsr r4,#1 ;mantissa of result>>1
                add r9,#1 ;rexponent of result+1

skip            and r4,#0xff7fffff
                lsl r9,#23
                add r0,r4,r9
```

Write the `computeF` subroutine in ARM assembly language, which computes the following function: $f(x, y) = (x + y/x) / 2$.
The subroutine receives in input:
1. x: a 32-bit value which represents an unsigned fixed-point number
2. n: the number of fractional digits in x
3. y: a 32-bit value which represents another unsigned fixed-point number
4. m: the number of fractional digits in y

The subroutine returns $f(x, y)$, with a number of fractional digits equal to $max(n, m)$.

Let a be the number of fractional digits of the dividend, b the number of fractional digits of the divisor, and c the position of the leftmost bit set to 1 in the dividend (the bits are counted from left to right, i.e., the most significant bit is at position 0). First of all, in order to increase the precision of the division, the dividend must be shifted left by c positions (in this way, the most significant bit is set to 1). Then, the result of the division between the two fixed-point numbers is computed as the integer quotient of their underlying integers. The number of fractional digits of the result is $d = a + c - b$.

In order to add two fixed-point numbers, they must have the same number of fractional digits. If the number of fractional digits is different, the fixed-point number with few fractional digits must be shifted left by a proper number of positions.

It is suggested to implement the division by 2 with a right shift.

Example: $x = 0.11$, $n = 2$, $y = 101.001$, $m = 3$.
The '.' separating integer and fractional digits is shown only for the sake of clarity.

Step 1: division.
$a = 3$, $b = 2$, $c = 26$.
y is shifted left by 26 positions
10100100000000000000000000000000 / 11 = 11011010101010101010101010101010
$d = 3 + 26 – 2 = 27$: the corresponding fixed-point number is 110.11010101010101010101010101010

Step 2: addition.
$n < d$, so x is shifted left by $d – n = 25$ position.
0.110000000000000000000000000 + 110.11010101010101010101010101010 =
= 111.10010101010101010101010101010

Step 3: the division by 2 gives 11.110010101010101010101010101
The result is approximated to $max(n, m)$ = 3 fractional digits: the subroutine returns 11.110

```
;SUBROUTINE-----------------------------------------------------------------
computeF PROC
         push{r4-r11,LR}
         mov r4,r0 ; x ;binary 00000011, 3 decimal
         mov r5,r1 ; n ;2
         mov r6,r2 ; y ;binary 00101001, 41 decimal
         mov r7,r3 ; m ;3
         mov r10,#0 ;count for bits

c_find   add r10,#1   ;counter updating loop ;r10++
         lsr r6,#1    ;right shift till all zeros
         cmp r6,#0
         bne c_find
         rsb r10,#32  ;32-c = 26 i.e. bits required to align y to left most bit

         lsl r2,r10    ;divident shifted left by c pos MSB 1
         udiv r2,r2,r4 ;shifted y / x

         sub r8,r10,r5 ; c-b
         add r8,r8,r7 ; d=a+c-b
         ;step 1 done here

         cmp r5,r8
         it lt
         sublt r9,r8,r5  ;d-n
                          ; subge r9,r5,r8 ;maybe
         lsl r4,r9

         add r4,r4,r2
         ;step 2 done here

         lsr r4,#1  ;div by 2

         cmp r1,r3
         bgt n_gt
         sub r8,r3  ;m greater
         lsr r4,r8  ;d - m bits right shift ;given example 27-3=24 i.e 11110 01010101010101010101010
         b skip

n_gt     sub r8,r1  ;n greater
         lsr r4,r8  ;d - n bits right shift
skip     mov r0,r4

         pop{r4-r11,PC}
         ENDP
```

```
Reset_Handler   PROC
                EXPORT  Reset_Handler              [WEAK]
                ldr r0,=x
                mov r1,#2 ;n
                ldr r2,=y
                mov r3,#3 ;m
                bl computeF
;                       ldr r0, =controlStatusRegister
;                       mov r1, #0  ;  step 1
;                       str r1, [r0]
;                       ldr r0, =reloadValueRegister
;                       mov32 r1, #1048576 ; example
;                       str r1, [r0]; step 2
;                       ldr r0, = currentValueRegister
;                       str r1, [r0]; step 3
;                       ldr r0, = controlStatusRegister
;                       mov r1, #7
;                       str r1, [r0]; step 4
stop            b stop
                ENDP

                area    systicHandlerData, data, readwrite
numIteration    space 4  ; 4 byte
arrayF          space 40 ; 40 bytes (10 elements)

                AREA    |.text|, CODE, READONLY
x equ 2_00000011
y equ 2_00101001

controlStatusRegister equ 0xE000E010
reloadValueRegister   equ 0xE000E014
currentValueRegister  equ 0xE000E018
```

The subroutine developed in the previous exercise can be called iteratively to compute the square root of a number y. In this exercise, we want to compute the square root of $y = 2022$.

1) In a data area, allocate space for:
   - *numIteration*: a 32-bit variable
   - *arrayF*: an array of 10 elements; each element is a word (32 bits)

   Inizialize *numIteration* to 0 and the first element of *arrayF* to 1, to be represented as a fixed-point value with 12 fractional digits. The other elements are not initialized.

2) In the Reset_Handler, configure the SYSTICK timer in order to generate an interrupt every 2^{20} clock cycles. Then, enter in an infinite loop.

   The SYSTICK timer is configured by means of the following registers:
   - Control and Status Register: size 32 bits, address 0xE000E010
   - Reload Value Register: size 24 bits, address 0xE000E014
   - Current Value Register: 24 bits, address 0xE000E018

The meaning of the bits in the Control and Status Register is as follows:
- Bit 16 (read-only): it is read as 1 if the counter reaches 0 since last time this register is read; it is cleared to 0 when read or when the current counter value is cleared
- Bit 2 (read/write): if 1, the processor free running clock is used; if 0, an external reference clock is used
- Bit 1 (read/write): if 1, an interrupt is generated when the timer reaches 0; if 0, the interrupt is not generated
- Bit 0 (read/write): if 1, SYSTICK timer is enabled; if 0, SYSTICK timer is disabled.

The Reload Value Register stores the value to reload when the timer reaches 0.
The Current Value Register stores the current value of the timer. Writing any number clears its content.

) In the handler of the SYSTICK timer, read the value of *numIteration* stored in memory. It counts the number of times the `computeF` subroutine has been called so far. Get the value k of the element of *arrayF* at position *numIteration*. Call the `computeF` subroutine with the following parameters: $x = k$, $n = 12$, $y = 2022$, $m = 0$.
Increment *numIteration* by 1 and save the new result in memory.
Get the value returned by `computeF` and save it in *arrayF* at position *numIteration* (after the increment, this is the first empty element of the array).
Note that the `computeF` subroutine has to be called 10 times (at 10 different SYSTICK timer interrupts) because this is the lenght of *arrayF*. At the end, the last element of *arrayF* stores the best approximation of the square root of 2022.

```
SysTick_Handler PROC
                EXPORT  SysTick_Handler            [WEAK]
                push{r4-r11,LR}

                ldr r4,=numIteration
                ldr r5,=arrayF

                mov r0,#1
                lsl r0,#12  ;fixed-point value with 12 fractional digits so lsl
                str r0,[r5] ;init first el of arrayF

                mov r6,#0    ;numIteration
                mov r10,#10  ;counter

                ; x = k,
sytickloop      mov r1,#12      ;n=12
                mov r2,#2022    ;y=2022
                mov r3,#0       ;m=0
                bl computeF


                add r6,#1
                str r6,[r4] ;  numIteration+=1

                str r0,[r5,r6,LSL #2]

                sub r10,#1
                cmp r10,#0
                bne sytickloop

                pop {r4-r11,PC}
                ENDP
```

```
                AREA    constants, DATA, READONLY
inputData   DCB 3, -14, 15, -92, 65, 35, -89

                AREA variables, DATA, READWRITE
outputData space 12

                AREA    |.text|, CODE, READONLY


; Reset Handler

Reset_Handler   PROC
                EXPORT  Reset_Handler              [WEAK]
                ldr     r0,=inputData
                ldr     r1,=outputData
                mov     r2,#7
                bl      copydata
                ldr     r0,=outputData
                bl      insertionSort
stop            b       stop
                ENDP


copydata    PROC
            PUSH{r4-r11, LR}
            mov r4,r0
            mov r5,r1

            mov r6,#0

loop        cmp r6,r2           ; check if not last element
            bge skip

             ldrb r7,[r0,r6]
```

Given two areas of memory, the first one containing byte constants and the second one being uninitialized, write the `copyData` subroutine in ARM assembly language, which copies the content of the first area of memory to the second one. The subroutine receives in input:
- the address of the first area of memory
- the address of the second area of memory
- the number of elements declared in the first area of memory

The procedure does not return any value.
The size of the second area of memory is higher than or equal to the size of the first area.
Example:

```
          AREA constants, DATA, READONLY
inputData DCB 3, -14, 15, -92, 65, 35, -89

          AREA variables, DATA, READWRITE
outputData space 12
```

Then, write the `insertionSort` subroutine, which receives in input:
- the address of an area of memory (`READWRITE`)
- the number of elements in the area (7 in the previous example)

The procedure does not return any value. It sorts the elements, rewriting the area of memory, by means of the insertion sort algorithm. The pseudocode of the insertion sort is the following (A is the array to sort):

```
1.  i ← 1
2.  while i < length(A)
3.      x ← A[i]
4.      j ← i - 1
5.      while j >= 0 and A[j] > x
6.          A[j+1] ← A[j]
7.          j ← j - 1
8.      end while
9.      A[j+1] ← x
```

```
cont            add r4,r5 ;sum of mantissas
                mov r5,r4 ;copy of sum
                and r5,#0x01000000 ;checking 24th bit of sum
                cmp r5,#0x01000000
                bne skip
                lsr r4,#1 ;mantissa of result>>1
                add r9,#1 ;rexponent of result+1

skip            and r4,#0xff7fffff
                lsl r9,#23
                add r0,r4,r9

                pop{r4-r11,pc}
                endp
```

6. result = 0100 0010 1000 1110 1000 0000 0000 0000

The aliquot sum *s(n)* of a natural numbers *n* is the sum of all proper divisors of *n*, that is, all divisors of *n* other than *n* itself. For example, the aliquot sum of 28 is: s(28) = 1 + 2 + 4 + 7 + 14 = 28.
A natural number equal to its aliquot sum is called a *perfect number*. For example, 28 is a perfect number.
Two different natural numbers related in such a way that the aliquot sum of each is equal to the other number are called *amicable numbers*. For example, 220 and 284 are amicable numbers:
s(220) = 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284
s(284) = 1 + 2 + 4 + 71 + 142 = 220
Numbers whose aliquot sums form a periodic sequence are called *sociable numbers*. They are generalizations of the concepts of perfect numbers and amicable numbers. For example:
s(12496) = 1 + 2 + 4 + 8 + 11 + 16 + 22 + 44 + 71 + 88 + 142 + 176 + 284 + 568 + 781 + 1136 + 1562 + 3124 + 6248 = 14288
s(14288) = 1 + 2 + 4 + 8 + 16 + 19 + 38 + 47 + 76 + 94 + 152 + 188 + 304 + 376 + 752 + 893 + 1786 + 3572 + 7144 = 15472
s(15472) = 1 + 2 + 4 + 8 + 16 + 967 + 1934 + 3868 + 7736 = 14536
s(14536) = 1 + 2 + 4 + 8 + 23 + 46 + 79 + 92 + 158 + 184 + 316 + 632 + 1817 + 3634 + 7268 = 14264
s(14264) = 1 + 2 + 4 + 8 + 1783 + 3566 + 7132 = 12496.

You have to write the `isSociable` subroutine in ARM assembly language, which receives a natural number *n* in input, and repetitively computes the sequence of aliquot sums, until one of the following conditions occurs:

a) the last computed aliquot sum is equal to the input parameter (i.e., the number is perfect, amicable, or sociable)
b) the last computed aliquot sum is equal to 1 (i.e., the number is neither perfect, amicable, or sociable)
c) 8 terms in the sequence have been computed (i.e., the number is neither perfect or amicable and probably is not sociable).

In the first case, the subroutine returns the length of the sequence. In the last two cases, it returns 0. For example:

- with *n* = 28, the subroutine computes s(28) = 28 and immediately stops returning 1
- with *n* = 220, the subroutine computes s(220) = 284, then s(284) = 220 and returns 2
- with *n* = 12496, the subroutine computes 5 aliquot sums and finally it returns 5
- with *n* = 100, it computes s(100) = 117, s(117) = 65, s(65) = 19, s(19) = 1. It stops returning 0.

In order to compute the aliquot sum of *n*, you have to use the following algorithm:

1) initialize *sum* = 1, *a* = 2
2) while(true):
3)   if *n* is divisible by *a*:
4)   *b* = *n* / *a*
5)   if *a* < *b*:
6)   *sum* = *sum* + *a* + *b*
7)   else:
8)   if *a* == *b*:
9)   *sum* = *sum* + *a*
10)   exit while and stop
11)   *a* = *a* + 1

```
Reset_Handler   PROC
                EXPORT  Reset_Handler           [WEAK]
                mov     r0,#28      ;returns 1
;               mov     r0,#220     ;returns 2
;               mov     r0,#12496   ;returns 5
;               mov     r0,#100     ;returns 0
                bl      subroutine1
stop            b       stop
                endp

subroutine1     PROC
                PUSH{r4-r11,LR}
                mov r4,r0           ; n
                mov r5,#1           ; sum = 1
                mov r6,#2           ; a = 2
                eor r11,r11         ; counter for max terms

whileloop
;check if n is divisible by a i.e remainder = 0
                udiv r7,r4,r6
                mul  r7,r6          ; quotient*divisor
                sub  r7,r4,r7       ; remainder = quotient*divisor

                cmp r7,#0
                bne next
                udiv r8,r4,r6       ; b = n/a

;checking if a<=b or not

                cmp r6,r8           ; a=r6, b=r8
                bge elseCondition
                add r9,r6,r8        ; a+b

                add r5,r9           ; sum = sum + (a+b)
                b   next

elseCondition   cmp r6,r8
                bne check1
                add r5,r6           ; sum = sum + a

next            add r6,#1           ; a = a + 1
                b   whileloop

check1          add r11,1           ; length of sequence++
                cmp r0,r5           ; check if sum = original number
                bne check2
                mov r0,r11          ; return length
                b   exit

check2          cmp r5,#1           ; check if sum = 1
                bne check3
                b   return0         ; return 0

check3          cmp r11,#8          ; check if length = 8
                beq return0         ; return 0, if true

                mov r4,r5           ; new n
                mov r5,#1           ; reset sum
                mov r6,#2           ; reset a
                b   whileloop

return0         mov r0,#0

exit            POP{r4-r11,PC}
                ENDP
```

```
copydata    PROC
            PUSH{r4-r11, LR}
            mov r4,r0
            mov r5,r1

            mov r6,#0

loop        cmp r6,r2           ; check if not last element
            bge skip

            ldrb r7,[r0,r6]
            strb r7,[r1,r6]     ;copy value from r0+offset i.e. input to r1+offset i.e ouput
            add r6,#1
            b   loop
skip
            POP{r4-r11,PC}
            ENDP

insertionSort   PROC
                PUSH{r4-r11, LR}
                mov r4,r0           ;address of area of memory
                mov r5,#7           ;number of elements in the area

                mov r6,#1           ;i
while           cmp r6,r5           ;check i < length
                bge endwhile

                ldrsb r7,[r4,r6]      ; x <- A[i]
                sub r8,r6,#1        ; j= i - 1

innerWhile      cmp r8,#0           ;check j>=0
                blt end_innerWhile

                ldrsb r9,[r4,r8]        ; A[j]
                cmp r9,r7
                ble end_innerWhile

                add r10,r8,#1       ;j+1
                strb r9,[r4,r10]        ;A[j+1] <- A[j]
                sub r8,#1
                b   innerWhile
end_innerWhile
                add r10,r8,#1               ;j+1
                strb r7,[r4,#10]            ;A[j+1] <- x
                add r6,#1
                b   while
endwhile
                POP{r4-r11, PC}
                ENDP
```

array to sort):

1. i ← 1
2. while i < length(A)
3.   x ← A[i]
4.   j ← i - 1
5.   while j >= 0 and A[j] > x
6.   A[j+1] ← A[j]
7.   j ← j - 1
8.   end while
9.   A[j+1] ← x
10.   i ← i + 1
11. end while

In the example above, if the subroutine receives the address of `outputData`, at the end the area contains the values `-92, -89, -14, 3, 15, 35, 65`.