11/27/2022

ASSIGNMENT 2

COS10004 COMPUTER SYSTEM

STUDENT NAME: VI LUAN DANG
STUDENT ID: 103802759

Table of content

Assignment 2 testing video:

The video can be view at this link: https://www.youtube.com/watch?v=V_Pjp78De7I

Stage 1 description

Stage 1 consists of 3 smaller stages including:
- Finding the minimum value

- Finding the maximum value

- Finding the difference between minimum and maximum vale

We will discuss each smaller stage below.

Finding the minimum value – Stage1A

```
stagela_min:
  cmp r0, r1
  bls min2
  b min1

min1:
 cmp r1,r2
 movle r0,r1
 movge r0,r2
 b end1

min2:
 cmp r0,r2
 bls end1
 mov r0,r2
 b end1

end1:
bx lr
```

First of all, we will compare r0 with r1, if r0 is smaller, we can continue to compare r0 with r2 in the min2 function. On the scenario that r0 is also smaller than r2, then we don't need to do anything because r0 will automatically be passed to kernel.

If r0 is greater than r1 however, we will proceed to compare r1 with r2 and depending on the result, we will move the smaller value onto r0 and end this function. In min1, Movle means

move if the above compare results in a Z flag being turned on or N flag is different from V flag (lesser than or equal). Movge means move if the above compare results in a N flag equal to V flag ,or in other words, greater or equal.

Finding the maximum value – Stage1B

```
stage1b_max:
  cmp r0, r1
  bhs max2
  b max1

max1:
 cmp r1,r2
 movle r0,r2
 movge r0,r1
 b end1

max2:
 cmp r0,r2
 bhs end1
 mov r0,r2
 b end1
```

Stage 1B is not so different to stage 1A, however, this time we have to use BHS, which is branch if the above comparison is higher or same. We also need to change which register will be passed onto r0 in max1.

Finding the difference value – Stage1C

```
stage1c_diff:
    push {r8,r9}

    push {r0,r1,r2,lr}
    bl stage1a_min   ;call min
    mov r8, r0      ;get the min value
    pop  {r0,r1,r2,lr}

    push {r0,r1,r2,lr}
    bl stage1b_max    ;call max
    mov r9, r0        ; get max value
    pop  {r0,r1,r2,lr}

    sub r0, r9, r8    ;find the difference
    pop {r8,r9}

    b end1
```

Stage 1C is the easiest in stage 1 as we have all we need to make it works. Firstly, we need to push r0,r1,r2,lr onto our stack as during the function calling, these values will be changed so we need to preserve them. After that we move the minimum value onto r8. Finally, we need to pop the preserved values for stage1B to use. Similarly, the process of getting the maximum value is not different from the previous process. Ultimately, we get our maximum value stored onto r9.

Finally, we subtract r9 (the maximum value) with r8 (the minimum value) to get the difference and store the result onto r0.

Stage 2 description

Stage 2 requires us to flash the LED in accordance with the initialized array in the INIT file.

```
stage2_flash_array:
        push {r8}
        loop:
          ldr r8, [r2], #4 ;load the current index onto r8
          push {lr, r1-r2}
            mov r1,r8   ;initialized registers for flashing and pausing
            mov r2, $50000
            bl FLASH
            mov r1, $120000
            bl PAUSE
          pop {lr, r1-r2}
          sub r1, #1  ;minus the size, if the size reaches 0 then end the loop
          cmp r1, #0
         bgt loop
         pop {r8}

        b end1
```

This can be done by loading the first value of the array onto r8 using the LDR command. We then need to push lr,r1 ,and r2 onto the stack as we are about to call a function and we need these values for future use. After that the first value is store onto r1 and the FLASH function is called to flash according to the value of r8, the process is similar with PAUSE function.

After calling FLASH and PAUSE, we will subtract the size of the array and check if it is zero or not. BGT means branch if the above comparison is greater, therefore, if r1 is greater than 0 (meaning that the array still has value) the loop will continue.

Stage 3 description

Stage 3 requires us to sort the initialized array in the INIT file and flash the sorted value in the array.

```
stage3_bubblesort:
        push {r8-r11}

        push {r0}
        loop1:
                mov r11, r1
                ldr r8, [r11], #4 ;load the value of the current index onto r8
                mov r10, #1  ;index variable
                loop2:
                        ldr r9, [r11], #4 ;load the next value onto r9
                        cmp r8, r9
                        ble swap  ;if current value is less than the next value then swap them otherwise store the
                        str r8, [r1, r10, LSL #2]  ;lsl #2 means multiply by 4 therefor we can access a memory address of an index by lsl it by 2
                        sub r10, #1                ;this code means store r8 into r1 + memory address of r10
                        str r9, [r1, r10, LSL #2]  ; similar to r8
                        add r10, #1
                        mov r9, r8
                        swap:
                        mov r8, r9
                        add r10, #1 ;move to the next index
                        cmp r10, r0 ;compare if index is reaching end
                bls loop2  ;if not reaching the end then continue the loop
                sub r0, #1 ;minus the size to end if reaching 0
                cmp r0, #0
        bgt loop1
        pop {r0}
        push {lr, r0-r2} ;pushing the preserved register for future use
                mov r8, r2 ;initialized para for flashing
                mov r9, r1
                mov r1, r0
                mov r0, r8
                mov r2, r9
                bl stage2_flash_array
        pop {lr, r0-r2}
        pop {r8-r11}
bx lr
```

My sorting process can be divided into 3 stages: get the first value, compare, and swap if needed.

The process starts with loading the first value of the array onto r8, we then get our index variable onto r10 and go to a loop.

In this loop we will load the next value of the array onto r9 and compare them, if the comparison yields that r8 is smaller than r9 (BLE means branch if smaller or equal) then they are in their place so we will swap r9 to be our first value and compare it with the next value in the array.

If r8 is greater than r9 ,however, this will mean that they are not in the right location, so we will swap their location in the memory (LSL #2 is used in this particular case), let's say the location of our array in the memory is 0x00000000 then the first value will be located at 0x00000004. That is the reason why we LSL #2 because LSL #2 means multiple by 4, so for each index if we

multiply by 4 we will get its location in the memory. We also need to subtract the index and add the index to avoid swapping the incorrect value.

The process continue until the array reaches the end and we have our sorted array.

Finally, we will store the needed registers for stage2 to flash our array.

Stage 4 description

Stage 4 is just the same as stage 3 but this time we have to use quick sort rather than bubble sort. In this stage I decided to use the Median of 3 method to implement our quicksort so that it will be more efficient. As it ensures that one common case remains optimal, it is more difficult to manipulate into giving worst case.

```
stage4_quicksort:
push {lr} ;call the sorting function
bl qfunction
pop {lr}

mov r0, BASE  ;flash the sorted array
mov r1, 8
adr r2, numarray2
push {lr}
bl stage2_flash_array
pop {lr}
b end1



qfunction:
qsort:
push {r0 - r10, lr}
mov r4,r1 ;address of array
mov r5,r0 ; size of array
cmp r5,#1 ;if size less than 1 done
ble qsort_done
cmp r5, #2 ;if size is 2 then compare two number
beq qsort_check
```

```
qsort_partition:
mov r1,#1
lsr r2,r5,r1 ;find the middle element index
ldr r6,[r4] ;value of the first
ldr r7,[r4,r2,lsl #2] ;value of the middle
sub r8,r5,#1 ;find the last element index
ldr r8,[r4,r8,lsl #2] ;value of the last
cmp r6,r7 ;compare and sort 3 values to find
movgt   r9,r6
movgt   r6,r7
movgt   r7,r9
cmp     r7,r8
movgt   r9,r7
movgt   r7,r8
movgt   r8,r9
cmp     r6,r7
movgt   r9,r6
movgt   r6,r7
movgt   r7,r9
mov r6,r7 ;pivot
mov r7,#0  ;index of the first element in bounds
sub r8,r5,#1 ;index of the last element in bounds

qsort_loop:
ldr r0,[r4,r7,lsl #2] ;lower value
ldr r1,[r4,r8,lsl #2] ;upper value
cmp r0,r6 ;compare lower value to pivot
beq qsort_loop_u  ;if = do nothing
addlt r7,r7,#1  ;if < move to next
strgt r0,[r4,r8,lsl #2]  ;if > swap value
strgt r1,[r4,r7,lsl #2]
subgt r8,r8,#1  ;decrease upper index
cmp r7,r8  ;if index are the same recurse
beq qsort_recurse
ldr r0,[r4,r7,lsl #2] ;lower value
ldr r1,[r4,r8,lsl #2] ;upper value
```

```
qsort_loop_u:
cmp r1,r6  ;compare upper value to pivot
subgt r8,r8,#1 ;if > decrease
strlt r0,[r4,r8,lsl #2] ;if < swap value
strlt r1,[r4,r7,lsl #2]
addlt r7,r7,#1 ;increase lower index
cmp r7,r8 ;if the same recurse
beq qsort_recurse
b qsort_loop ;continue the loop

qsort_recurse:
mov r1,r4 ;location of the first bucket
mov r0,r7 ;length of the first bucket
bl qsort ;sort the bucket
add r8,r8,#1 ;index past final index
cmp r8,r5 ;compare final index to original length
bge qsort_done ;= return
add r1,r4,r8,lsl #2 ;location of the second bucket
sub r0,r5,r8 ;length of the second bucket
bl qsort ;sort second buccket
b qsort_done ;return

qsort_check:
ldr r0, [r4] ;load first value in r0
ldr r1, [r4,#4] ;load second in r1
cmp r0,r1
ble qsort_done ;if less than then done
str r1,[r4] ;otherwise swap
str r0,[r4,#4]
```

```
qsort_done:
pop {r0-r10,lr}
;cmp r11, #1
;beq flashend
bx lr

;flashend:
;push {r0-r2,lr}
;mov r0, BASE
;mov r1, 8
;adr r2, numarray2
;bl stage2_flash_array
;pop {r0-r2,lr}
;bx lr
```

The code seems a little bit mouthful, but I will explain my process in

First, we will push our needed registers onto our stacks (this will be push each time our recursion occurs) and if the size of our array is less than one then it's done, if there are only two values then we only need to compare those two. We then move to partition stage to find the pivot.

Rather than picking a random pivot at the end or middle or the beginning of the array, I will take the beginning index value, the middle index value, and the last index value to compare them together. Our pivot will be the middle value after we have compare them all. We also store our first element in bounds onto r7 and index of the last element in bounds onto r8.

Moving to the loop process, we will store the lower value and upper value onto r0 and r1 consecutively and compare the lower value with the pivot, this might result in 3 scenarios:

- If it is equal, then we move to compare the upper value.

- If it is lesser, then we move to the next lower value as the lower value is in its right position.

- If it is greater, then we have to swap the upper value and the lower value

After that we will compare r7 and r8 if they are the same then we will recurse.

The process of comparing the upper value is not so different:

- If it is larger, then it is in the right position.

- If it is lesser, we have to swap the upper value with the lower value.

The recursion steps are just a smaller version of what we just did above and it will compare each bucket to get us the sorted list.

The check function is used to compare the two values, if the lower value is lesser than the upper value then it is in the right place, and we are done. If they are not, then we swap them.

After sorting the array we will move them to the appropriate registers and proceed to flash them using Stage 2 function.