University of Dhaka

Department of Computer Science and Engineering

CSE-3113 : Microprocessor and Assembly Language Lab

Experiment 5: To understand and have familiarize with register based assembly programming for Cortex M4 processor for handling non-recursive, recursive and nested function call.

Submitted By:

Name: Mahmudul Hasan

Roll No : 20

Submitted On :

April 09, 2023

Submitted To :

Dr. Upama Kabir, Professor

Dr. Md. Mustafizur Rahman, Professor

1 Introduction

The Cortex-M4 is a widely used processor in embedded systems, particularly in microcontrollers. Understanding how to program the Cortex-M4 using assembly language can provide significant benefits, such as increased performance, reduced memory usage, and the ability to perform low-level operations. In this lab, we will focus on register-based assembly programming for the Cortex-M4 processor and explore how to handle non-recursive, recursive, and nested function calls.

2 Objectives

The main objective of this lab is to provide hands-on experience with assembly language programming for the Cortex-M4 processor, particularly in the context of function calls. By the end of the lab, we should be able to:

1. Understand the ARM Procedure Call Standard (APCS) for function calls on the Cortex-M4 processor.
2. Implement non-recursive, recursive, and nested function calls using register-based assembly programming.
3. Use the stack efficiently to save and restore registers and function call parameters.
4. Optimize function calls for performance and memory usage.
5. Debug and troubleshoot assembly language programs for the Cortex-M4 processor.

3 Theory

Functions are fundamental components of code that allow for modularity and reusability, leading to more efficient and streamlined programs. When a function is called, control is transferred from the calling function to the called function (callee), which performs the requested task and returns control to the calling function. To handle function calls properly in ARM programming, the ARM Procedure Call Standard (APCS) must be followed.

When a function is called, the BL instruction is used to store the return address in R14 (LR). Registers R0-R3 (a1-a4) are used to pass argument values into a function and to return a result value from a function. Registers R4-R8, R10, and R11 (v1-v5, v7, and v8) are used to hold the values of a function's local variables. These registers must have unchanged values when control returns to the calling function, and if the called function needs these registers for extra workspace, then it must preserve the contents of the registers R4-R8, R10, R11, and SP (callee saved).

The stack is used to store values in memory, and the ARM uses the "full descending" approach, where SP points to the topmost filled location. Multiple register values can be stored in a block using the LDM and STM instructions, and data stored on the stack forms part of the stack frame for that function invocation. The stack frame contents need to be popped when the function is exited and returned to the caller. All the local variables vanish when this situation occurs, and the lowest-address item goes to the lowest numbered registers.

It is important to note that in a leaf function, where the function does not call any other function, we do not need to store LR. However, in all other cases, we must do so. Properly handling function calls can greatly impact the efficiency and performance of a program, and following the ARM Procedure Call Standard and using the stack can lead to more streamlined and efficient code. By understanding these concepts, programmers can create more effective and efficient programs.

4 Methodology

4.1 Write an assembly language program to identify prime numbers from a list of array by calling a function called prime.

Solution:

```
    AREA Task1, CODE, READONLY
    ENTRY
    EXPORT main
; define the array of integers
arr         DCD 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 17, 19, 23, 29
; define the prime function
; input: integer in r0
; output: 0 if not prime, non-zero if prime
prime
    MOV r1, #2              ; initialize divisor to 2
    CMP r0, #1              ; check if input is 1
    BEQ not_prime           ; branch if input is 1
loop
    CMP r0, r1              ; compare input to divisor
    BEQ is_prime            ; branch if input equals divisor
    MOV r2, #0              ; clear remainder register
    MOV r3, r0              ; copy input to working register
    SDIV r3, r3, r1         ; divide input by divisor
    MUL r3, r3, r1          ; multiply quotient by divisor
    SUB r2, r0, r3          ; calculate remainder
    CMP r2, #0              ; compare remainder to 0
    BEQ not_prime           ; branch if remainder is 0
    ADD r1, r1, #1          ; increment divisor
    B loop                  ; repeat loop with new divisor
is_prime
    MOV r0, #1             ; set return value to 1 (prime)
    BX lr                  ; return to calling function
not_prime
    MOV r0, #0            ; set return value to 0 (not prime)
    BX lr                 ; return to calling function

main
    MOV r4, #0              ; initialize array index to 0
    MOV r5, #10             ; set array size to 10
    LDR r7, =arr
    BL check_primes

check_primes
    CMP r4, r5              ; compare array index to size
    BEQ STOP       ; branch if array is complete
    LDR r0, [r7]                    ; reading array element
    ADD r7, r7, #4                    ; incrementing array position
    ADD r4, r4, #1          ; increment array index
    BL prime
    B check_primes          ; repeat loop with next integer

STOP B STOP
    END
```

4.2 Write an assembly language program to perform the summation of two numbers by call a function sum(arg1, arg2) using call-by-value.

Solution:

```
    AREA sum_CallByValue, CODE, READONLY
    ENTRY
    EXPORT main
addd
    ; This function takes two arguments passed by value
    POP{r3, r4}                ; arg1: r0, arg2: r1
    ADD r0, r3, r4     ; Add the two arguments
    PUSH{r0}                  ; Pushing result to stack
    BX  lr             ; Return to caller
    ENDP

; main function
main
    ; Initialize the arguments and call the sum function
    MOV r0, #10         ; arg1 = 10
    MOV r1, #20         ; arg2 = 20

     PUSH {lr}                  ;pushing return address to stack
    PUSH {r0, r1}         ;Pushing arguments to the stack
    BL  addd          ; Call addd(arg1, arg2)
    ; The result is now in r0
    ; Do something with the result
    POP{r5}                           ; getting the result from stack
    POP {pc}                   ;restoring return address to program counter
    MOV r0, #0        ; Return 0 to OS
    BX  lr            ; Return from main function
```

4.3 Write an assembly language program to perform the summation of two numbers by call a function sum(arg1, arg2) using call-by-reference

Solution:

```
    AREA sum_CallByReference, CODE, READONLY
    ENTRY
    EXPORT main
addd
    ; This function takes two arguments passed by value
    POP{r3, r4}                ; arg1: r0, arg2: r1
    ADD r0, r3, r4    ; Add the two arguments
    PUSH{r0}                 ; Pushing result to stack
    BX  lr            ; Return to caller
    ENDP

; main function
main
    ; Initialize the arguments and call the sum function
    MOV r0, #10        ; arg1 = 10
    MOV r1, #20        ; arg2 = 20

     PUSH {lr}                ;pushing return address to stack
    PUSH {r0, r1}        ;Pushing arguments to the stack
    BL  addd        ; Call addd(arg1, arg2)
    ; The result is now in r0
    ; Do something with the result
    POP{r5}                       ; getting the result from stack
    POP {pc}                ;restoring return address to program counter
    MOV r0, #0      ; Return 0 to OS
    BX  lr          ; Return from main function

```

4.4 Write an assembly language program to find and return the minimum element in an array, where the array and its size are given as parameters by using recursive function

Solution:

```
        AREA minval, DATA, READWRITE
arr DCD 5, 4, 1, 2, -1, 100 ; array containign integer values
sz DCD 6 ; array size

        AREA MAIN, CODE, READONLY
        ENTRY
        EXPORT main

min
        POP{r1, r2} ; r1=size, r2=min_element
        CMP r1, #0
        BEQ return
        LDR r3, [r0]
        ADD r0, r0, #4        ; incementing array size
        SUBS r1, r1, #1        ; size = size -1
        CMP r3, r2
        MOVLT r2, r3        ; if the current value is less than the min_element, than update the
        PUSH{r1, r2}; pushing the arguments to the stack
        BL min                          ; making recursive call to the function

return
        BX lr
main
        LDR r0, =arr ; loading array address to r0
        LDR r1, =sz          ; loading array size
        LDR r1, [r1]
        MOV r2, #999 ;considering max value possible in the array is 999
        PUSH{lr, r1, r2}
        BL min
STOP B STOP
        END
```

4.5 Write an assembly language program to perform the nested function call

Solution:
In the program given below, I’ve called the check$_{p}rimesfunctioninthemainfunction.inthecheck_{p}rimesfunctionI'vecal$

```
    AREA Task1, CODE, READONLY
    ENTRY
    EXPORT main
; define the array of integers
arr        DCD 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 17, 19, 23, 29
; define the prime function
; input: integer in r0
; output: 0 if not prime, non-zero if prime
prime
    MOV r1, #2            ; initialize divisor to 2
    CMP r0, #1            ; check if input is 1
    BEQ not_prime         ; branch if input is 1
loop
    CMP r0, r1            ; compare input to divisor
    BEQ is_prime          ; branch if input equals divisor
    MOV r2, #0            ; clear remainder register
    MOV r3, r0            ; copy input to working register
    SDIV r3, r3, r1       ; divide input by divisor
    MUL r3, r3, r1        ; multiply quotient by divisor
    SUB r2, r0, r3        ; calculate remainder
    CMP r2, #0            ; compare remainder to 0
    BEQ not_prime         ; branch if remainder is 0
    ADD r1, r1, #1        ; increment divisor
    B loop                ; repeat loop with new divisor
is_prime
    MOV r0, #1           ; set return value to 1 (prime)
    BX lr                ; return to calling function
not_prime
    MOV r0, #0          ; set return value to 0 (not prime)
    BX lr               ; return to calling function

main
    MOV r4, #0            ; initialize array index to 0
    MOV r5, #10           ; set array size to 10
    LDR r7, =arr
    BL check_primes

check_primes
    CMP r4, r5            ; compare array index to size
    BEQ STOP       ; branch if array is complete
    LDR r0, [r7]                 ; reading array element
    ADD r7, r7, #4                 ; incrementing array position
    ADD r4, r4, #1        ; increment array index
    BL prime
    B check_primes        ; repeat loop with next integer

STOP B STOP
    END
```

5 Experimental Result

5.1 Write an assembly language program to identify prime numbers from a list of array by calling a function called prime.

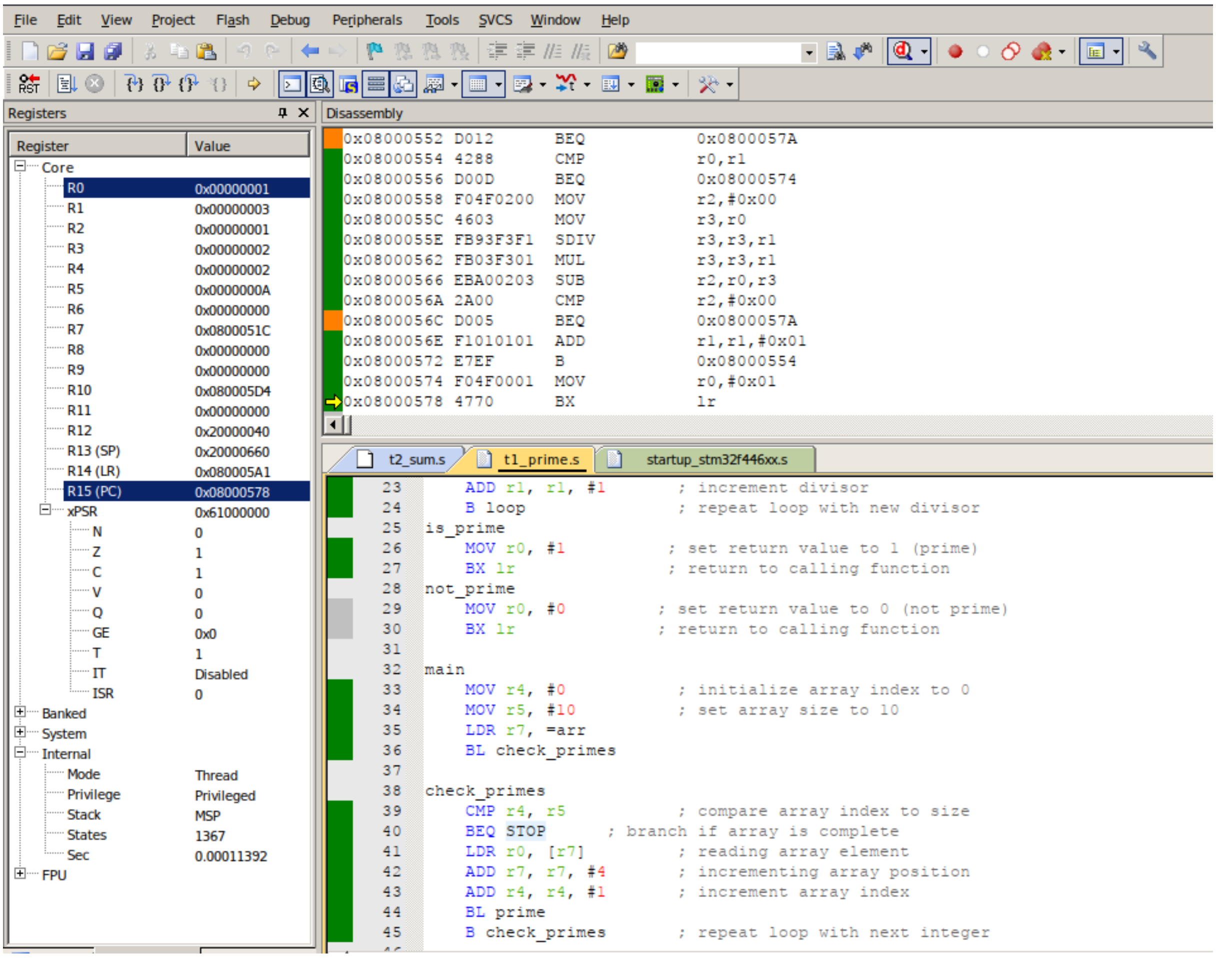

Figure 1: Identifying Prime Numbers

5.2 Write an assembly language program to perform the summation of two numbers by call a function sum(arg1, arg2) using call-by-value.

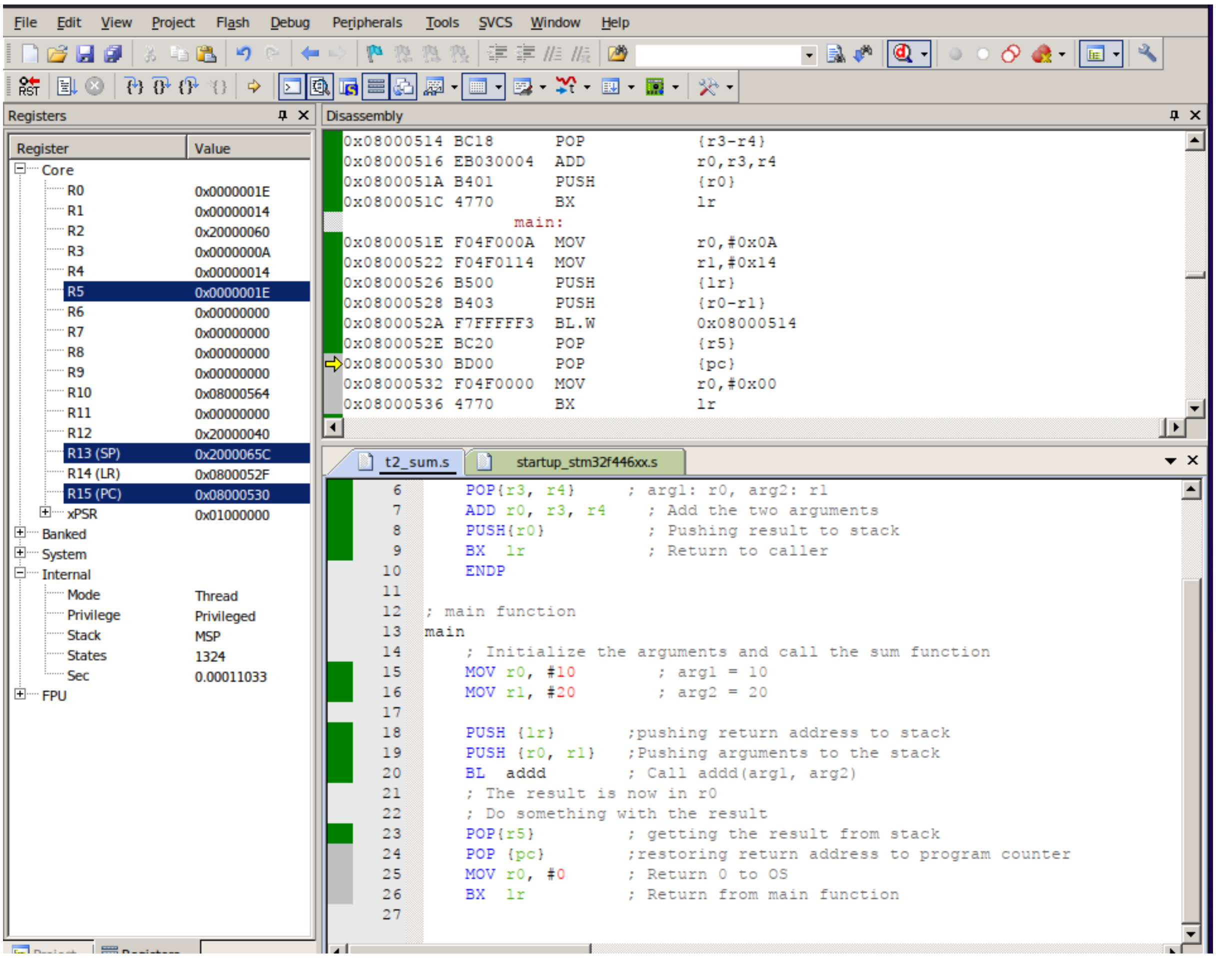

Figure 2: Adding two number by calling function(Call By Value)

5.3 Write an assembly language program to perform the summation of two numbers by call a function sum(arg1, arg2) using call-by-reference

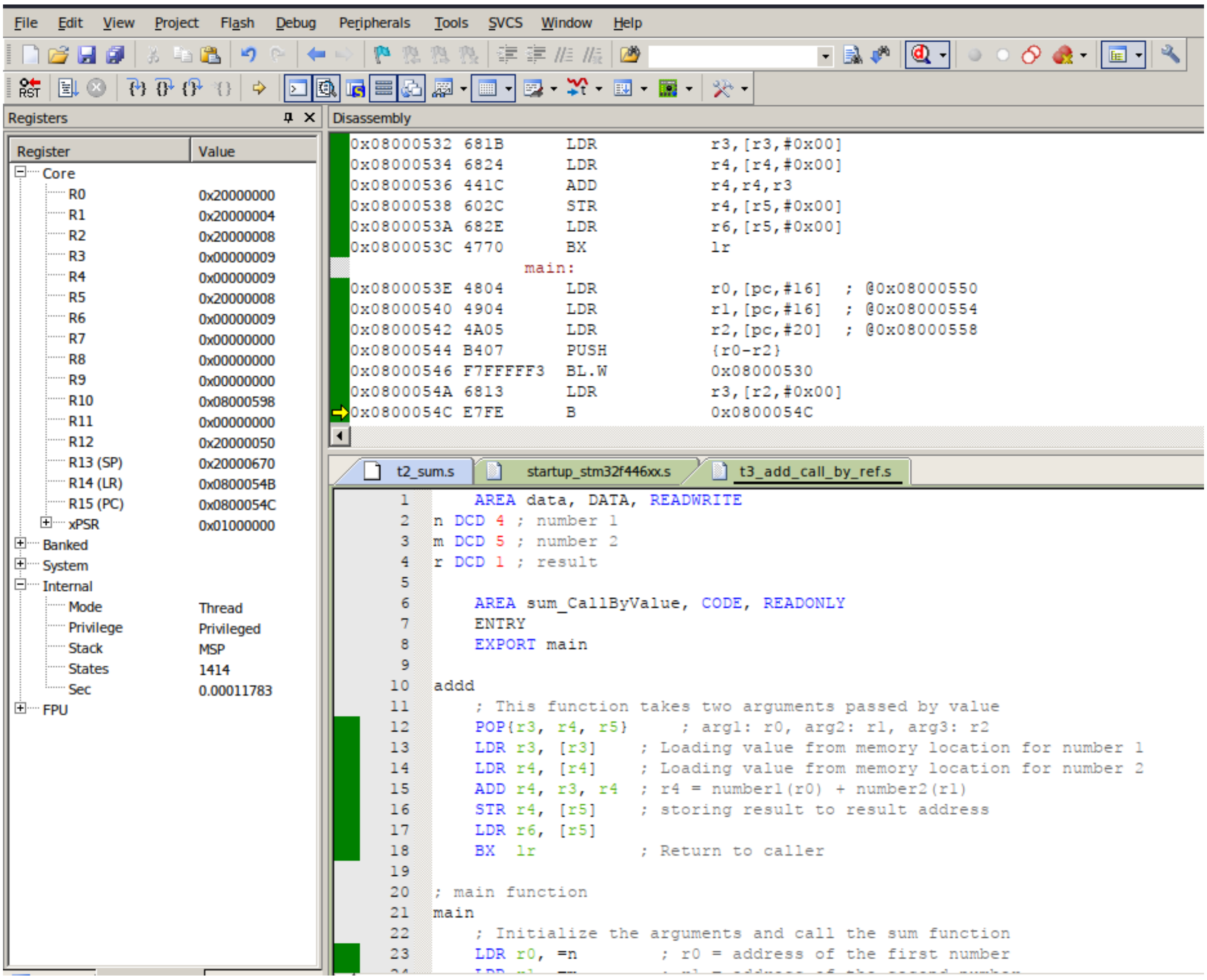

Figure 3: Adding two number by calling function(Call By Reference)

5.4 Write an assembly language program to find and return the minimum element in an array, where the array and its size are given as parameters by using recursive function

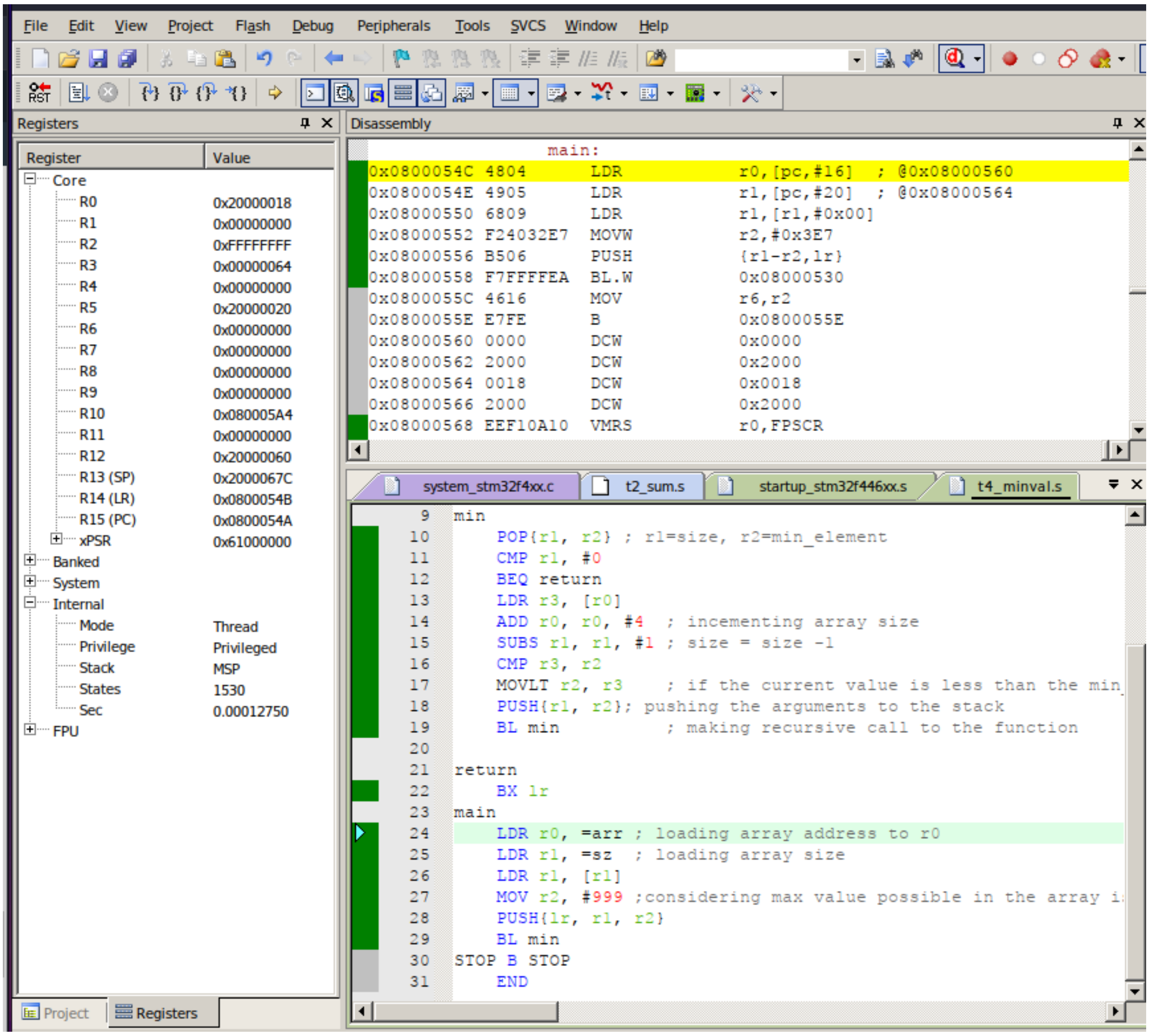

Figure 4: Finding minimum element of an array using recursive function

5.5 Write an assembly language program to perform the nested function call

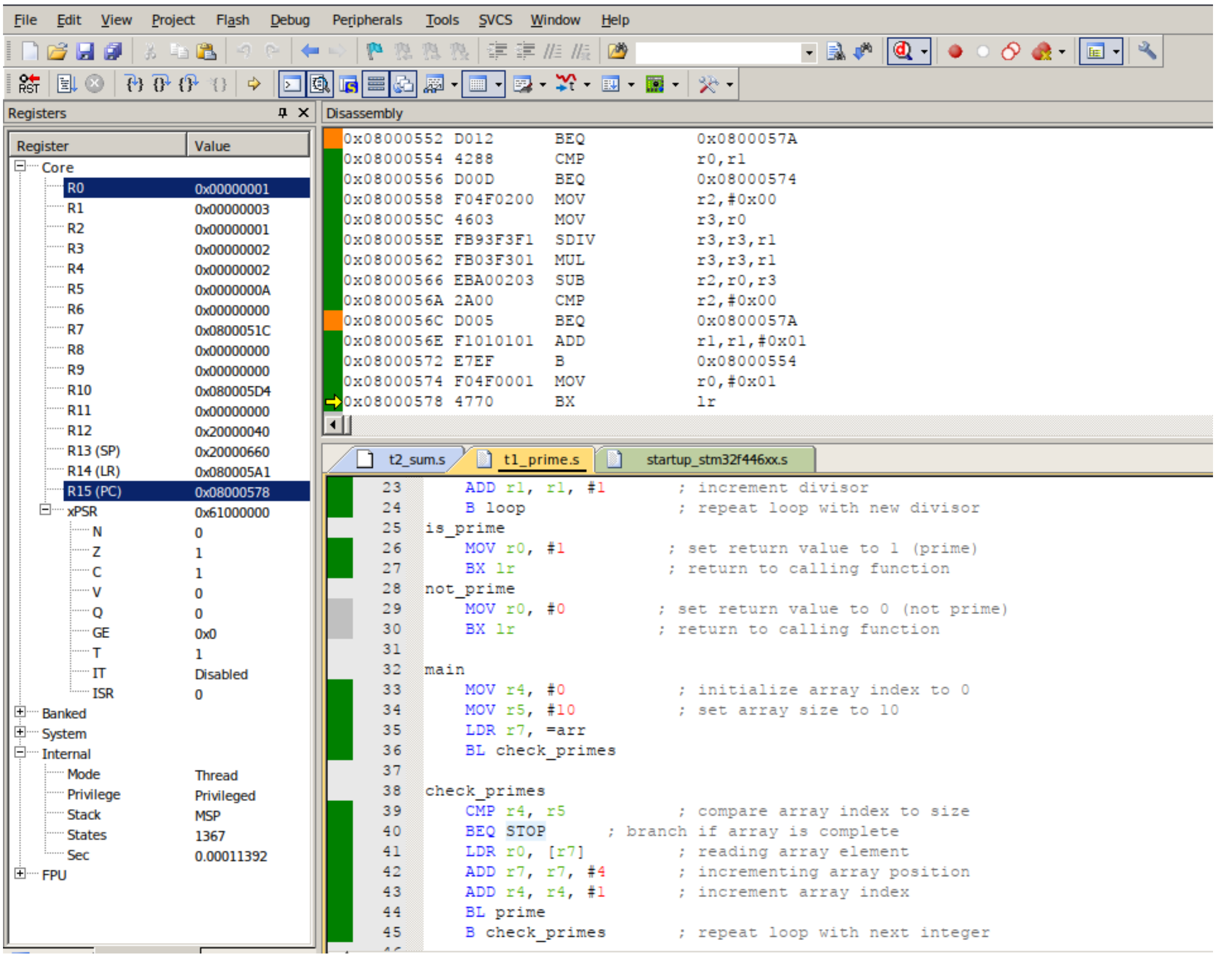

Figure 5: Nested Function Call Implementation

6 Experiences

During the lab, we've to face various challenges to setup Keil uVision IDE including solving debugger issues. After all, we could complete the experiment successfully and got output as expected. Before that lab, we would know about registers and their operations theoretically. But the lab experiment has given us the opportunity to observe running mathematical operations using registers directly.

7 Conclusion

The lab helped us to understand and familiarize with register-based assembly programming for the Cortex M4 processor for arithmetic operations. The results showed that this approach provides faster and more efficient execution compared to memory-based programming, making it a useful tool for embedded systems development.

In conclusion, the lab provides a solid foundation for further exploration of the benefits and limitations of register-based assembly programming for embedded systems development.