SANTA CLARA UNIVERSITY
Electrical and Computer Engineering Department

ELEN 120 – Embedded Computing Systems

Lab 4 – Stacks and Subroutines

Sal Martinez and Dylan Thornburg

Assignment/Learning Objectives: In this assignment, you will explore structured assembly language programming including the use of subroutine calls and passing data on the system stack. We will use the STM32L476G-DISCO Discovery kit and thus use the same programming framework as in Lab 3.

Lab Procedure:

The initialization files that we use in the Discovery kit framework allocate 4K bytes for the system stack and initialize r13 as the stack pointer. The stack is intended to grow "down" towards lower addresses. The initial stack pointer points to the word following the stack and thus must be decremented (by 4) prior to placing a value on the stack.

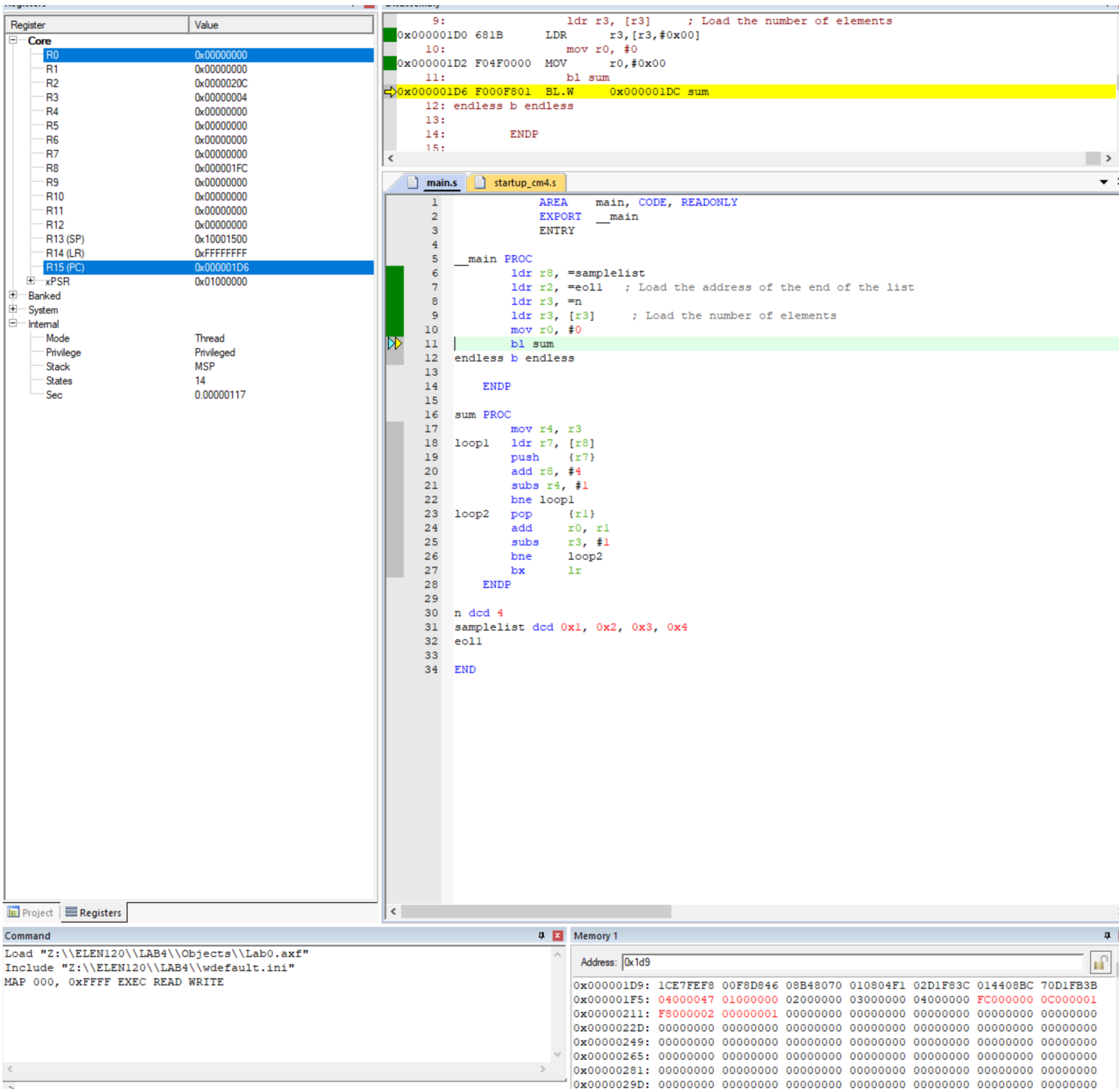
All of your subroutines in this lab should conform to the ARM subroutine register-passing standard which is at the end of the lab.

Problem 1

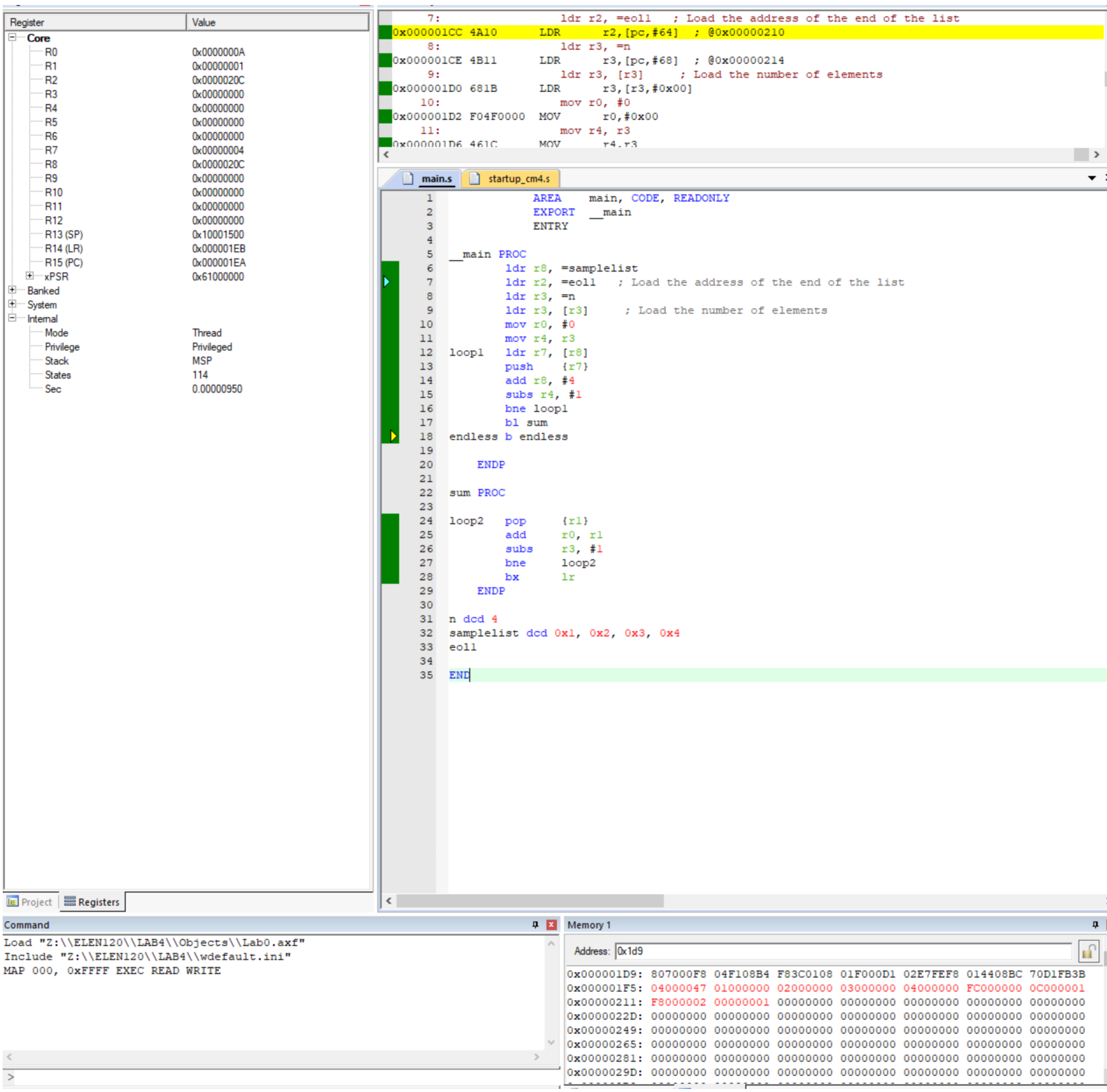
In this problem, you must write a subroutine called sum. In order to call sum, the calling routine should push *n* 32-bit signed integers onto the system stack then place *n* into r0. The subroutine sum will return the sum of these integers in r0. Remember that the ARM protocol requires that the stack pointer is unchanged when returning from a subroutine. That means that the called routine must restore/retain the stack pointer and also that the calling routine must remove the integers from the stack after calling sum.

Create a calling program in main to demonstrate that your subroutine works properly and demo to the TA. Turn in screenshots showing proper operation (before and after) and your source code.

Before:



After:



Code:

```
        AREA    main, CODE, READONLY
        EXPORT  __main
        ENTRY

__main PROC
```

```
            ldr r8, =samplelist
            ldr r2, =eol1   ; Load the address of the end of the list
            ldr r3, =n
            ldr r3, [r3]    ; Load the number of elements
            mov r0, #0
            mov r4, r3
loop1   ldr r7, [r8]
            push    {r7}
            add r8, #4
            subs r4, #1
            bne     loop1
            bl sum
endless b endless

        ENDP

sum PROC

loop2   pop     {r1}
            add     r0, r1
            subs    r3, #1
            bne             loop2
            bx              lr
        ENDP

n dcd 4
samplelist dcd 0x1, 0x2, 0x3, 0x4
eol1

END
```

Problem 2

In this problem you are going to write a subroutine for calculating the Fibonacci sequence. There are many ways to do this and the recursive algorithm is generally the worst one no matter what measure you use – but it is a good example for learning, so we are going to use it here.

You are going to write a subroutine, compliant with the ARM subroutine rules, called fib. fib will receive one unsigned integer parameter (p) in r0. It must return the p^{th} Fibonacci number in r0. It does so by doing the following:

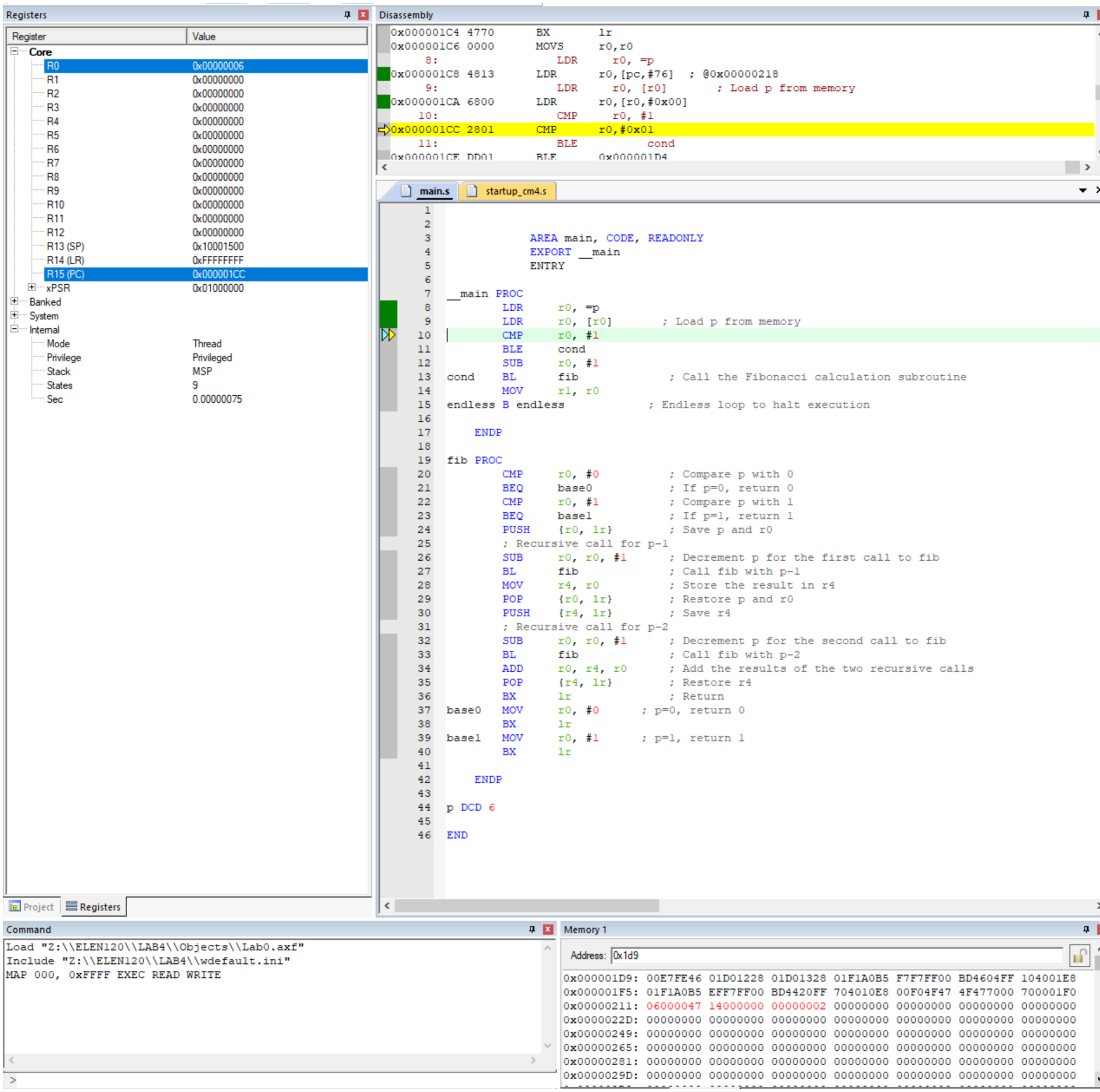
- If p=0, return 0
- If p=1 return 1
  - If p>1 then return the sum of fib(p-1) and fib(p-2) **by calling fib twice and adding the results**

Your code must work as specified above or it is not acceptable.

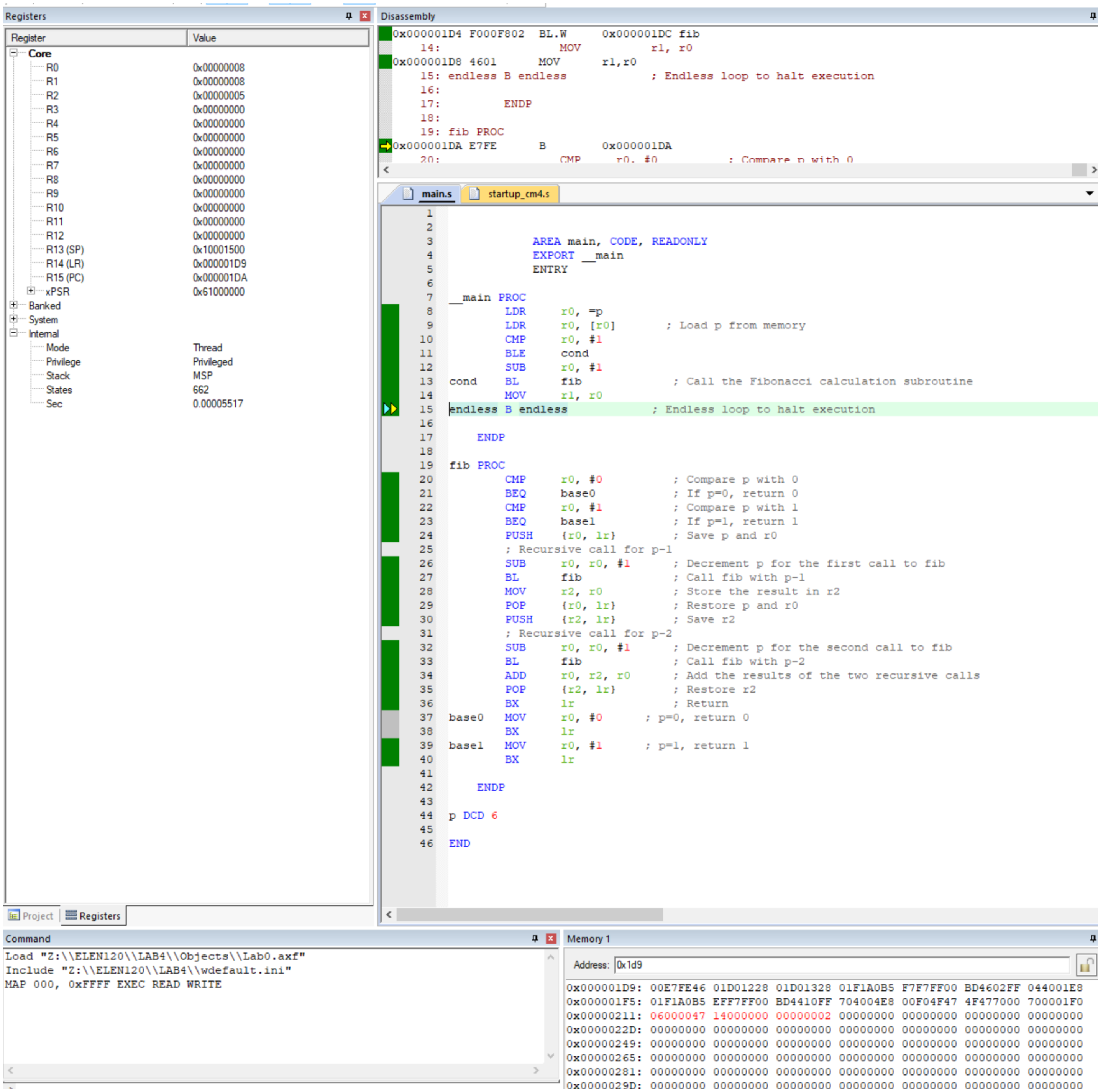
Obviously, writing a routine that calls itself requires a great deal of precision. You must follow all of the subroutine rules and you must use the stack to store things that may be destroyed in a subroutine call. Pay special attention to the link register.

Create a calling program in main to demonstrate that your subroutine works properly and demo to the TA. Turn in screenshots showing proper operation (before and after) and your source code.

BEFORE:



AFTER:



CODE:

```
                AREA main, CODE, READONLY
                EXPORT __main
                ENTRY

__main PROC
```

```
                    LDR     r0, =p
                    LDR     r0, [r0]       ; Load p from memory
                    CMP    r0, #1
                    BLE              cond
                    SUB              r0, #1
cond    BL      fib            ; Call the Fibonacci calculation subroutine
                    MOV             r1, r0
endless B endless          ; Endless loop to halt execution

          ENDP

fib PROC
                    CMP     r0, #0         ; Compare p with 0
                    BEQ     base0          ; If p=0, return 0
                    CMP     r0, #1         ; Compare p with 1
                    BEQ     base1          ; If p=1, return 1
                    PUSH    {r0, lr}                    ; Save p and r0
                    ; Recursive call for p-1
                    SUB     r0, r0, #1      ; Decrement p for the first call to fib
                    BL      fib            ; Call fib with p-1
                    MOV     r2, r0         ; Store the result in r2
                    POP     {r0, lr}           ; Restore p and r0
                    PUSH    {r2, lr}                    ; Save r2
                    ; Recursive call for p-2
                    SUB     r0, r0, #1      ; Decrement p for the second call to fib
                    BL      fib            ; Call fib with p-2
                    ADD     r0, r2, r0      ; Add the results of the two recursive calls
                    POP     {r2, lr}           ; Restore r2
                    BX      lr             ; Return
base0   MOV     r0, #0      ; p=0, return 0
      BX      lr
base1   MOV     r0, #1      ; p=1, return 1
      BX      lr

          ENDP

p DCD 6

END
```

Problem 3

In this lab you are going to read the joystick button on the Discovery board and record the button presses. You will use subroutines to implement much of the code. Your individual subroutines may not need to do much sophisticated register or stack management in this example, but this is a good tool for organizing your program. My preference is to place the subroutines for this project into a separate source code file jstick.s and to include headers in a header file jstick.h. Hopefully, I have already showed you how to do this in class.

The Joystick is connected to 5 pins of port A as shown in the board schematic.

There are 5 physical switches in the joystick and when pressed left, right, up, down, or center, they connect one of the switch pins to the common pin. The common pin is connected to 3 Volts through a small resistor. The 5 remaining switch pins are connected to bits 0-3 and 5 of port A on the STM32L476VGT6. What happened to bit 4? Did they leave it out just to annoy you? Maybe. It is actually used for something else. There is a good reason that we will see in Lab 5. Someone ask me to explain it this week.

Each of these switch inputs also has a capacitor. Along with the resistor on the common, this acts as a low-pass filter to debounce the switch. The inputs on the STM32L476VGT6 include a Schmitt Trigger which applies hysteresis to further reduce switch noise. I have found this circuit to be reliably debounced.

Since pressing the switch connects it to the 3V common line, that will pull the input high. When it is not pressed, you must pull it low by configuring each of these Port A inputs to include a pull-down resistor.

Step 1:

Step 1 is to write and test a subroutine that configures Port A in the manner that you want it configured for this program. This requires:

1. Enable clock on port A (Using RCC_AHBENR)
2. Configure port A pins 0,1,2,3,5 as inputs with the mode register (GPIOA_MODER)
3. Configure these pins as pull-down inputs. (GPOIA_PUPDR)

Remember that the best way to access these registers is to use the mnemonics in the stm32l476xx_constants.s file and to combine the base address of each register block with an offset. The descriptions of these 3 registers is below.

You should write a subroutine called porta_init that requires no parameters. It should perform all of the configuration of Port A. Add code to main.s to call this routine and test it in the simulator.

Step 2:

Now write a subroutine called read_jstick that reads Port A using the GPIOA_IDR register then clears all the bits other than the 5 bits we are interested in. Those 5 bits are returned in r0.

Add code to main.s to test this subroutine.

Step 3:

Write a subroutine called decode_stick that receives the value returned by read_stick as a parameter in r0. It returns a single character (in r0) representing which switch has been activated.

	Character
Center	c
Left	l
Right	r
Up	u
Down	d

If more than one switch has been pressed, your routine can return a letter that matches any one that has been pressed.

Add code to main.s to test this subroutine.

Step 4:

You are now going to use these 3 subroutines to create a final program.
Your program should include a data buffer that can hold 5 bytes of data. This is where you will record your button presses.

You should first configure port A. Next you should have a loop that:

1. Reads Port A
2. Determines if any of the buttons have changed since the last time you checked. 3. If a new button has been pressed, decode the button press and store the corresponding character in the next empty slot in your data buffer. Remember to pay attention to the data type you are using.
4. When the data buffer is full, exit this loop and run an endless loop at the end of the program.

You may need to do additional initialization or keep track of additional data. Remember that subroutine calls may alter certain register values and will preserve others.

Debug this program and demo. If you set a breakpoint on the endless loop and run the program in the debugger, it should stop after 5 button presses. You can then see the 5 characters representing those button presses in the memory window of the debugger (if you set it up right).

Capture a before and after screenshot showing that your program works and turn in the source code. Demo for the TA.

ARM Procedure Call Standard

Register	Usage	Subroutine Preserved	Notes
r0	Argument 1 and return value	No	If return has 64 bits, then r0:r1 hold it. If argument 1 has 64 bits, r0:r1 hold it.
r1	Argument 2	No	
r2	Argument 3	No	If the return has 128 bits, r0-r3 hold it.
r3	Argument 4	No	If more than 4 arguments, use the stack
r4	General-purpose V1	Yes	Variable register 1 holds a local variable.
r5	General-purpose V2	Yes	Variable register 2 holds a local variable.
r6	General-purpose V3	Yes	Variable register 3 holds a local variable.
r7	General-purpose V4	Yes	Variable register 4 holds a local variable.
r8	General-purpose V5	YES	Variable register 5 holds a local variable.
r9	Platform specific/V6	No	Usage is platform-dependent.
r10	General-purpose V7	Yes	Variable register 7 holds a local variable.
r11	General-purpose V8	Yes	Variable register 8 holds a local variable.
r12 (IP)	Intra-procedure-call register	No	It holds intermediate values between a procedure and the sub-procedure it calls.
r13 (SP)	Stack pointer	Yes	SP has to be the same after a subroutine has completed.
r14 (LR)	Link register	No	LR does not have to contain the same value after a subroutine has completed.
r15 (PC)	Program counter	N/A	Do not directly change PC

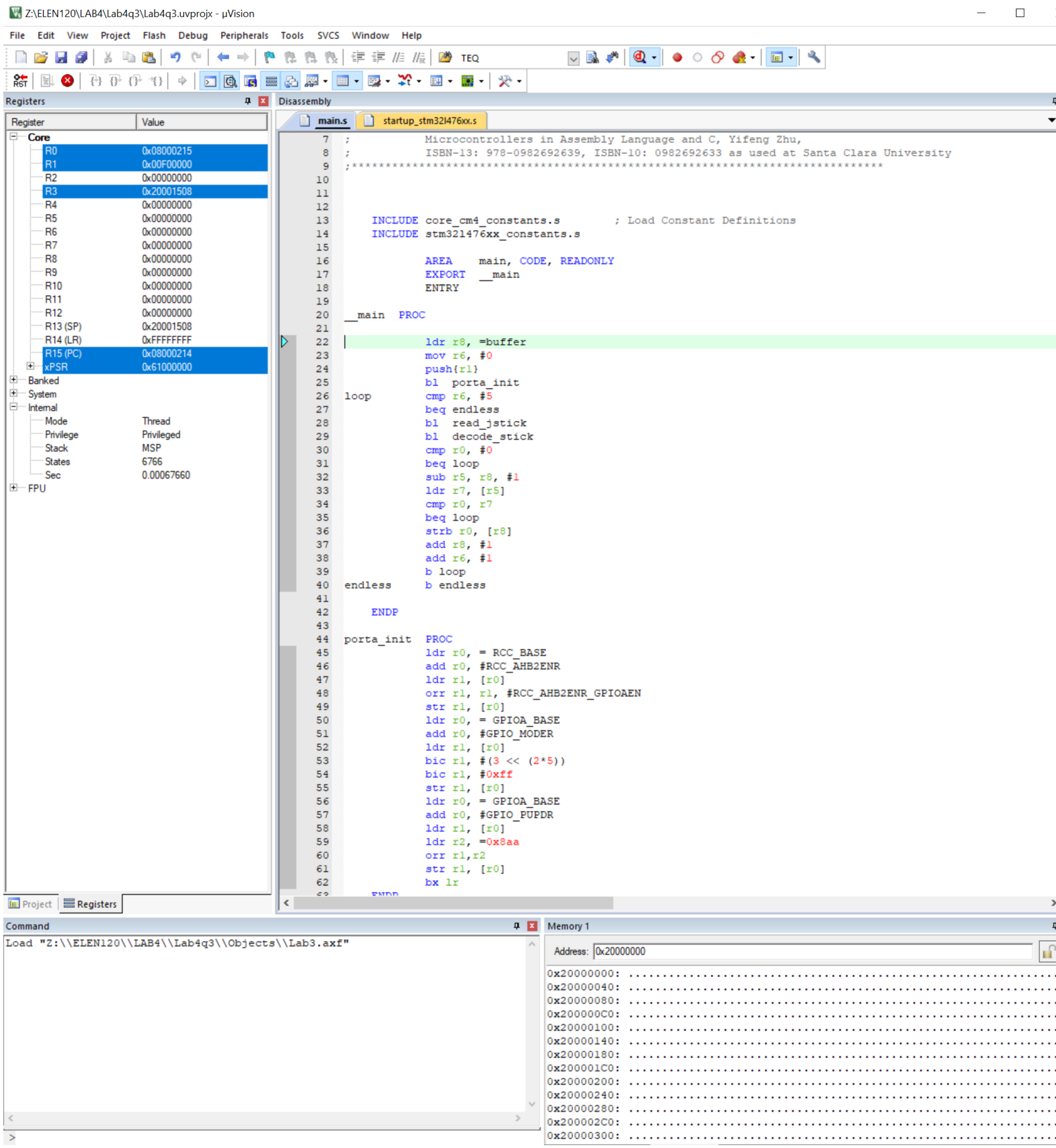
CODE:

```
;******************** (C) Andrew Wolfe ******************************************
; @file    main_hw_proto.s
; @author  Andrew Wolfe
; @date    August 18, 2019
; @note
;         This code is for the book "Embedded Systems with ARM Cortex-M
;         Microcontrollers in Assembly Language and C, Yifeng Zhu,
;         ISBN-13: 978-0982692639, ISBN-10: 0982692633 as used at Santa Clara University
```

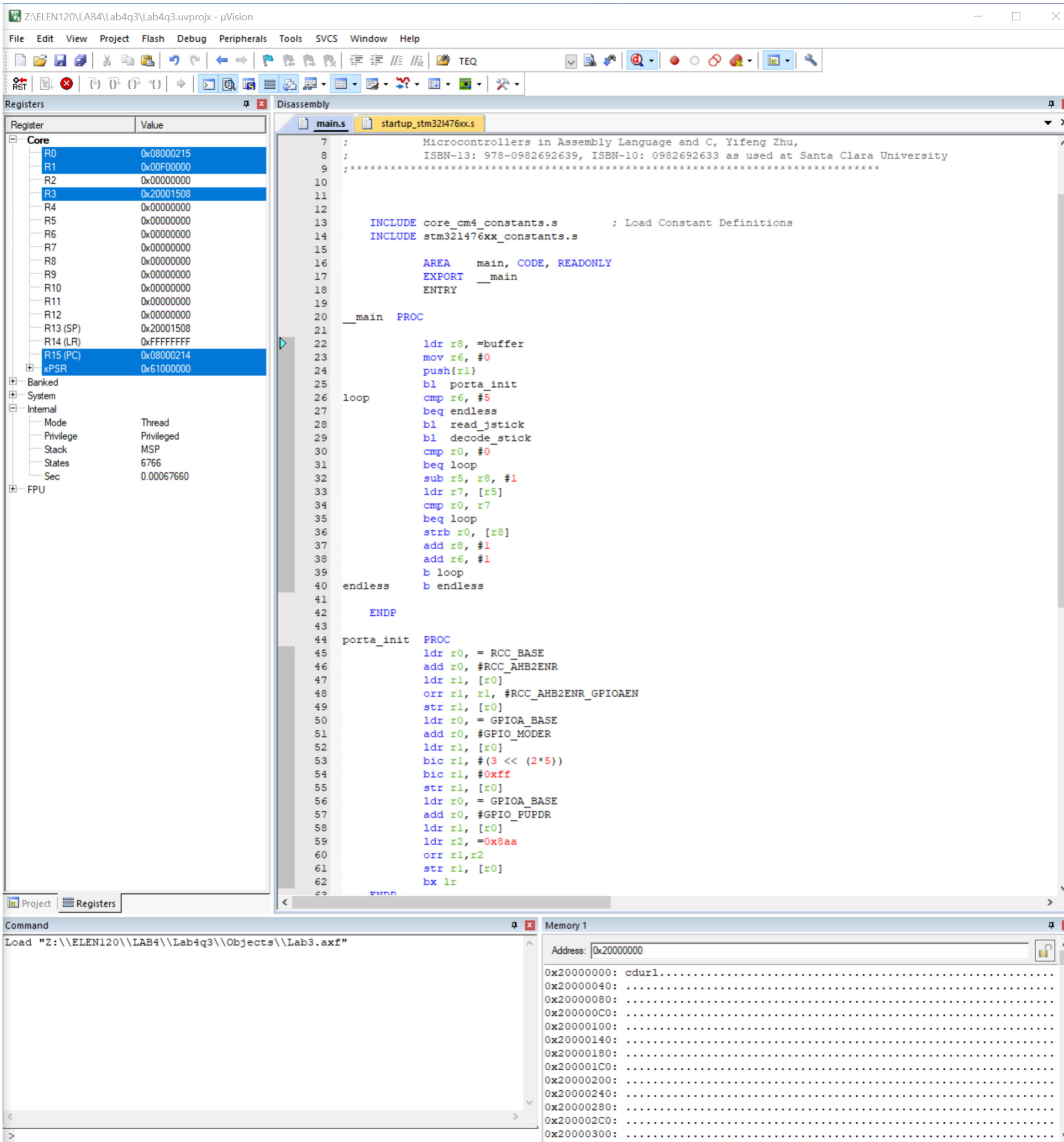
```
;***************************************************************************
**


        INCLUDE core_cm4_constants.s            ; Load Constant Definitions
        INCLUDE stm32l476xx_constants.s

                AREA    main, CODE, READONLY
                EXPORT      __main
                ENTRY

__mainPROC

                ldr r8, =buffer
                mov r6, #0
                push{r1}
                bl      porta_init
loop            cmp r6, #5
                beq endless
                bl      read_jstick
                bl      decode_stick
                cmp r0, #0
                beq loop
                sub r5, r8, #1
                ldr r7, [r5]
                cmp r0, r7
                beq loop
                strb r0, [r8]
                add r8, #1
                add r6, #1
                b loop
endless         b endless

        ENDP

porta_init      PROC
                ldr r0, = RCC_BASE
                add r0, #RCC_AHB2ENR
                ldr r1, [r0]
                orr r1, r1, #RCC_AHB2ENR_GPIOAEN
                str r1, [r0]
                ldr r0, = GPIOA_BASE
                add r0, #GPIO_MODER
                ldr r1, [r0]
                bic r1, #(3 << (2*5))
                bic r1, #0xff
                str r1, [r0]
                ldr r0, = GPIOA_BASE
                add r0, #GPIO_PUPDR
```

```
                ldr r1, [r0]
                ldr r2, =0x8aa
                orr r1,r2
                str r1, [r0]
                bx lr
        ENDP
read_jstick     PROC
                ldr r0, =GPIOA_BASE
                ldrb r1, [r0, #GPIO_IDR]
                and r0, r1, #0x2f
                bx lr
        ENDP
decode_stick    PROC

                push {lr}
                cmp r0, #0
                ldreq r2, =0x0
                cmp r0, #1
                ldreq r2, =0x63 ; c
                cmp r0, #2
                ldreq r2, =0x6C ; l
                cmp r0, #4
                ldreq r2, =0x72 ; r
                cmp r0, #8
                ldreq r2, =0x75 ; u
                cmp r0, #0x20
                ldreq r2, =0x64 ; d
                mov r0, r2
                pop {lr}
                bx lr

        ENDP

            ENDP
                ALIGN
                AREA    myData, DATA, READWRITE
                ALIGN

buffer  DCB     0,0,0,0,0

        END
```

BEFORE:


Z:\ELEN120\LAB4\Lab4q3\Lab4q3.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Registers
Register Value
Core
R0 0x08000215
R1 0x00F00000
R2 0x00000000
R3 0x20001508
R4 0x00000000
R5 0x00000000
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x20001508
R14 (LR) 0xFFFFFFFF
R15 (PC) 0x08000214
xPSR 0x61000000
Banked
System
Internal
Mode Thread
Privilege Privileged
Stack MSP
States 6766
Sec 0.00067660
FPU
Project Registers
Disassembly
main.s startup_stm32l476xx.s
; Microcontrollers in Assembly Language and C, Yifeng Zhu,
; ISBN-13: 978-0982692639, ISBN-10: 0982692633 as used at Santa Clara University
INCLUDE core_cm4_constants.s ; Load Constant Definitions
INCLUDE stm32l476xx_constants.s
AREA main, CODE, READONLY
EXPORT __main
ENTRY
__main PROC
ldr r8, =buffer
mov r6, #0
push{r1}
bl porta_init
loop cmp r6, #5
beq endless
bl read_jstick
bl decode_stick
cmp r0, #0
beq loop
sub r5, r8, #1
ldr r7, [r5]
cmp r0, r7
beq loop
strb r0, [r8]
add r8, #1
add r6, #1
b loop
endless b endless
ENDP
porta_init PROC
ldr r0, = RCC_BASE
add r0, #RCC_AHB2ENR
ldr r1, [r0]
orr r1, r1, #RCC_AHB2ENR_GPIOAEN
str r1, [r0]
ldr r0, = GPIOA_BASE
add r0, #GPIO_MODER
ldr r1, [r0]
bic r1, #(3 << (2*5))
bic r1, #0xff
str r1, [r0]
ldr r0, = GPIOA_BASE
add r0, #GPIO_PUPDR
ldr r1, [r0]
ldr r2, =0x8aa
orr r1,r2
str r1, [r0]
bx lr
Command
Load "Z:\\ELEN120\\LAB4\\Lab4q3\\Objects\\Lab3.axf"
Memory 1
Address: 0x20000000
0x20000000:
0x20000040:
0x20000080:
0x200000C0:
0x20000100:
0x20000140:
0x20000180:
0x200001C0:
0x20000200:
0x20000240:
0x20000280:
0x200002C0:
0x20000300:

AFTER:


Z:\ELEN120\LAB4\Lab4q3\Lab4q3.uvprojx - µVision
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
TEQ
Registers
Register Value
Core
R0 0x08000215
R1 0x00F00000
R2 0x00000000
R3 0x20001508
R4 0x00000000
R5 0x00000000
R6 0x00000000
R7 0x00000000
R8 0x00000000
R9 0x00000000
R10 0x00000000
R11 0x00000000
R12 0x00000000
R13 (SP) 0x20001508
R14 (LR) 0xFFFFFFFF
R15 (PC) 0x08000214
xPSR 0x61000000
Banked
System
Internal
Mode Thread
Privilege Privileged
Stack MSP
States 6766
Sec 0.00067660
FPU
Project Registers
Disassembly
main.s startup_stm32l476xx.s
; Microcontrollers in Assembly Language and C, Yifeng Zhu,
; ISBN-13: 978-0982692639, ISBN-10: 0982692633 as used at Santa Clara University
;***
INCLUDE core_cm4_constants.s ; Load Constant Definitions
INCLUDE stm32l476xx_constants.s
AREA main, CODE, READONLY
EXPORT __main
ENTRY
__main PROC
ldr r8, =buffer
mov r6, #0
push{r1}
bl porta_init
loop cmp r6, #5
beq endless
bl read_jstick
bl decode_stick
cmp r0, #0
beq loop
sub r5, r8, #1
ldr r7, [r5]
cmp r0, r7
beq loop
strb r0, [r8]
add r8, #1
add r6, #1
b loop
endless b endless
ENDP
porta_init PROC
ldr r0, = RCC_BASE
add r0, #RCC_AHB2ENR
ldr r1, [r0]
orr r1, r1, #RCC_AHB2ENR_GPIOAEN
str r1, [r0]
ldr r0, = GPIOA_BASE
add r0, #GPIO_MODER
ldr r1, [r0]
bic r1, #(3 << (2*5))
bic r1, #0xff
str r1, [r0]
ldr r0, = GPIOA_BASE
add r0, #GPIO_PUPDR
ldr r1, [r0]
ldr r2, =0x8aa
orr r1,r2
str r1, [r0]
bx lr
Command
Load "Z:\\ELEN120\\LAB4\\Lab4q3\\Objects\\Lab3.axf"
Memory 1
Address: 0x20000000
0x20000000: cdurl...
0x20000040: ....
0x20000080: ....
0x200000C0: ....
0x20000100: ....
0x20000140: ....
0x20000180: ....
0x200001C0: ....
0x20000200: ....
0x20000240: ....
0x20000280: ....
0x200002C0: ....
0x20000300: ....