EE 3220 – System-on-Chip Design Assignment 2

Question 1:

```
fn(int*, long*, float*):
```

1. `push {r11, lr}`
   Save register r11 and Link register (LR) on the stack.
2. `mov r11, sp`
   Load stack pointer (SP) in r11.
3. `sub sp, sp, #40`
   Subtract 40 from the stack pointer register and stores it back into stack pointer register.
4. `str r0, [r11, #-4]`
   Store **integer a** to memory pointed to by **r11 at offset -4**.
5. `str r1, [r11, #-8]`
   Store **long b** to memory pointed to by **r11 at offset -8**.
6. `str r2, [r11, #-12]`
   Store **float c** to memory pointed to by **r11 at offset -12**.
7. `mov r0, #15`
   Save **15** in r0 for **a1**.
8. `str r0, [r11, #-16]`
   Store **a1** to memory pointed to by r11 at offset -16.
9. `mvn r1, #13`
   Save not(13) which is **-14** in r1 for **a2**.
10. `str r1, [sp, #20]`
    Store **a2** to memory pointed to by SP at offset 20.
11. `ldr r2, [r11, #-16]`
    Load **a1** from stack into r2.
12. `ldr r3, [sp, #20]`
    Load **a2** from stack into r3.
13. `mul r12, r2, r3`
    Multiply **a1** and **a2**, store result in r12.
14. `ldr r2, [r11, #-4]`
    Load **r11 at offset -4** from stack into r2.
15. `str r12, [r2]`
    Store **a1*a2 result** to memory pointed to by **r11 at offset -4** which is **a**.
16. `str r0, [sp, #16]`
    Store **15** to memory pointed to by SP at offset 16 for **b1**.
17. `str r1, [sp, #12]`
    Store **-14** to memory pointed to by SP at offset 12 for **b2**.
18. `ldr r0, [sp, #16]`
    Load **b1** from stack into r0.
19. `ldr r1, [sp, #12]`
    Load **b2** from stack into r1.

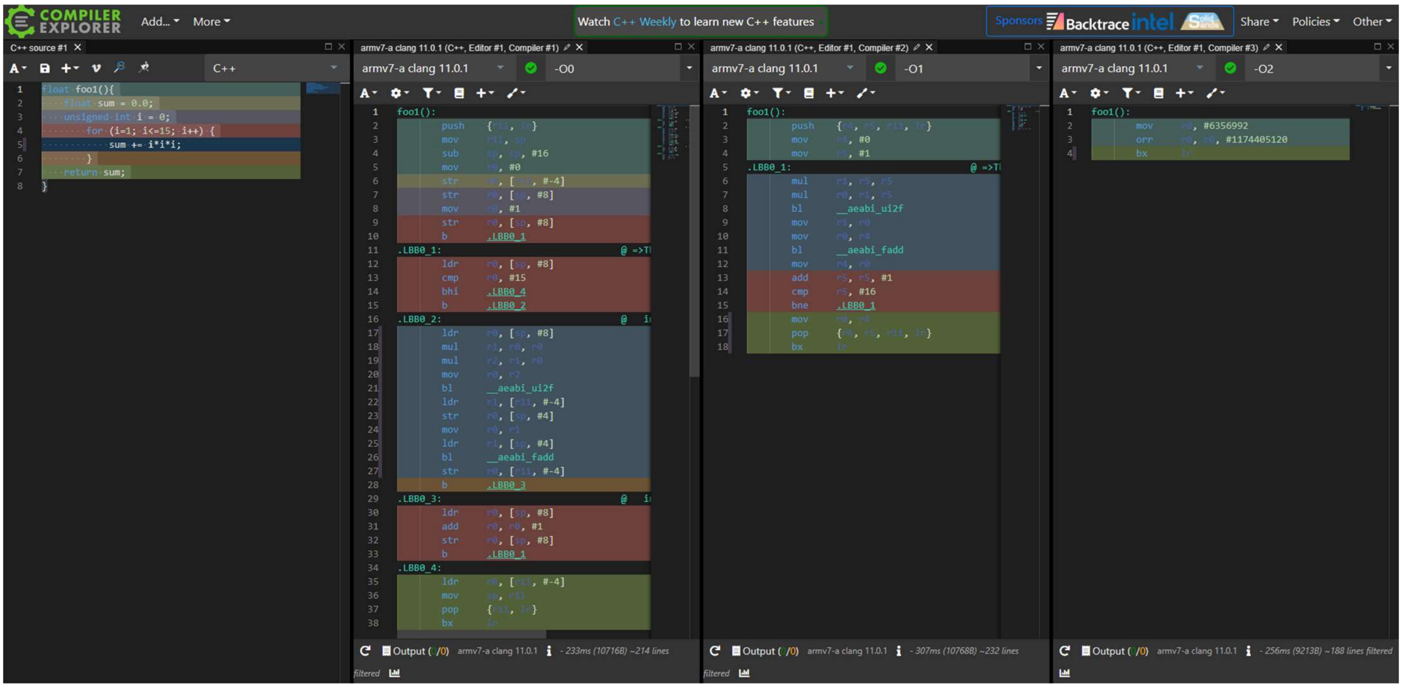
20. `mul r2, r0, r1`
    Multiply **b1** and **b2**, store result in r2.
21. `ldr r0, [r11, #-8]`
    Load **r11 at offset -8** from stack into r0.
22. `str r2, [r0]`
    Store **b1*b2 result** to memory pointed to by **r11 at offset -8** which is **b**.
23. `mov r0, #24117248`
    Save **24117248** in r0.
24. `orr r0, r0, #1073741824`
    Logical or for **24117248** and **1073741824** which is **15f**, store result in r0.
25. `str r0, [sp, #8]`
    Store **15f** to memory pointed to by SP at offset 8 for **c1**.
26. `mov r0, #23068672`
    Save **23068672** in r0.
27. `orr r0, r0, #-1073741824`
    Logical or for **23068672** and **-1073741824** which is **-14f**, store result in r0.
28. `str r0, [sp, #4]`
    Store **-14f** to memory pointed to by SP at offset 4 for **c2**.
29. `ldr r0, [sp, #8]`
    Load **c1** from stack into r0.
30. `ldr r1, [sp, #4]`
    Load **c2** from stack into r1.
31. `bl __aeabi_fmul`
    Saves PC+4 in LR then Branch with link to **__aeabi_fmul**, which calculate float multiplication and store result at r0.
32. `ldr r1, [r11, #-12]`
    Load **r11 at offset -12** from stack into r1.
33. `str r0, [r1]`
    Store **c1*c2 result** to memory pointed to by **r11 at offset -12**.
34. `mov sp, r11`
    Load r11 in stack pointer.
35. `pop {r11, lr}`
    Restore register r11 and LR to original values by popping them off from the stack.
36. `bx lr`
    Branch and exchange (instructions) to address LR which is a return address from a function.

`main:`

1. `push {r11, lr}`
   Save register r11 and Link register (LR) on the stack.
2. `mov r11, sp`
   Load stack pointer (SP) in r11.
3. `sub sp, sp, #24`
   Subtract 24 from the stack pointer register and stores it back into stack pointer register.
4. `mov r0, #0`
   Save **0** in r0.
5. `str r0, [r11, #-4]`
   Store **0** to memory pointed to by **r11 at offset -4**.
6. `ldr r1, [r11, #-8]`
   Load **r11 at offset -8** from stack into r1, for **a**.
7. `ldr r2, [sp, #12]`
   Load **SP at offset 12** from stack into r2, for **b**.
8. `ldr r3, [sp, #8]`
   Load **SP at offset 8** from stack into r3, for **c**.

9. `str r0, [sp, #4]`
   Store **0** to memory pointed to by **SP at offset 4**.
10. `mov r0, r1`
    Save **a** in r0.
11. `mov r1, r2`
    Save **b** in r1.
12. `mov r2, r3`
    Save **c** in r2.
13. `bl fn(int*, long*, float*)`
    Saves PC+4 in LR then Brach with link to **fn(int*, long*, float*)**.
14. `ldr r0, [sp, #4]`
    Load **0** from stack into r0.
15. `mov sp, r11`
    Load r11 in stack pointer.
16. `pop {r11, lr}`
    Restore register r11 and LR to original values by popping them off from the stack.
17. `bx lr`
    Branch and exchange (instructions) to address LR which is a return address from a function.

Question 2:



In -O2 optimization, the "mov   r0, #6356992" and "orr   r0, r0, #1174405120" instructions are used to computes the return values **14400f** directly.

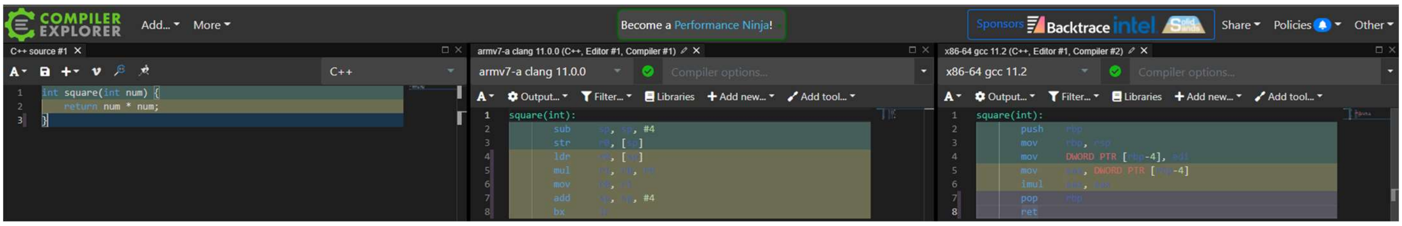
Under -O0 optimization, the compiler compiles all the codes from the source code without any optimization, which the complied code will be highly correlated with source code including dead code or unused code, which is easy to debug.

Under -O1 optimization, the compiler tries to reduce code size and execution time without performing any optimizations that affect a lot on compilation time, dead code and unused code will not be compiled, the code is still relatively easy to debug.

Under -O2 optimization, the compiler performs nearly all supported optimizations that do not involve a space-speed trade-off, which is this case the compiler computes the result of **sum of cube from 1 to 15** directly, and store the result **14400f** by move and logical OR function. Although the codes are fully optimized by the compiler, it is hard to debug from the code.

With higher optimization, the compiler will try to compile the code for higher performance, but at the same time harder to debug and it can increase compilation time or the binary size.

Question 3:



1. Using armv7-a clang 11.0

`square(int):`

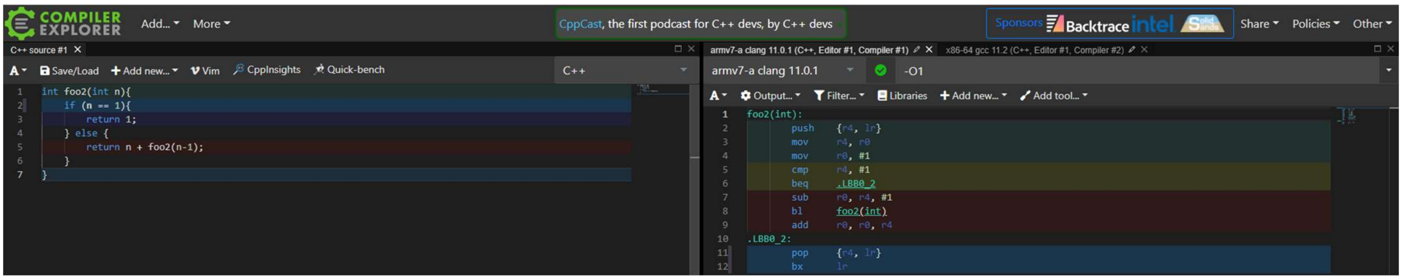
1. `sub     sp, sp, #4`
   Subtract 4 from the stack pointer register and stores it back into stack pointer register.
2. `str     r0, [sp]`
   Store **num** to SP.
3. `ldr     r0, [sp]`
   Load **num** from stack into r0.
4. `mul     r1, r0, r0`
   Multiply **num** with itself, store result in r1.
5. `mov     r0, r1`
   Load **result** in r0 for return.
6. `add     sp, sp, #4`
   Add 4 from the stack pointer register and stores it back into stack pointer register.
7. `bx      lr`
   Branch and exchange (instructions) to address LR which is a return address from a function.

2. Using x86-64 gcc 11.2

`square(int):`

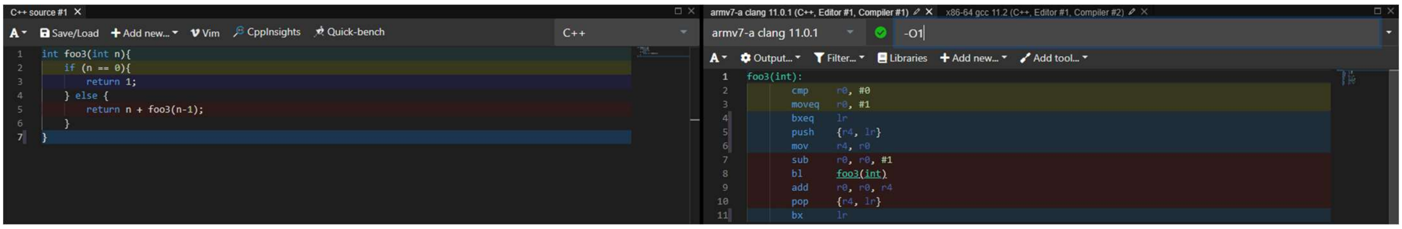
1. `push    rbp`
   Save register rbp on the stack.
2. `mov     rbp, rsp`
   Move rsp to rbp
3. `mov     DWORD PTR [rbp-4], edi`
   Move edi which stores **num** to memory pointed to by rbp at offset -4.
4. `mov     eax, DWORD PTR [rbp-4]`
   Move **num** to eax.
5. `imul    eax, eax`
   Multiply **num** with itself, store result in eax.
6. `pop     rbp`
   Restore register rbp to original values by popping it off from the stack.
7. `Ret`
   Return from Procedure

Question 4:



A. Which register(s) are saved before entering the recursive call? Explain why.

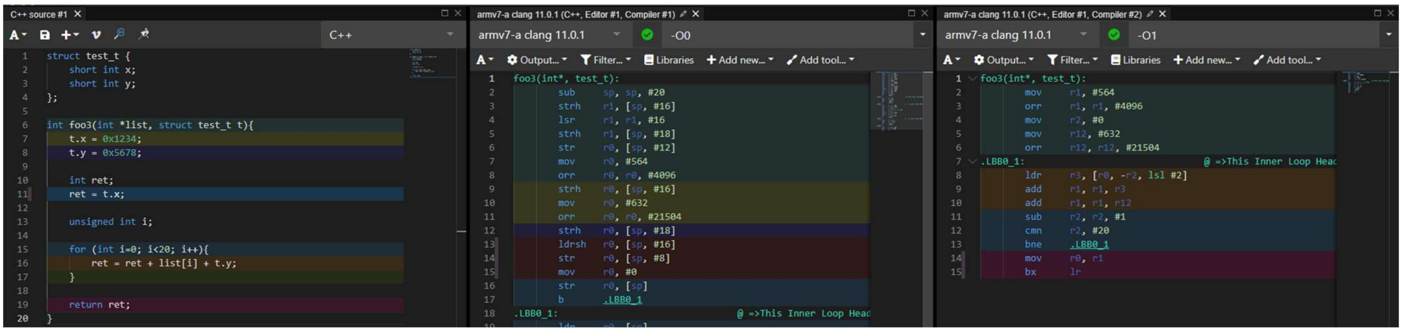
1. LR, which is the link register, is saved. LR is used to hold the address to which a function should return when it finishes executing. When the recursive is called, the returning address of the subroutine needs to be recorded for retuning after the subroutine's work is done.
2. R4, which is the callee-saved register for local variables. R4 is used to store the variable **n** in the C-program. When the recursive is called, the original value of R4 needs to be recorded for retuning after the subroutine's work is done.



B. Next modify line 2 from (n == 1) to (n == 0), Explain what has been changed?

The method of comparing numbers is changed. In stead of moving n from r0 to r4 first then check if **n** is equal to 1, then exit the subroutine else enter the recursive part; the modified version will compare r0 which is **n**, to 0 directly first, and if **n** is equal to 0, move 1 to **n** and exit the subroutine else enter the recursive part.

Question 5:



A. How many registers are used for the function arguments?

Two are used for the function argument.

B. If t.x = 0x1234 and t.y = 0x5678, how is the argument t stored in the parameter register(s)?

The argument t is stored in two registers, r1 and r12 under -O1 optimization, t.x is store to r1 with 564 then or with 4096 then store in r1 as 0x1234, t.y is store to r12 with 632 then or with 21504 then store in r1 as 0x5678. Under -O0 optimization, the instruction is similar but store the value to a stack pointer with offset later on instead of storing the value directly on the register. As t is a struct with two short int, t will be access as 0x0000123400005678.

C. If (i<20) is now updated to (i<5) in the for-loop, explain which assembly instruction is updated and why?

Instruction "**cmp**" is update from "**cmp r0, #19**" to "**cmp r0, #4**", which the number comparing with r0 is update from 19 to 4 so now the loop will end if r0 is greater than 4 (i<5) instead of 19 (i<20).

D. In O2, how many loop iterations do the assembly code perform for 1) (i<20), and 2) (i<100), and why?

1. For i<20, the number of iterations the assembly code perform is 1 as the loop is fully unrolled.
2. For i<100, the number of iterations the assembly code perform is 100 as r0 (i) need to add by 1 100 times (add r0, r0, #1) to break the loop with r0 greater than 99.

Question 6:

Reset handler in ARM processor is a function that is called whenever the processor resets. When the reset button is pressed, the Reset exception raised, and the reset handler will run. The reset handler is the first software to execute after a system reset. Usually, the reset handler is used for setting up configuration data for the C start-up code such as address range for stack and heap memories, which then branches into the C start-up code.

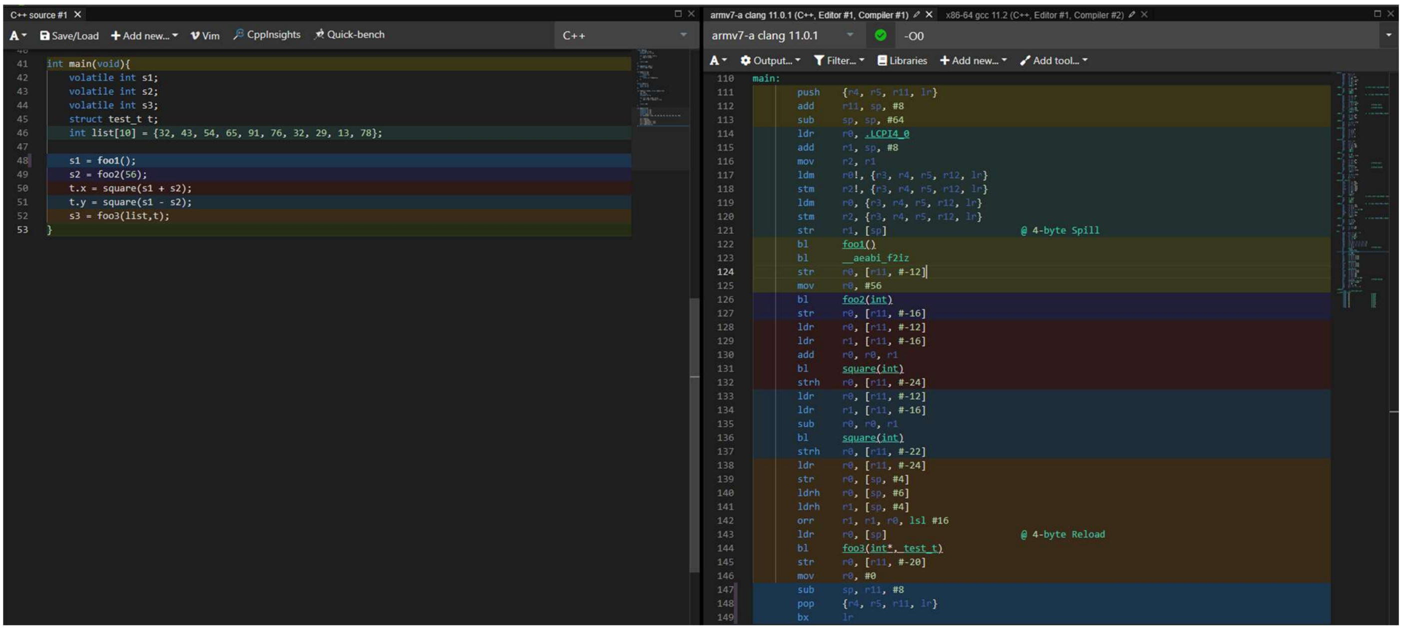
Example Code:

```
;Reset handler
Reset_Handler         PROC

EXPORT  Reset_Handler [WEAK]
IMPORT  SystemInit
IMPORT  __main

LDR     R0, =SystemInit
BLX     R0
LDR     R0, =__main
BX      R0
END
```

Question 7:



Local Variables	Offset from Stack Pointer (SP)
s1	sp + 60
s2	sp + 56
list	sp + 8
t.x	sp + 48
t.y	sp + 50

The integer array is represented as a label in the ARM assembly code, the location of the label is then loaded onto the register when being used.

```
.LCPI4_0:
        .long   .L__const.main.list
.L__const.main.list:
        .long   32                          @ 0x20
        .long   43                          @ 0x2b
        .long   54                          @ 0x36
        .long   65                          @ 0x41
        .long   91                          @ 0x5b
        .long   76                          @ 0x4c
        .long   32                          @ 0x20
        .long   29                          @ 0x1d
        .long   13                          @ 0xd
        .long   78                          @ 0x4e
```

Question 8:

```
int unknown(int num)
{
    if (num < 1)
        return 1;

    else
        return unknown(num - 1) * num;
}
```

Question 9:

1. Describe the procedure to complete this compilation.

The command "**arm-linux-gnueabihf-objdump -d helloworld32dyh**" is used to disassemble a binary into assembly code. Which "**arm-linux-gnueabihf**" is the toolchain for C, C++ and Assembly programming targeting for AArch32 architecture in a cross-platform environment, is this case, an amd64 environment. The "**objdump**" is a tool within the toolchain to display information from object files, with the flag "**-d**" means display the assembler mnemonics for the machine instructions from the input binary using default the architecture AArch32. And finally, "**helloworld32dyh**" is the binary to be disassemble.

2. Describe the meaning of the above terms: elf32-littlearm, .init, objdump, and push {lr}.

elf32-littlearm is the file format of the binary, which is an Executable and Linkable Format 32-bit Object Files (**elf32**) with Little-endian Arm architecture (**littlearm**).

.init is the section holds executable instructions that contribute to the process initialization code. When a program starts to run, the system arranges to execute the code in this section before calling the main program entry point.

objdump is a tool to display information from object files in order to debug with the object file, in which, the flag "-d" or "--disassemble" will disassemble the object file.

push {lr} mean push the Link Register into the Stack which LR is used to hold the address to which a function should return when it finishes executing. When a subroutine is called, the returning address of the subroutine needs to be recorded for retuning after the subroutine's work is done, thus, storing the address onto the stack when a subroutine is called.

Question 10:

`main:`

3. `push {lr}`
   Save Link register (LR) on the stack.
4. `ldr r4, .LCPI0_0`
   Load the **Output String** from stack into r4.
5. `mov r0, r4`
   Save the **Output String** to r0.
6. `mov r1, #0`
   Save **0** to r1, for **i**.
7. `mov r2, #0`
   Save **0** to r2, for **j**.

8. `mov r3, #1`
   Save **1** to r3, for **array[i][j]**.
9. `bl printf`
   Saves PC+4 in LR then Branch with link to printf.
10. `mov r0, r4`
    Save the **Output String** to r0.
11. `mov r1, #0`
    Save **0** to r1, for **i**.
12. `mov r2, #1`
    Save **1** to r2, for **j**.
13. `mov r3, #2`
    Save **2** to r3, for **array[i][j]**.
14. `bl printf`
    Saves PC+4 in LR then Branch with link to printf.
15. `mov r0, r4`
    Save the **Output String** to r0.
16. `mov r1, #1`
    Save **1** to r1, for **i**.
17. `mov r2, #0`
    Save **0** to r2, for **j**.
18. `mov r3, #3`
    Save **3** to r3, for **array[i][j]**.
19. `bl printf`
    Saves PC+4 in LR then Branch with link to printf.
20. `mov r0, r4`
    Save the **Output String** to r0.
21. `mov r1, #1`
    Save **1** to r1, for **i**.
22. `mov r2, #1`
    Save **1** to r2, for **j**.
23. `mov r3, #4`
    Save **4** to r3, for **array[i][j]**.
24. `bl printf`
    Saves PC+4 in LR then Branch with link to printf.
25. `mov r0, #0`
    Save **0** to r0, for **return**.
26. `pop {lr}`
    Restore LR to original values by popping them off from the stack.
27. `bx lr`
    Branch and exchange (instructions) to address LR which is a return address from a function.

```
.LCPI0_0:
    .long   .L.str
.L.str:
    .asciz  "array[%d] [%d] = %d \n"
```

Q1 source

https://godbolt.org/z/3o638zqGo

Decimal to Hexadecimal Converter (rapidtables.com)

Online Hex Converter - Bytes, Ints, Floats, Significance, Endians - SCADACore

What does bx lr do in ARM assembly language? - Stack Overflow

Q2 source

Compiler Getting Started Guide: Selecting optimization options (keil.com)

Optimize Options (Using the GNU Compiler Collection (GCC))

https://godbolt.org/z/6YMcrPab9

Online C++ Compiler - online editor (onlinegdb.com)

Q3 source

https://godbolt.org/z/Tac54dsYW

x86 and amd64 instruction reference (felixcloutier.com)

Q4 source

https://godbolt.org/z/hh19rK7TW

Q5 source

https://godbolt.org/z/Pj5s5En6c

Q6 source

What is reset handler in arm? – QuickAdviser (quick-adviser.com)

ARM Cortex M processor reset sequence (fastbitlab.com)

Q7 source

https://godbolt.org/z/5dq669E9s

Q9 source

Arm GNU Toolchain | GNU-A Downloads – Arm Developer

objdump(1) - Linux manual page (man7.org)

Chapter 9. elf32: Executable and Linkable Format 32-bit Object Files (tortall.net)

windows - arm - how to check endianness of an object file - Stack Overflow

Special Sections (linuxbase.org)

elf(5) - Linux manual page (man7.org)

How to use push{lr} pop{pc} in ARM Assembly - Stack Overflow

what is the purpose of push{lr} and pop{pc} in arm-asm - Raspberry Pi Forums

Q10 source

https://godbolt.org/z/os1d9z56o