

EE3220 Assignment 2 – Suggested Solution

Q1. Consider the assembly code which the compiler generates for a C function. Explain what each assembly instruction does and describe what data is in the register.

1. `;;;5 void fn(int8_t * a, int32_t * b, float * c) {
000000 b5f0 PUSH {r4-r7,lr}`

Saves registers r4-r7 and LR on stack to save registers which need to be restored to their original values.

2. `000002 b085 SUB sp,sp,#0x14`

Subtract 0x14 from the stack pointer register and stores it back into stack pointer register.

3. `000004 4604 MOV r4,r0`

Save first argument (a) in r4.

4. `000006 460d MOV r5,r1`

Save second argument (b) in r5.

5. `000008 4616 MOV r6,r2`

Save third argument (c) in r6.

6. `;;;6 volatile int8_t a1, a2;
;;;7 volatile int32_t b1, b2;
;;;8 volatile float c1, c2;
;;;9
;;;10 a1 = 15;
00000a 270f MOVS r7,#0xf`

Load 15 (0xf) into register for a1.

7. `;;;11 a2 = -14;
00000c 200d MOVS r0,#0xd`

Load 14 into register for a2.

8. `00000e 43c0 MVNS r0,r0`

Complement (of r0) 14 to -14 using Move Inverse instruction.

9. `000010 9004 STR r0,[sp,#0x10]`

Store a2 on stack at offset 0x10.

10. ;;;12 *a = a1*a2;
000012 9804 LDR r0,[sp,#0x10]

Reload a2 from stack into r0

11. 000014 4378 MULS r0,r7,r0

Multiply a2 by a1, putting result in register for a1.

12. 000016 b240 SXTB r0,r0

Sign extend a1.

13. 000018 7020 STRB r0,[r4,#0]

Store **one-byte result** in memory location pointed to by r4, which is argument a.

14. ;;;13
;;;14 b1 = 15;
00001a 200f MOVS r0,#0xf

Load register r0 (for variable b1) with 15 (0xf).

15. 00001c 9003 STR r0,[sp,#0xc]

Save on stack at offset 0xc.

16. ;;;15 b2 = -14;
00001e 200d MOVS r0,#0xd

Load 14 into register for (register r0) b2.

17. 000020 43c0 MVNS r0,r0

Complement 14 to -14 using Move Inverse instruction.

18. 000022 9002 STR r0,[sp,#8]

Store b2 on stack at offset 8.

19. ;;;16 *b = b1*b2;
000024 9902 LDR r1,[sp,#8]

Load r1 with b2 from stack.

20. 000026 9803 LDR r0,[sp,#0xc]

Load r0 with b1 from stack.

21. 000028 4348 MULS r0,r1,r0

Multiply b1 and b2, and store the result in r0.

22. 00002a 6028 STR r0,[r5,#0]

Store **longword result** in memory location pointed to by r5, which is argument b.

23. ;;;17
 ;;;18 c1 = 15;
00002c 4809 LDR r0,|L1.84|

Load r0 with floating point value of 15 from literal pool.

24. 00002e 9001 STR r0,[sp,#4]

Store in c1's location on the stack.

25. ;;;19 c2 = -14;
000030 4809 LDR r0,|L1.88|

Load r0 with floating point value of 15 from literal pool.

26. 000032 9000 STR r0,[sp,#0]

Store in c2's location on the stack.

27. ;;;20 *c = c1*c2;
000034 9900 LDR r1,[sp,#0]

Load r1 with c2.

28. 000036 9801 LDR r0,[sp,#4]

Load r0 with c1.

29. 000038 f7fffffe BL __aeabi_fmul

Call floating point multiply instruction to multiply c1 and c2.

30. 00003c 6030 STR r0,[r6,#0]

Store longword result (returned in register r0) to memory pointed to by r6, which is argument c.

31. ;;;21
 ;;;22 }
00003e b005 ADD sp,sp,#0x14

Deallocate the stack space for this function.

32. 000040 bdf0 POP {r4-r7,pc}

Restore registers r4 through r7 to original values by popping them off of the stack and then return from subroutine (return address also saved on stack).

Use an online compiler (<https://godbolt.org/>) to complete the following questions.

2. Compile the following C code snippet with optimization level **O0** and **O1**

```
int foo1() {  
    int sum = 0;  
    unsigned int i = 0;  
    for (i = 1; i <= 15; i++) {  
        sum += i*i*i;  
    }  
    return sum;  
}
```

Which instruction(s) computes the return value of the function? Compare and explain the difference between two optimization levels. (Hint: **movw** moves a 16-bit immediate to the lower 16-bit of a register)

Solution

The screenshot displays the Godbolt online compiler interface for the C code snippet. It compares the assembly output for two optimization levels: O0 (left) and O1 (right).

O0 Assembly (Left):

```
1  foo1():  
2      sub    sp, sp, #8  
3      mov    r0, #0  
4      str    r0, [sp, #4]  
5      str    r0, [sp]  
6      mov    r0, #1  
7      str    r0, [sp]  
8      b      .LBB0_1  
9  .LBB0_1:  
10     ldr    r0, [sp]  
11     cmp    r0, #15  
12     bhi    .LBB0_4  
13     b      .LBB0_2  
14  .LBB0_2:  
15     ldr    r0, [sp]  
16     mul    r1, r0, r0  
17     ldr    r2, [sp, #4]  
18     mla    r3, r1, r0, r2  
19     str    r3, [sp, #4]  
20     b      .LBB0_3  
21  .LBB0_3:  
22     ldr    r0, [sp]  
23     add    r0, r0, #1  
24     str    r0, [sp]  
25     b      .LBB0_1  
26  .LBB0_4:  
27     ldr    r0, [sp, #4]  
28     add    sp, sp, #8  
29     bx     lr
```

O1 Assembly (Right):

```
1  foo1():  
2      mov    r0, #14400  
3      bx     lr
```

movw r0, #18496

At optimization level `-O0`, no optimization is done by the compiler, which generates code for all of the instructions including the dead code. The code is easy to debug.

At optimization level `-O1`, the code is optimized and the compiler does not generate the assembly instructions for the dead code. However, the optimized code has less number of instruction which will be less traceable in the debugging process.

3. Compile the following C code snippet with optimization level **O1**

```
int foo2(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n + foo2(n-1);  
    }  
}
```

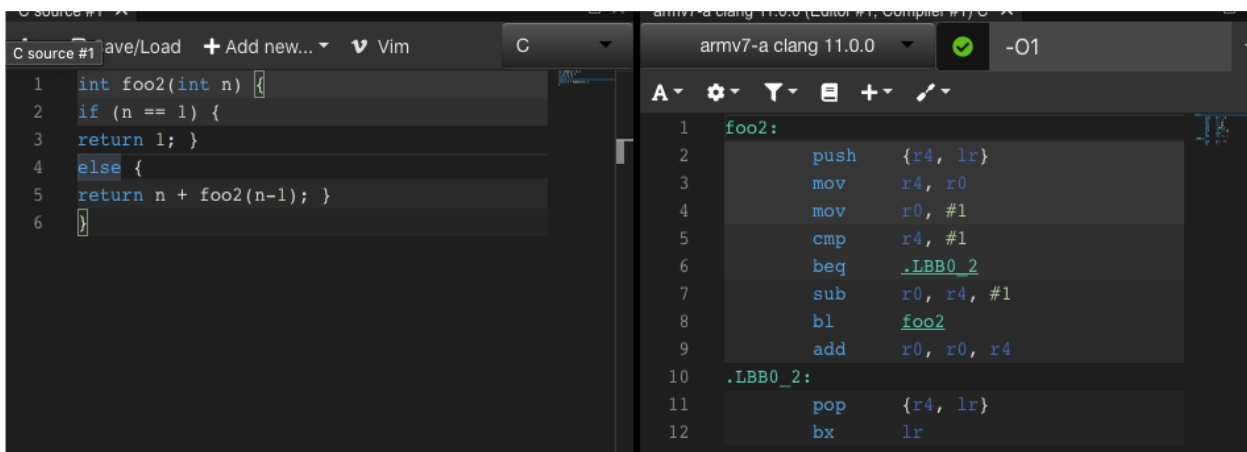
Which register(s) are saved before entering the recursive call? Explain why. (Hint: the registers have their special purposes)

Solution

`r11` is the frame pointer register which is used as the reference pointer for local variables.

`lr` is the link register which holds the return address of the subroutine.

The `r4` and `r10` is a callee-saved register for local variables. Their register values must be preserved across calls.



The screenshot shows a code editor with two panels. The left panel displays the C source code for the `foo2` function. The right panel shows the assembly code generated by the compiler at optimization level `-O1`. The assembly code for `foo2` is as follows:

```
1  foo2:  
2      push    {r4, lr}  
3      mov     r4, r0  
4      mov     r0, #1  
5      cmp     r4, #1  
6      beq     .LBB0_2  
7      sub     r0, r4, #1  
8      bl      foo2  
9      add     r0, r0, r4  
10     .LBB0_2:  
11     pop     {r4, lr}  
12     bx      lr
```

4. Compile the following C code snippet with optimization level **O0** and **O2**

```
struct test_t {  
    short int x;  
    short int y;  
};  
  
int foo3(int *list, struct test_t t) {  
    int ret;  
    ret = t.x;  
    unsigned int i;  
    for (int i = 0; i < 10; i++) {  
        ret = ret + list[i] + t.y;  
    }  
    return ret;  
}
```

- a. How many registers are used for the function arguments?

Solution

2 registers.

- b. If $t.x = 0x5539$ and $t.y = 0x3b47$, how is the argument t stored in the parameter register(s)?

Hint

- asr: Arithmetic Shift Right
- sxth: signed extended lower 16-bit

Solution

0x0000553900003b47

- c. For optimization level **O2**, how many loop iterations do the assembly code perform?

Solution

1 iteration. The loop is fully unrolled.

5. Combine all functions above with the following main function. Compile it with optimization level **00**.

```
int main(void) {  
    volatile int s1;  
    volatile int s2;  
    volatile int s3;  
    struct test_t t;  
    int list[10] = {32, 43, 25, 43, 91, 66, 32, 29, 13, 77};  
  
    s1 = foo1();  
    s2 = foo2(56);  
    t.x = s1 + s2;  
    t.y = s1 - s2;  
    s3 = foo3(list, t);  
}
```

Hint: Checks all **ldr**

Fill in the following table

Local Variables	Offset from Stack Pointer (SP)
s1	sp+68
s2	
list	
t.y	
t.y	

Solution

Local Variables	Offset from Stack Pointer (SP)
s1	sp+68
s2	sp+64
list	sp+4
t.y	sp+12
t.y	sp+14

6. Consider the following assembly code. Write an equivalent C program.

```
bar(int):
    push    {r4, r5, r11, lr}
    add     r11, sp, #8
    mov     r4, r0
    mov     r0, #1
    cmp     r4, #2
    blo     .LBB0_2
    sub     r0, r4, #1
    bl      bar(int)
    mov     r5, r0
    sub     r0, r4, #2
    bl      bar(int)
    mul     r0, r0, r5
    add     r0, r0, #3
.LBB0_2:
    pop     {r4, r5, r11, pc}
```

Hint

- blo: Branch if unsigned less than
- bl: Saves PC+4 in link register and jumps to a function

Solution

```
int bar(int n) {
    if (n == 0 | n == 1) {
        return 1;
    } else {
        return bar(n-1) * bar(n-2) + 3;
    }
}
```