GEBZE TECHNICAL UNIVERSITY

ELECTRONICS ENGINEERING

ELEC334

MICROPROCESSORS

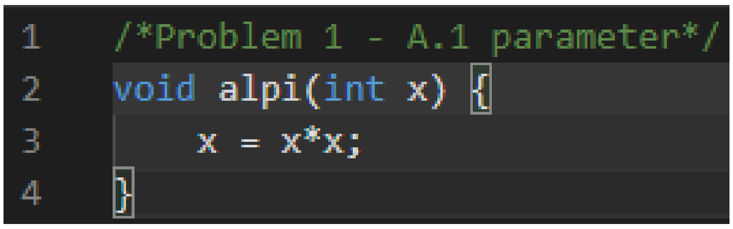
Homework #3

Prepared by
1801022022 – Alperen Karataş

Problem 1. Function Calls

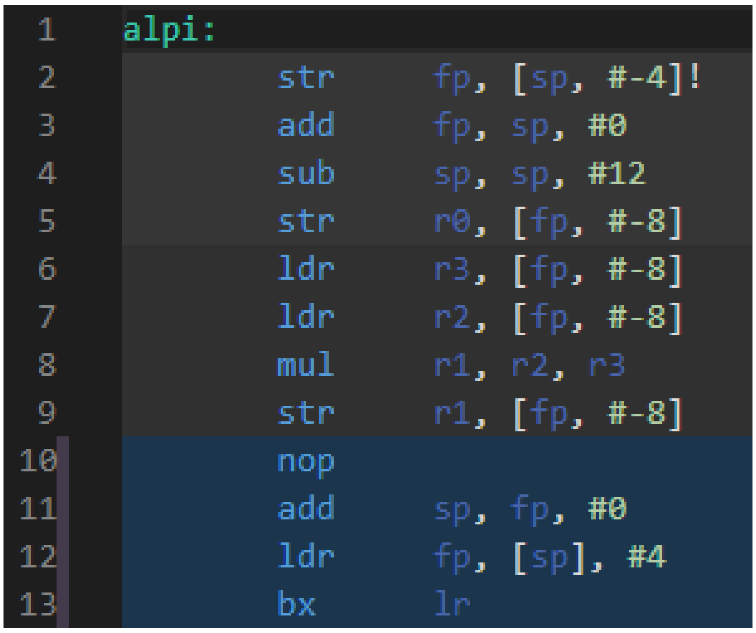
To find out which registers a function written in the C Programming Language uses, the Assembly Language for Cortex M0+ is looked at.

A. 1 parameter



```c
/*Problem 1 - A.1 parameter*/
void alpi(int x) {
    x = x*x;
}
```

Figure 1. C function with 1 parameter



```
alpi:
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        sub     sp, sp, #12
        str     r0, [fp, #-8]
        ldr     r3, [fp, #-8]
        ldr     r2, [fp, #-8]
        mul     r1, r2, r3
        str     r1, [fp, #-8]
        nop
        add     sp, fp, #0
        ldr     fp, [sp], #4
        bx      lr
```

Figure 2. C function with 1 parameter Assembly representation

The Assembly equivalent of the C function, which is written as an example, seen in Figure 1, is seen in Figure 2. The registers used for this function can be explained as follows, based on Figure 2.

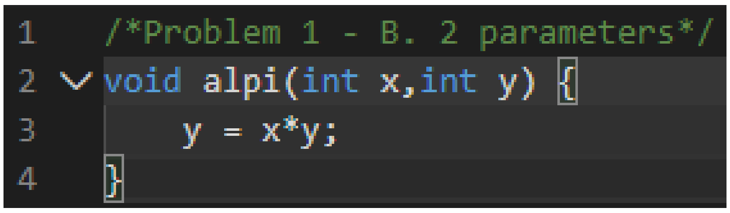
`r0,r1,r2,r3` → General Purpose Registers. R0-R12 registers are used for general purpose, while R12-R15 registers are used for specific use. Because of the related registers are included in the R0-R12 range, these registers can be included in the General Purpose Registers class. These registers, as the name suggests, are used for general purpose writing.

`fp (frame pointer)` → The frame pointer is used as a base pointer in local variables in the stack. That means it points to the base of the stack. It does not move for the duration of the subroutine call.

`sp (stack pointer)` → The stack pointer points to the data stored by coming to the stack with the push down method. A stack pointer is a register that stores and shows the address of this data. (`sp=R13`)

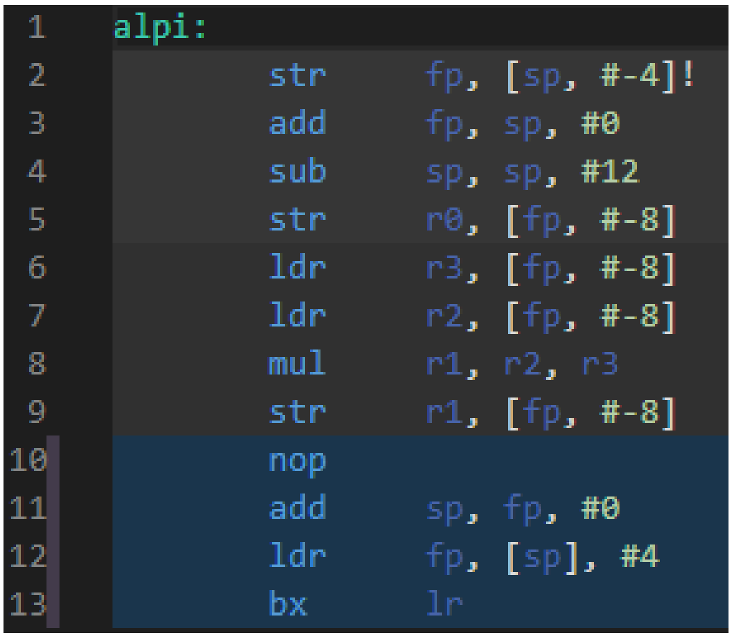
`lr (link register)` → Link Register stores the Return Link. Link register is a register value that stores the address to return when returning from an address that is going with the **"branch"** instruction. A link register value is an updateable value. (`lr=R14`)

B. 2 parameters



```c
/*Problem 1 - B. 2 parameters*/
void alpi(int x,int y) {
    y = x*y;
}
```

Figure 3. C function with 2 parameters



```
alpi:
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        sub     sp, sp, #12
        str     r0, [fp, #-8]
        ldr     r3, [fp, #-8]
        ldr     r2, [fp, #-8]
        mul     r1, r2, r3
        str     r1, [fp, #-8]
        nop
        add     sp, fp, #0
        ldr     fp, [sp], #4
        bx      lr
```

Figure 4. C function with 2 parameters Assembly representation

Figure 4 is reviewed.

Used registers:

`r0,r1,r2,r3 (General Purpose Registers)`

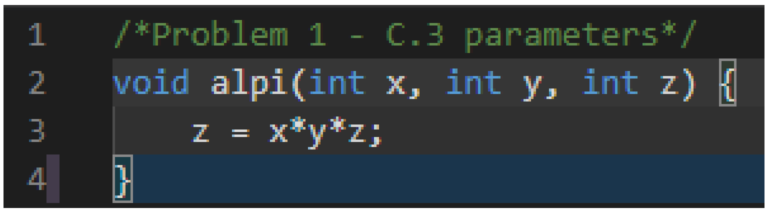
`fp (frame pointer)`

`sp (stack pointer,R13)`

`lr (link register)`

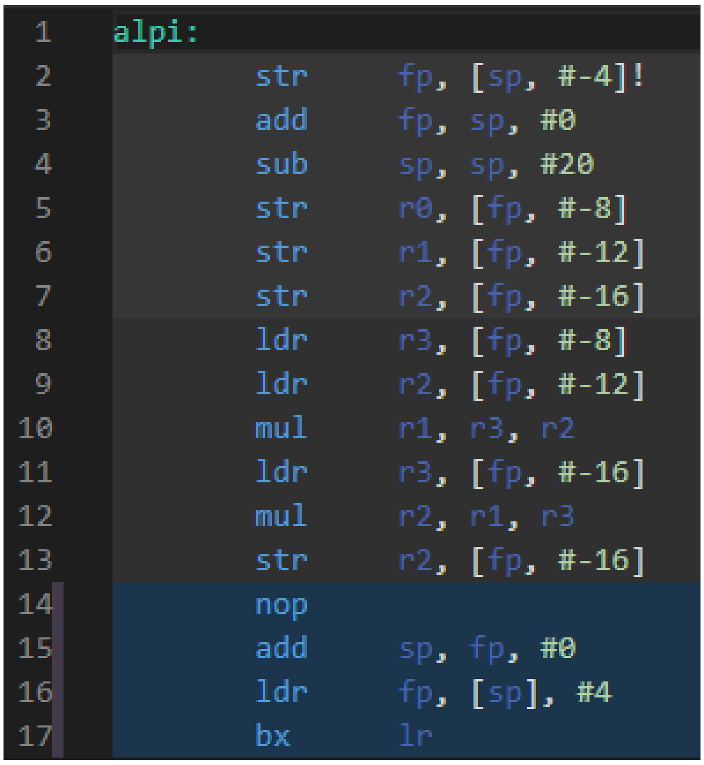
This registers work principle is described in option A.

C. 3 parameters



```c
/*Problem 1 - C.3 parameters*/
void alpi(int x, int y, int z) {
    z = x*y*z;
}
```

Figure 5. C function with 3 parameters



```
alpi:
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        sub     sp, sp, #20
        str     r0, [fp, #-8]
        str     r1, [fp, #-12]
        str     r2, [fp, #-16]
        ldr     r3, [fp, #-8]
        ldr     r2, [fp, #-12]
        mul     r1, r3, r2
        ldr     r3, [fp, #-16]
        mul     r2, r1, r3
        str     r2, [fp, #-16]
        nop
        add     sp, fp, #0
        ldr     fp, [sp], #4
        bx      lr
```

Figure 6. C function with 3 parameters Assembly representation

Figure 6 is reviewed.

Used registers:

`r0,r1,r2,r3 (General Purpose Registers)`

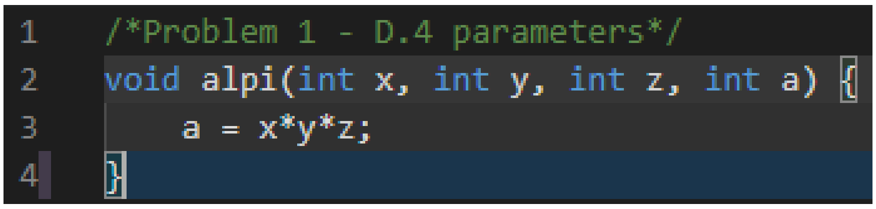
`fp (frame pointer)`

`sp (stack pointer,R13)`

`lr (link register)`

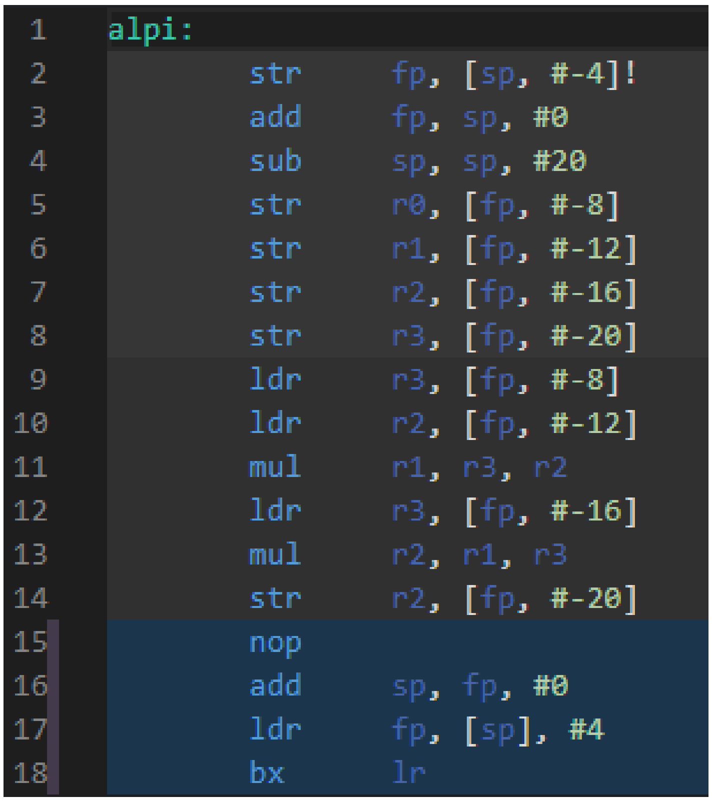
This registers work principle is described in option A.

D. 4 parameters



```c
/*Problem 1 - D.4 parameters*/
void alpi(int x, int y, int z, int a) {
    a = x*y*z;
}
```

Figure 7. C function with 4 parameters



```
alpi:
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        sub     sp, sp, #20
        str     r0, [fp, #-8]
        str     r1, [fp, #-12]
        str     r2, [fp, #-16]
        str     r3, [fp, #-20]
        ldr     r3, [fp, #-8]
        ldr     r2, [fp, #-12]
        mul     r1, r3, r2
        ldr     r3, [fp, #-16]
        mul     r2, r1, r3
        str     r2, [fp, #-20]
        nop
        add     sp, fp, #0
        ldr     fp, [sp], #4
        bx      lr
```

Figure 8. C function with 4 parameters Assembly representation

Figure 8 is reviewed.

Used registers:

`r0,r1,r2,r3 (General Purpose Registers)`

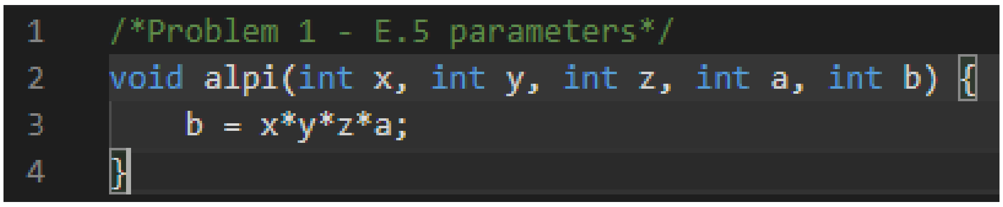
`fp (frame pointer)`

`sp (stack pointer,R13)`

`lr (link register)`

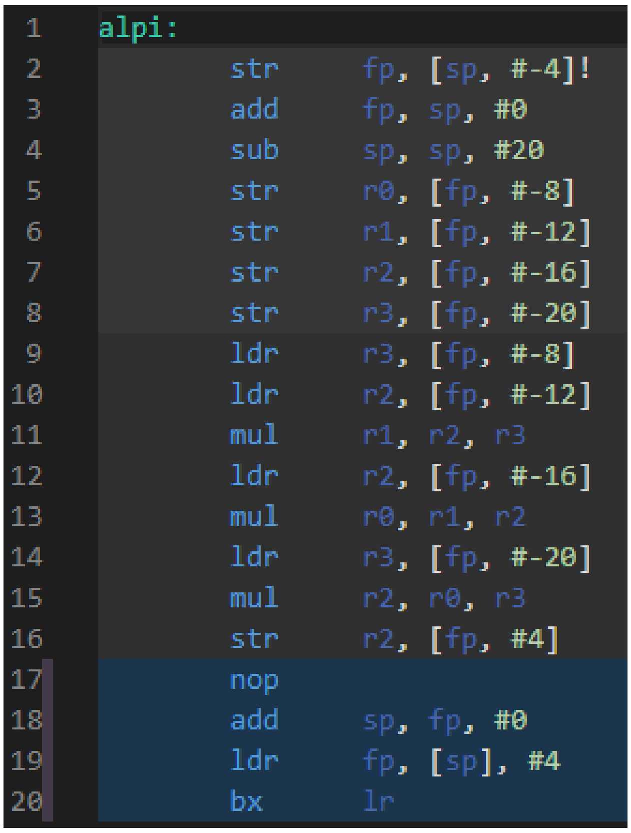
This registers work principle is described in option A.

E. 5 parameters



```c
/*Problem 1 - E.5 parameters*/
void alpi(int x, int y, int z, int a, int b) {
    b = x*y*z*a;
}
```

Figure 9. C function with 5 parameters



```
alpi:
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        sub     sp, sp, #20
        str     r0, [fp, #-8]
        str     r1, [fp, #-12]
        str     r2, [fp, #-16]
        str     r3, [fp, #-20]
        ldr     r3, [fp, #-8]
        ldr     r2, [fp, #-12]
        mul     r1, r2, r3
        ldr     r2, [fp, #-16]
        mul     r0, r1, r2
        ldr     r3, [fp, #-20]
        mul     r2, r0, r3
        str     r2, [fp, #4]
        nop
        add     sp, fp, #0
        ldr     fp, [sp], #4
        bx      lr
```

Figure 10. C function with 5 parameters Assembly representation

Figure 10 is reviewed.

Used registers:

`r0,r1,r2,r3 (General Purpose Registers)`

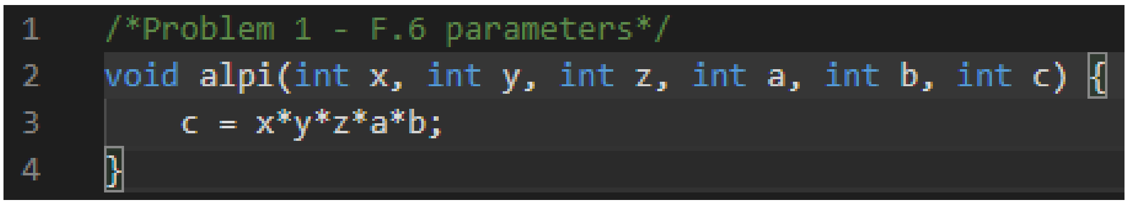
`fp (frame pointer)`

`sp (stack pointer,R13)`

`lr (link register)`

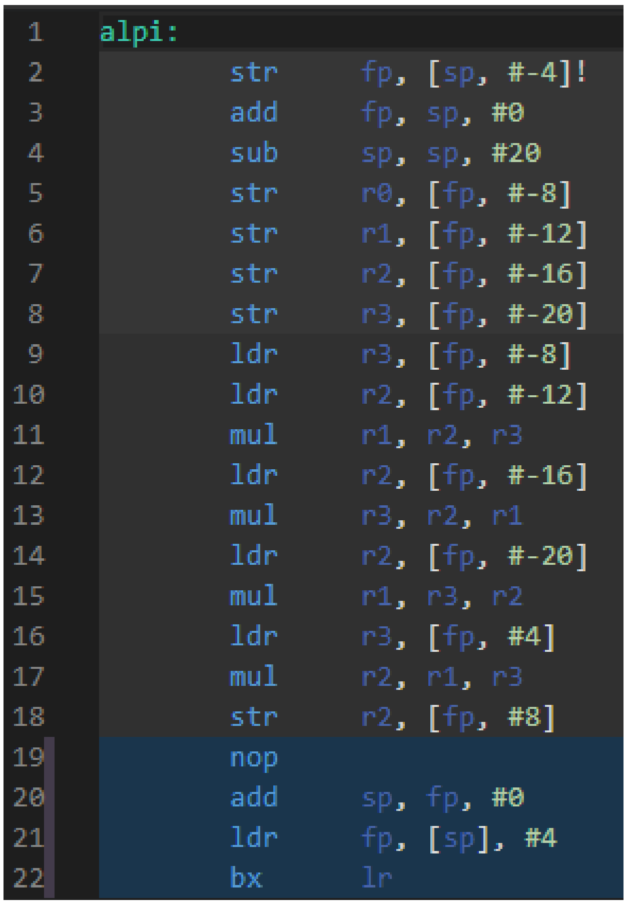
This registers work principle is described in option A.

F. 6 parameters



```c
/*Problem 1 - F.6 parameters*/
void alpi(int x, int y, int z, int a, int b, int c) {
    c = x*y*z*a*b;
}
```

Figure 11. C function with 6 parameters



```asm
alpi:
        str     fp, [sp, #-4]!
        add     fp, sp, #0
        sub     sp, sp, #20
        str     r0, [fp, #-8]
        str     r1, [fp, #-12]
        str     r2, [fp, #-16]
        str     r3, [fp, #-20]
        ldr     r3, [fp, #-8]
        ldr     r2, [fp, #-12]
        mul     r1, r2, r3
        ldr     r2, [fp, #-16]
        mul     r3, r2, r1
        ldr     r2, [fp, #-20]
        mul     r1, r3, r2
        ldr     r3, [fp, #4]
        mul     r2, r1, r3
        str     r2, [fp, #8]
        nop
        add     sp, fp, #0
        ldr     fp, [sp], #4
        bx      lr
```

Figure 12. C function with 6 parameters Assembly representation

Figure 12 is reviewed.

Used registers:

`r0,r1,r2,r3 (General Purpose Registers)`

`fp (frame pointer)`

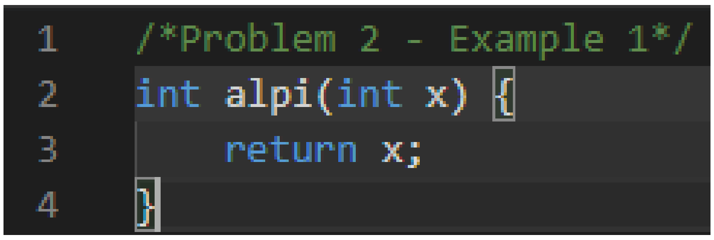
`sp (stack pointer,R13)`

`lr (link register)`

This registers work principle is described in option A.

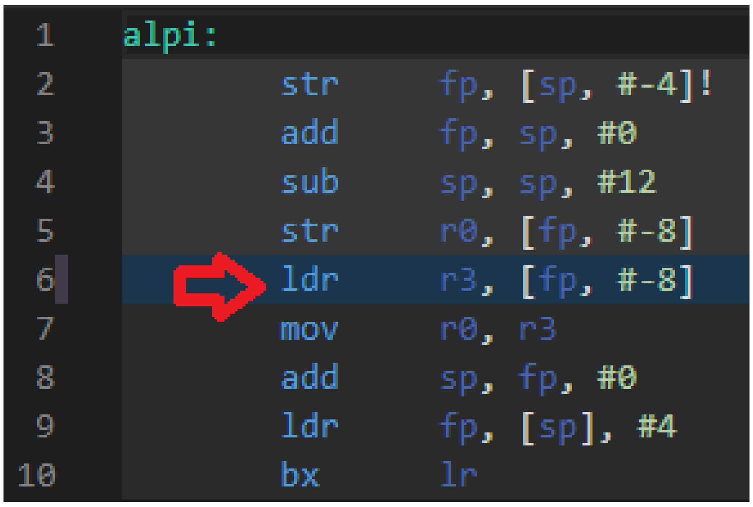
Problem 2. Return Values

Example 1:



```c
/*Problem 2 - Example 1*/
int alpi(int x) {
    return x;
}
```

Figure 13. C function with 1 parameter



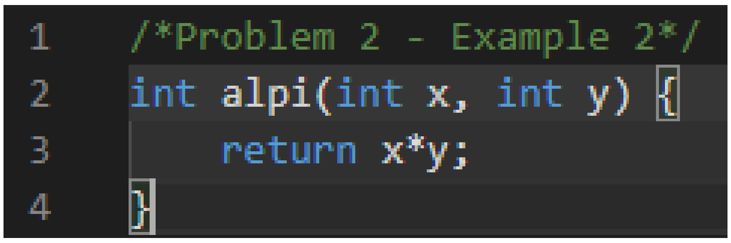
```
 1  alpi:
 2          str     fp, [sp, #-4]!
 3          add     fp, sp, #0
 4          sub     sp, sp, #12
 5          str     r0, [fp, #-8]
 6          ldr     r3, [fp, #-8]
 7          mov     r0, r3
 8          add     sp, fp, #0
 9          ldr     fp, [sp], #4
10          bx      lr
```

Figure 14. C function with 1 parameter Assembly representation

The Assembly Language equivalent of a single-parameter function written in C, Figure 14. The 6th line of the code block in Figure 14, that is, the section shown with the red arrow, corresponds to return.

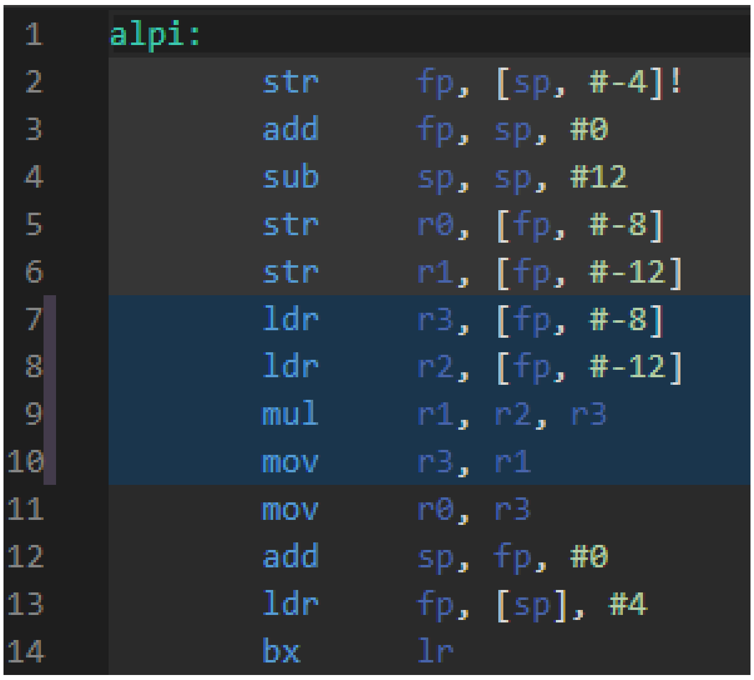
As can be understood in this function, the return process is the writing of a general purpose register (in this case it's `r3`),where the **frame pointer** decimally holds a address value of less than 8.

Example 2:



```c
/*Problem 2 - Example 2*/
int alpi(int x, int y) {
    return x*y;
}
```

Figure 15. C function with 2 parameters



```
1   alpi:
2           str     fp, [sp, #-4]!
3           add     fp, sp, #0
4           sub     sp, sp, #12
5           str     r0, [fp, #-8]
6           str     r1, [fp, #-12]
7           ldr     r3, [fp, #-8]
8           ldr     r2, [fp, #-12]
9           mul     r1, r2, r3
10          mov     r3, r1
11          mov     r0, r3
12          add     sp, fp, #0
13          ldr     fp, [sp], #4
14          bx      lr
```

Figure 16. C function with 2 parameters Assembly representation

The Assembly equivalent of the C code seen in Figure 16. In Figure 16, the code block (rows 7-8-9-10) painted in blue corresponds to return.

This block of code can be explained as the `r3` and `r2` registers being written to the decimal 8 and 12 less value addresses of the **frame pointer**, respectively, and they are multiplied and written to the `r1` register, and the result is thrown back to `r3`.

As can be seen from the examples, the **return** operation can be seen as writing values in assembly language to a number of addresses with the **`ldr`** instruction.

Problem 3. Reverse me if you can

The disassembleed version of the given elf file is as follows:

```
08000000 <v>:
 8000000:       10002000        .word   0x10002000
 8000004:       08000021        .word   0x08000021
 8000008:       0800002b        .word   0x0800002b
 800000c:       0800002b        .word   0x0800002b
 8000010:       10000000        .word   0x10000000
 8000014:       10000000        .word   0x10000000
 8000018:       10000000        .word   0x10000000
 800001c:       10000000        .word   0x10000000

08000020 <r>:
 8000020:       481b            ldr     r0, [pc, #108]  ; (8000090
<lizard+0x10>)
 8000022:       4685            mov     sp, r0
 8000024:       f000 f802       bl      800002c <main>
 8000028:       e7fe            b.n     8000028 <r+0x8>

0800002a <d>:
 800002a:       e7fe            b.n     800002a <d>

0800002c <main>:
 800002c:       4919            ldr     r1, [pc, #100]  ; (8000094
<lizard+0x14>)
 800002e:       4a1a            ldr     r2, [pc, #104]  ; (8000098
<lizard+0x18>)
 8000030:       2300            movs    r3, #0
```

```
08000032 <rock>:
 8000032:       f000 f807       bl      8000044 <paper>
 8000036:       6010            str     r0, [r2, #0]
 8000038:       3104            adds    r1, #4
 800003a:       3204            adds    r2, #4
 800003c:       3301            adds    r3, #1
 800003e:       2b04            cmp     r3, #4
 8000040:       d1f7            bne.n   8000032 <rock>
 8000042:       e017            b.n     8000074 <eof>

08000044 <paper>:
 8000044:       b40e            push    {r1, r2, r3}
 8000046:       4e15            ldr     r6, [pc, #84]   ; (800009c
<lizard+0x1c>)
 8000048:       00f7            lsls    r7, r6, #3
 800004a:       6809            ldr     r1, [r1, #0]
 800004c:       4c14            ldr     r4, [pc, #80]   ; (80000a0
<lizard+0x20>)

0800004e <scissors>:
 800004e:       4a15            ldr     r2, [pc, #84]   ; (80000a4
<lizard+0x24>)
 8000050:       6815            ldr     r5, [r2, #0]
 8000052:       0108            lsls    r0, r1, #4
 8000054:       1940            adds    r0, r0, r5
 8000056:       b401            push    {r0}
 8000058:       6855            ldr     r5, [r2, #4]
 800005a:       0948            lsrs    r0, r1, #5
 800005c:       1940            adds    r0, r0, r5
 800005e:       19ca            adds    r2, r1, r7
 8000060:       4050            eors    r0, r2
 8000062:       bc04            pop     {r2}
 8000064:       4050            eors    r0, r2
```

```
8000066:       1a09            subs    r1, r1, r0
 8000068:       1bbf            subs    r7, r7, r6
 800006a:       0864            lsrs    r4, r4, #1
 800006c:       d1ef            bne.n   800004e <scissors>
 800006e:       0008            movs    r0, r1
 8000070:       bc0e            pop     {r1, r2, r3}
 8000072:       4770            bx      lr

08000074 <eof>:
 8000074:       e7fe            b.n     8000074 <eof>
 8000076:       46c0            nop                     ; (mov r8,
r8)

08000078 <spock>:
 8000078:       138a5b9c        .word   0x138a5b9c
 800007c:       83b19de5        .word   0x83b19de5

08000080 <lizard>:
 8000080:       a2390c55        .word   0xa2390c55
 8000084:       113f39fc        .word   0x113f39fc
 8000088:       6140f4fd        .word   0x6140f4fd
 800008c:       d3926c34        .word   0xd3926c34
 8000090:       10002000        .word   0x10002000
 8000094:       08000080        .word   0x08000080
 8000098:       10000200        .word   0x10000200
 800009c:       14159265        .word   0x14159265
 80000a0:       00000080        .word   0x00000080
 80000a4:       08000078        .word   0x08000078
```

Unfortunately, the solution to the problem has not been followed from this point on.

Problem 4. Reading:

Faults, Injection Methods, and Fault Attacks

Collecting information from hidden keys in embedded systems, and thing of errors for purposes such as intervention in the process algorithm, is called an fault attack. An attacker could carry out various types of attacks, and there could be several models of error types. Interventions in these errors can also have several methods.

One type of attack, glitch attack, feed voltage, and changes in the external clock are examples of this type of attack. Fault detectors and DC filters are used to prevent such attacks.

Temperature attack, as its name suggests, is a type of attack using a temperature factor. Most devices cannot work under high temperatures. To prevent this situation, many cards have a temperature detector.

It would not be wrong to say that light attack is the most effective method of attack for today. The fact that it is cheap and effective enough to be done even with a camera flash, that it can be made to a designated point as opposed to the glitch attack attack type, and that it can be easily applied to the back of a chip, are the biggest examples.

It is one of the types of attacks that can be carried out in an attack as a result of the application of magnetic emission by pouring current over a coil and obtaining a magnetic field.

An attack can cause two types of faults, a permanent fault and a transient fault. A permanent fault may damage ram or EEPROM and the damage may be large. The transient fault will not be permanent as understood. Methods such as reset of the circuit may be the solution to this error.

A bit versus byte model, a type of error model, is a model that is trying to create an error by trying to play on a bit or byte. An attacker might try to randomly change the value of the data, which is a little easier than others. An attack model that targets an error on memory or during a calculation is known as static versus computational. It is a difficult option to intervene in memory compared to calculation error. Control errors may occur due to a skip state that occurs in some operations or operations due to errors. A control error, although difficult in terms of induction, can be quite powerful.

Problem 5. Reading:

Controlling PC on ARM Using Fault Injection

In this article, we talk about the error injection attack technique for ARM 32-bit with ARM architecture. Now we will talk about some bug settled techniques and error models.

The Clock Fault Injection Faults method can take effect when an attacker uses a chip that uses an external clock signal. An injection is applied in the transition between faster and slower clock signals.

Voltage Fault Injection Faults, as the name suggests, affect the target area by switching between different voltage levels.

With Electromagnetic Fault Injection, it is possible to influence certain areas of the chip. At the core of this technique are electromagnetic emissions from a coil with high current flowing over it.

The Optical Fault Injection Faults method can be expressed as sending a laser beam to a local area inside the chip.

ARM has two instructions, LDR and STR, that can do things like write data to memory and pull data from memory. With these instructions, data can be controlled. It is not considered security sensitive. Additionally, the attacker does not need to follow specific instructions. For an attacker, these operations and instructions can be seen as targets.

Two types of attacks can be mentioned. One is done during boot, while the other is done during operation. When booting, the attack is called boot attack. Applies during startup of the embedded system. Volumes such as ROM, RAM, and Flash are open to external influences. The SOC that contains these volumes does not rely on external codes. It tries to prevent an attack. An attack during operation is known as a runtime attack. In a process based on copy exchange between REE and TEE volumes in the embedded system, TEE may be attacked when trying to copy data from REE.

…continues from next page

The probability of externally controlled data being uploaded to the PC by an installation instruction has been controlled by a simulation. The simulation program does the job of loading data using the LDR instruction. As long as the bit to be set to do the copy job works properly, a glitch that might work can convert processed data into anything. As a result of the simulation, it was observed that the PC register can modify the LDR and LDMIA instructions using error injection as long as it is suitable for injecting errors.

When the time came to test what was done, the experiment applied in the simulation was used for testing. The core that powers the ARM processor is designated as the target and a glitch attack has been applied to this zone. Voltage Fault Injection Faults method was used to solve the error. When all the parameters required to solve the error were applied, a full 27 glitches were detected in the serial interface.

In addition, in the fields of hardware and software, it would be very useful to base the three basic principles called deflect, detect and react to reduce errors.