

**ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO**  
**DISCIPLINA: LABORATÓRIO DE PROCESSADORES- PCS3732**  
**1º QUADRIMESTRE/2021**



**Aula 8**  
**01 de Julho de 2021**

**GRUPO 10**

Arthur Pires da Fonseca  
Bruno José Mório  
Iago Soriano Roque Monteiro

NUSP: 10773096  
NUSP: 10336852  
NUSP: 8572921

# Sumário

<b>8.2.1 - Compilar código assembly e código C e linkar ambos.</b>	<b>3</b>
<b>8.2.2 - Alterar o código C inserindo assembly no meio do código C.</b>	<b>4</b>
<b>8.2.3 - Observar como funciona a recursão.</b>	<b>6</b>
<b>Apêndice</b>	<b>8</b>
8.2.1	8
int2str.s	8
imprime.c	8
impnum.c	9
8.2.2	9
8.2.3	10

## 8.2.1 - Compilar código assembly e código C e linkar ambos.

O código `impnum.c` chama `int2str.s` com um nusp de 8 dígitos.

```
File Edit View Search Terminal Help
Register group: general
r0 0xa 10 r1
r3 0x0 0 r4
r6 0x0 0 r7
r9 0x10b60 68448 r10
r12 0x1fff00 2096896 sp
pc 0x8298 33432 fps

6 {
7     int num = 0x12344321;
8     char *st = malloc(5*sizeof(char));
9     int2str(num, st);
10    puts(st);
> 11    printf("%s\n", st);
12    return 0;
13
14 }
15 /*
16 int main()
```

```
sim process 42 In: main
Loading section .jcr, size 0x4 vma 0x10a38
Start address 0x8110
Transfer rate: 281056 bits/sec.
(gdb) b main
Breakpoint 1 at 0x826c: file impnum.c, line 7.
(gdb) r
Starting program: /home/student/src/lab8/a.out

Breakpoint 1, main () at impnum.c:7
(gdb) n
12344321
(gdb) □
```

O código `imprime.c` chama `int2str.s` por chamadas recursivas, começando em 5.

```
File Edit View Search Terminal Help
Register group: general
r0 0xffffffff -1 r1
r3 0xffffffff -1 r4
r6 0x0 0 r7
r9 0xbdd0 48592 r10
r12 0x1fff58 2096984 sp
pc 0x82b4 33460 fps

imprime.c
9     return;
10    }
11    // char ptr[6];
12    char *ptr = malloc(6*sizeof(char));
13    int2str(N, ptr);
14    puts(ptr);
15    imprime(N-1);
> 16 }
17
18 int main(void){
19     imprime(5);
B+
```

```
sim process 42 In: imprime
Breakpoint 1, main () at imprime.c:19
(gdb) s
imprime (N=5) at imprime.c:8
(gdb) n
(gdb) n
5
4
3
2
1
0
(gdb) □
```

## 8.2.2 - Alterar o código C inserindo assembly no meio do código C.

Estude o código `imprime.s` vindo de `imprime.C`

1. Existe algum formato para os labels gerados automaticamente em `imprime.s`?

Sim, o formato é “.L” seguido de um número.

2. Como o número é passado como parâmetro? Como o `fp` é usado para isso?

```
str r0, [fp, #-16]
```

O número é passado como parâmetro através de shiftar em 16 bytes o endereço do frame pointer.

3. Quais são os registradores atribuídos a: `fp`, `ip`, `sp`, `lr`?

`fp = r11`      `ip = r12`      `sp = r13`      `lr = r14`

4. Para que serve o `fp`?

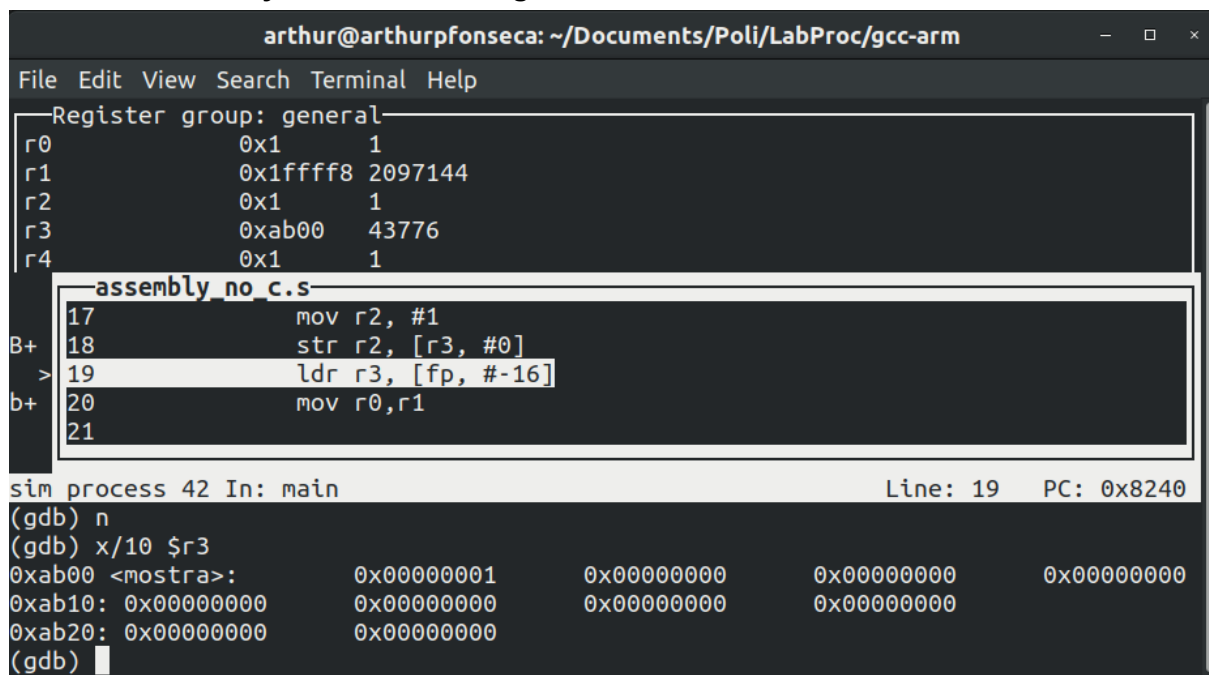
O registrador `fp` registra o topo da fila em cada chamada de uma função.

O `fp` acompanha as variáveis de uma função para outra. Ele é um quadro da pilha para aquela função.

5. Para que serve o `ip`?

Ele tem mais de um uso: pode ser um “registrador de rascunho” para guardar valores intermediários entre chamadas de subrotina. Também pode ser usado como link entre duas subrotinas.

Inserindo assembly no meio do código C



```
arthur@arthurpfonseca: ~/Documents/Poli/LabProc/gcc-arm
File Edit View Search Terminal Help
Register group: general
r0      0x1      1
r1      0x1ffff8 2097144
r2      0x1      1
r3      0xab00   43776
r4      0x1      1
assembly_no_c.s
17      mov r2, #1
18      str r2, [r3, #0]
19      ldr r3, [fp, #-16]
20      mov r0, r1
21
sim process 42 In: main                               Line: 19   PC: 0x8240
(gdb) n
(gdb) x/10 $r3
0xab00 <mostra>:  0x00000001  0x00000000  0x00000000  0x00000000
0xab10: 0x00000000  0x00000000  0x00000000  0x00000000
0xab20: 0x00000000  0x00000000
(gdb)
```

*Antes de inserir o assembly*

```
student:~/src/Lab7$ arm-elf-objdump -t a.out | grep mostra
0000aae8 g      0 .bss      00000004 mostra
student:~/src/Lab7$
```

*Endereço da variável “mostra”*

```

arthur@arthurpfonseca: ~/Documents/Poli/LabProc/gcc-arm
File Edit View Search Terminal Help
Register group: general
r0      0x1      1      r1      0xaae8    43752
r2      0x7b     123    r3      0xa9cc    43468
r4      0x1      1      r5      0x1ffff8  2097144
r6      0x0      0      r7      0x0       0
r8      0x0      0      r9      0x0       0
r10     0x200100 2097408 r11     0x1ffff4  2097140
r12     0x1ffff8 2097144 sp      0x1fffe8  2097128
lr      0x81fc   33276  pc      0x8230  33328
fps     0x0      0      cpsr    0x60000013 1610612755

6      {
7          //mostra = 1234;
8
9
10     __asm__(
11         "ldr r1, =mostra\n"
12         "mov r2, #123\n"
13         "str r2, [r1]\n"
14     );
15
16     return (0);
17 }
18

sim process 42 In: main                                Line: 16   PC: 0x8230
Loading section .ctors, size 0x8 vma 0xa9c8
Loading section .dtors, size 0x8 vma 0xa9d0
Loading section .jcr, size 0x4 vma 0xa9d8
Start address 0x8110
Transfer rate: 83680 bits in <1 sec.
(gdb) b 16
Breakpoint 1 at 0x8230: file assembly_no_c.c, line 16.
(gdb) r
Starting program: /home/student/src/Lab7/a.out

Breakpoint 1, main () at assembly_no_c.c:16
(gdb) p mostra
$1 = 123
(gdb)

```

*Assembly após a inserção das instruções no código em C.*

O código em C pode ser encontrado no Apêndice, em “8.2.2”.

### 8.2.3 - Observar como funciona a recursão.

1. **Como o parâmetro é passado para imprime? Na resposta explique o caso da rotina ter 5 parâmetros (via registrador e pilha) e da rotina ter poucos parâmetros (via registrador).**

O parâmetro é passado para a função imprime pelo registrador r0. Dentro da função imprime ele é colocado em uma stack em uma posição deslocada de 16 de fp. Com 5 parâmetros não é possível registrar tudo que é necessário entre r0-r3, logo os 4 primeiros valores são alocados nos registradores disponíveis e os demais são colocados na pilha.

Para acessar os parâmetros da pilha, utiliza-se um offset aplicado ao endereço que está em fp.

2. **Como esse parâmetro é empilhado (isso é necessário em caso de chamadas recursivas)? Como é aberto um espaço na pilha para o parâmetro de imprime? Onde isso é feito no código?**

Ao entrar no código da função imprime, o sp é salvo em \$ip e os registradores fp, ip, lr e pc (r11 - r14) são empilhados na stack. O primeiro valor a ser empilhado é o argumento da chamada recursiva, seguido do fp, ip, lr e pc.

Em chamadas recursivas este empilhamento é necessário para que se tenha um histórico das execuções. O parâmetro imprime sempre é salvo na pilha com um offset de 16 bytes abaixo em relação ao frame pointer atual, respeitando o intervalo onde os valores intermediários para que nenhuma informação se perca. Assim, o parâmetro é salvo com a instrução `str r0, [fp, #-16]`.

3. **Por que se faz fp-16 para acessar o parâmetro?**

Precisamos voltar 16 bytes para acessar o parâmetro porque o frame pointer é full descending. Assim, o parâmetro recém-inserido encontra-se num endereço anterior ao pointer atual.

4. **Como esse parâmetro é desempilhado? Observe que o ip eh repassado para sp e com isso, a alteração na pilha para abrir o espaço para o parâmetro de "imprime" é automaticamente feito.**

Para desempilhar, ao final do código usa-se o IP (que guarda o endereço da chamada inicial) para encontrar a chamada anterior da rotina recursiva, desempilhando até que encontre a origem das rotinas de chamadas.

**Declare a variável local "int lixo" na função imprime.  
Dentro da função recursiva faça, lixo++.**

**Observe o código assembly gerado pelo compilador.**

**Crie imprime.s para que imprima 7 números de 1 a 7**

Pergunta:

- 1. Responda no relatório: Como "lixo" foi referenciado? Como foi aberto espaço na pilha para o "lixo"? Retire printscreens comparando a pilha sem o uso do int lixo e com o uso de int lixo. Variáveis locais devem ser empilhadas, pois, caso a função seja recursiva, elas fazem parte da recursão.**

A variável "lixo" vai se encontrar no endereço -20 em relação ao original em que foi declarada.

- 2. Quando imprime chama recursivamente imprime é necessário que haja um ponteiro para o fp anterior. Como isso é feito? Quando imprime retorna para uma instância anterior é necessário que o fp retorne para servir de base para os parâmetros e variáveis locais anteriores. Explique como isso é feito.**

O ponteiro para o fp anterior é armazenado na pilha, sempre pela instrução "stmfd sp!, {fp, ip, lr, pc}" depois é recuperado com "ldmfd sp, {fp, sp, pc}"

Quer dizer, o registrador fp sempre guarda qual era o topo da pilha antes da chamada da função atual. Chamadas recursivas, ou de várias funções, armazenam todas as informações necessárias para a volta da função que será chamada, para que depois os endereços de volta sejam, um a um, desempilhados; possibilitando, dessa forma, que exista uma função recursiva.

## Apêndice

### 1. 8.2.1

int2str.s

```
@ Exercicio 8.2.1 do blog
@ Para debugar este codigo:
@ gcc int2str.s && gdb a.out

.text
.globl int2str
int2str:
    @ Recebe os argumentos pelos registradores r0 e r1
    @ r0 = inteiro (em hexadecimal) 0x12344321 ->
    @ sp -> pontstr

    ldr r2, =0xf          @ constante 15 (mascara de 4 bits)
    ldr r4, =0x30;
    ldr r5, =0x1;
    ldmfd sp, {r1}        @ Pega ponteiro pra string
    mov r9, r1;

loop:
    and r3, r2, r0        @ pega 4 bits mais significativos
    add r3, r3, r4        @ soma 30 para transformar em ascii
    strb r3, [r1], #0
    add r1, r1, r5
    mov r0, r0, lsr #4    @ divide por 16 para colocar o próximo no menos sig.

    cmp r0, #0
    bgt loop             @ continua o loop se numero nao foi descascado totalmente
    mov r3, #0
    strb r3, [r1], #0    @ coloca 0 no final
    add r1, r1, r5
    mov r1, r9;
    mov pc, lr          @ acabou a funcao
```

imprime.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

extern void int2str(int nusp, char pontstr[]);

void imprime(int N){
```



```

        if (N<0){
            return;
        }
        // char ptr[6];
        char *ptr = malloc(6*sizeof(char));
        int2str(N, ptr);
        puts(ptr);
        imprime(N-1);
    }

int main(void){
    imprime(5);
    return 0;
}

```

impnum.c

```

#include <stdio.h>
#include <stdlib.h>
extern void int2str(int inteiro, char *pontstr);

int main()
{
    int num = 0x12344321;
    char *st = malloc(5*sizeof(char));
    int2str(num, st);
    puts(st);
    printf("%s\n", st);
    return 0;
}

```

2. 8.2.2

assembly\_no\_c.c

```

#include <stdio.h>

int mostra;

int main()
{
    //mostra = 1234;
}

```

```

__asm__(
    "ldr r1, =mostra\n"
    "mov r2, #123\n"
    "str r2, [r1]\n"
);

return (0);
}

```

assembly\_no\_c.s

```

.file "assembly_no_c.c"
.section .debug_abbrev,"",%progbits
.Ldebug_abbrev0:
.section .debug_info,"",%progbits
.Ldebug_info0:
.section .debug_line,"",%progbits
.Ldebug_line0:
.text
.Ltext0:
.align 2
.global main
.type main, %function
main:
.LFB2:
.file 1 "assembly_no_c.c"
.loc 1 6 0
@ args = 0, pretend = 0, frame = 0
@ frame_needed = 1, uses_anonymous_args = 0
mov ip, sp
.LCFI0:
stmfd sp!, {fp, ip, lr, pc}
.LCFI1:
sub fp, ip, #4
.LCFI2:
.loc 1 10 0
ldr r1, =mostra
mov r2, #123
str r2, [r1]

.loc 1 16 0
mov r3, #0
.loc 1 17 0
mov r0, r3
ldmfd sp, {fp, sp, pc}
.LFE2:

```

```

.size main, .-main
.commmostra,4,4
.section .debug_frame,"",%progbits
.Lframe0:
    .4byte .LECIE0-.LSCIE0
.LSCIE0:
    .4byte 0xffffffff
    .byte 0x1
    .ascii "\000"
    .uleb128 0x1
    .sleb128 -4
    .byte 0xe
    .byte 0xc
    .uleb128 0xd
    .uleb128 0x0
    .align 2
.LECIE0:
.LSFDE0:
    .4byte .LEFDE0-.LASFDE0
.LASFDE0:
    .4byte .Lframe0
    .4byte .LFB2
    .4byte .LFE2-.LFB2
    .byte 0x4
    .4byte .LCFI0-.LFB2
    .byte 0xd
    .uleb128 0xc
    .byte 0x4
    .4byte .LCFI1-.LCFI0
    .byte 0x8e
    .uleb128 0x2
    .byte 0x8d
    .uleb128 0x3
    .byte 0x8b
    .uleb128 0x4
    .byte 0x4
    .4byte .LCFI2-.LCFI1
    .byte 0xc
    .uleb128 0xb
    .uleb128 0x4
    .align 2
.LEFDE0:
    .text
.Letext0:
    .section .debug_info
    .4byte 0x10d
    .2byte 0x2

```

```
.4byte .Ldebug_abbrev0
.byte 0x4
.uleb128 0x1
.4byte .Ldebug_line0
.4byte .Ltext0
.4byte .Ltext0
.ascii "GNU C 3.4.3\000"
.byte 0x1
.ascii "assembly_no_c.c\000"
.ascii "/home/student/src/Lab7\000"
.uleb128 0x2
.4byte .LASF0
.byte 0x4
.byte 0x7
.uleb128 0x3
.ascii "long int\000"
.byte 0x4
.byte 0x5
.uleb128 0x3
.ascii "long long int\000"
.byte 0x8
.byte 0x5
.uleb128 0x3
.ascii "int\000"
.byte 0x4
.byte 0x5
.uleb128 0x3
.ascii "unsigned int\000"
.byte 0x4
.byte 0x7
.uleb128 0x2
.4byte .LASF0
.byte 0x4
.byte 0x7
.uleb128 0x3
.ascii "unsigned char\000"
.byte 0x1
.byte 0x8
.uleb128 0x3
.ascii "short int\000"
.byte 0x2
.byte 0x5
.uleb128 0x3
.ascii "char\000"
.byte 0x1
.byte 0x8
.uleb128 0x3
```

```
.ascii "short unsigned int\000"  
.byte 0x2  
.byte 0x7  
.uleb128 0x3  
.ascii "long long unsigned int\000"  
.byte 0x8  
.byte 0x7  
.uleb128 0x4  
.byte 0x1  
.ascii "main\000"  
.byte 0x1  
.byte 0x6  
.4byte 0x70  
.4byte .LFB2  
.4byte .LFE2  
.byte 0x1  
.byte 0x5b  
.uleb128 0x5  
.ascii "mostra\000"  
.byte 0x1  
.byte 0x3  
.4byte 0x70  
.byte 0x1  
.byte 0x5  
.byte 0x3  
.4byte mostra  
.byte 0x0  
.section .debug_abbrev  
.uleb128 0x1  
.uleb128 0x11  
.byte 0x1  
.uleb128 0x10  
.uleb128 0x6  
.uleb128 0x12  
.uleb128 0x1  
.uleb128 0x11  
.uleb128 0x1  
.uleb128 0x25  
.uleb128 0x8  
.uleb128 0x13  
.uleb128 0xb  
.uleb128 0x3  
.uleb128 0x8  
.uleb128 0x1b  
.uleb128 0x8  
.byte 0x0  
.byte 0x0
```

.uleb128 0x2  
.uleb128 0x24  
.byte 0x0  
.uleb128 0x3  
.uleb128 0xe  
.uleb128 0xb  
.uleb128 0xb  
.uleb128 0x3e  
.uleb128 0xb  
.byte 0x0  
.byte 0x0  
.uleb128 0x3  
.uleb128 0x24  
.byte 0x0  
.uleb128 0x3  
.uleb128 0x8  
.uleb128 0xb  
.uleb128 0xb  
.uleb128 0x3e  
.uleb128 0xb  
.byte 0x0  
.byte 0x0  
.uleb128 0x4  
.uleb128 0x2e  
.byte 0x0  
.uleb128 0x3f  
.uleb128 0xc  
.uleb128 0x3  
.uleb128 0x8  
.uleb128 0x3a  
.uleb128 0xb  
.uleb128 0x3b  
.uleb128 0xb  
.uleb128 0x49  
.uleb128 0x13  
.uleb128 0x11  
.uleb128 0x1  
.uleb128 0x12  
.uleb128 0x1  
.uleb128 0x40  
.uleb128 0xa  
.byte 0x0  
.byte 0x0  
.uleb128 0x5  
.uleb128 0x34  
.byte 0x0  
.uleb128 0x3

```

.uleb128 0x8
.uleb128 0x3a
.uleb128 0xb
.uleb128 0x3b
.uleb128 0xb
.uleb128 0x49
.uleb128 0x13
.uleb128 0x3f
.uleb128 0xc
.uleb128 0x2
.uleb128 0xa
.byte 0x0
.byte 0x0
.byte 0x0
.section .debug_pubnames,"",%progbits
.4byte 0x22
.2byte 0x2
.4byte .Ldebug_info0
.4byte 0x111
.4byte 0xe4
.ascii "main\000"
.4byte 0xfb
.ascii "mostra\000"
.4byte 0x0
.section .debug_aranges,"",%progbits
.4byte 0x1c
.2byte 0x2
.4byte .Ldebug_info0
.byte 0x4
.byte 0x0
.2byte 0x0
.2byte 0x0
.4byte .Ltext0
.4byte .Ltext0-.Ltext0
.4byte 0x0
.4byte 0x0
.section .debug_str,"",%progbits
.LASF0:
.ascii "long unsigned int\000"
.ident "GCC: (GNU) 3.4.3"

```

### 3. 8.2.3

imprime-lixo7.s

```
.file "imprime.c"
.section .rodata
.align 2
.LC0:
.ascii "numero = %d\n\000"
.text
.align 2
.global imprime
.type imprime, %function
imprime:
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    mov     ip, sp
    stmfd   sp!, {fp, ip, lr, pc}
    sub     fp, ip, #4
    sub     sp, sp, #8
    str     r0, [fp, #-16]
    ldr     r3, [fp, #-16]
    cmp     r3, #0
    bgt     .L2
    b       .L1
.L2:
    ldr     r0, .L3
    ldr     r1, [fp, #-16]
    bl      printf
    ldr     r3, [fp, #-20]
    add     r3, r3, #1
    str     r3, [fp, #-20]
    ldr     r3, [fp, #-16]
    sub     r3, r3, #1
    mov     r0, r3
    bl      imprime
.L1:
    sub     sp, fp, #12
    ldmfd   sp, {fp, sp, pc}
.L4:
.align 2
.L3:
.word     .LC0
.size     imprime, .-imprime
.align 2
.global   main
```



```

.type    main, %function
main:
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 1, uses_anonymous_args = 0
    mov     ip, sp
    stmfd   sp!, {fp, ip, lr, pc}
    sub     fp, ip, #4
    mov     r0, #7
    bl      imprime
    mov     r0, r3
    ldmfd   sp, {fp, sp, pc}
    .size   main, .-main
    .ident  "GCC: (GNU) 3.4.3"

```

imprime-lixo7.c

```

void imprime(int N){
    int lixo;
    if (N<1){
        return;
    }
    printf("numero = %d\n", N);
    lixo++;
    imprime(N-1);
}

int main(void){

    imprime(7);
}

```