

Computer Systems Architecture

Laboratory - Part 0x01

Bogdan Macovei
Ruxandra Bălucea
Cristian Rusu

November 2022

Contents

1	The stack	2
2	Procedures in assembly language	4
2.1	Calling a procedure	4
2.2	Procedure arguments	5
2.3	Creating the call frame	7
2.4	Callee-saved and caller-saved registers	9
2.5	Returning values	11
3	Calling C functions	12
3.1	Calling the <code>printf</code> function	12
3.2	Calling the <code>scanf</code> function	14
3.3	An example of reading and displaying an undirected graph	15
4	Manipulating floating-point numbers in x86 AT&T	19
4.1	FPU	19
4.2	SSE	20
5	Exercises	23
6	Additional resources	25

1 The stack

The stack is a dynamic region within a process that operates according to the LIFO (Last In – First Out) principle. For each function call, a memory area is allocated on the stack where the **function arguments**, **local variables** and the **return address** are stored. With each allocation, the stack grows “downward”, toward lower addresses, as shown in Figure 1.

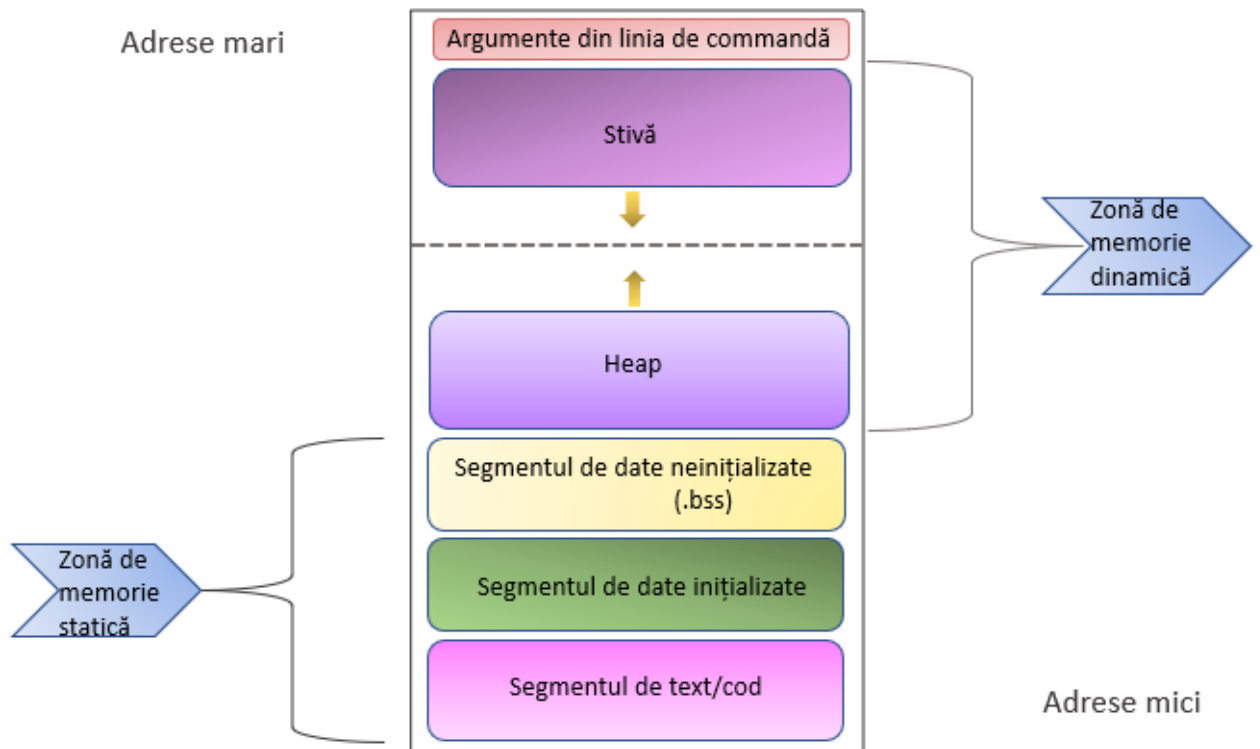


Figure 1: Address space of a process

To operate on the stack, x86 provides two instructions:

`push operand`

where **operand** can be a constant, a register, or a memory address. It is used to add a value to the stack;

`pop operand`

where **operand** can be a register or a memory address. It is used to remove values from the stack.

Note: Only **words** and **longs** can be added to the stack.

The **stack pointer** `%esp` is the register that always points to the top of the stack and through which pushes and pops are actually performed. For example, for each **pushl**, `%esp` decreases by 4 (the stack grows toward lower addresses), and a value is stored relative to the memory address indicated by `%esp`. For each **popl**, the value at the top of the stack is stored in the instruction operand, and `%esp` increases by 4.

Following the notation studied in the previous laboratory, $a(b, c, d)$, the actual value at the top of the stack is given by `0(%esp)`. In addition, note that the top of the stack will always point to the last byte of the last element added to the stack.

The **base pointer** is the register that always points to the base of the stack and is used to delimit call frames and access their corresponding elements.

An example for evaluating the expression

$$((x + y) * (x - y) * (x + z)) / (y + z)$$

using stacks is presented below. `x`, `y`, `z` are three globally declared long values.

```
.data
x: .long 2
y: .long 1
z: .long 3
e: .space 4
.text
.global main
main:

movl y, %eax
addl z, %eax
pushl %eax
; %esp: (y + z)

movl x, %eax
addl z, %eax
pushl %eax
; %esp: (x + z), (y + z)

movl x, %eax
subl y, %eax
pushl %eax
; %esp: (x - y), (x + z), (y + z)

movl x, %eax
addl y, %eax
pushl %eax
; %esp: (x + y), (x - y), (x + z), (y + z)
```

```

popl %eax
popl %ebx
mull %ebx

pushl %eax
; %esp: (x + y) * (x - y), (x + z), (y + z)

popl %eax
popl %ebx
mull %ebx

pushl %eax
; %esp: (x + y) * (x - y) * (x + z), (y + z)

popl %eax
popl %ebx
divl %ebx

pushl %eax
; %esp: ((x + y) * (x - y) * (x + z)) / (y + z)

popl e

mov $1, %eax
xor %ebx, %ebx
int $0x80

```

Note: Pay attention to the comments in the code above! As before, depending on the environment you use, comments can be introduced either with `#` or with `//`;

2 Procedures in assembly language

If an operation must be performed multiple times in the same way (possibly on different data), instead of rewriting that group of instructions several times in the program, we can encapsulate it in a procedure and, whenever we need it, call the procedure (possibly with new data passed as parameters).

In x86, procedures do not have a specific definition syntax. They are simple code blocks that are called in a specific manner. Obviously, they must be declared in the `.text` section and are identified by a name in the form of a label.

2.1 Calling a procedure

The control mechanism for jumping into a procedure and returning to the location from which it was called is implemented by two instructions – `call` and `ret`.

```
call proc
```

where **proc** is the label marking the beginning of the procedure. This instruction calls the **proc** procedure and records the return address (the location from which the procedure was called). In fact, the value in %eip (instruction pointer), which contains precisely the address of the next instruction, is saved on the stack. At the same time, %eip receives the address of the first instruction in the procedure, so execution continues from that point.

```
ret
```

This instruction jumps to the address at the top of the stack. Since, during the called procedure, all elements added to the stack are also removed, at the end the top of the stack will contain exactly the return address saved by the **call** instruction.

A simplified sketch is shown below.

```
.data
.text

proc1:

    ; equal number of pushes and pops

ret ; the stack looks as it did upon entering the procedure,
    ; so the address of the mov 1, eax instruction is at the top
    ; jump to the address at the top

.global main
main:

    call proc1 ; the address of mov 1, eax is added to the stack
                ; jump to proc

    mov $1, %eax
    xor %ebx, %ebx
    int $0x80
```

2.2 Procedure arguments

Next, to pass arguments to a procedure, they are also stored on the stack. Because a stack implements the LIFO (Last In, First Out) concept, the arguments must be stored on the stack in reverse order so that they can be accessed in their natural order. In the general case, consider the following call **proc(arg1, arg2, ..., argn)**. The call is sketched below.

```
pushl argn
pushl argn-1
...
pushl arg2
pushl arg1
call proc
popl %eax
```

```

popl %eax
...
popl %eax
; n pops corresponding to the n arguments

```

Note: All arguments loaded onto the stack must be removed when exiting the procedure.

Example: Consider now the **add** function with declaration **add(x,y,&s)**, which adds two integers **x** and **y** and stores the sum in **s**. At this point, a minimal program that calls this function with the arguments **2**, **3** and, respectively, the address of a variable in memory is shown below.

```

.data
x: .long 2
y: .long 3
s: .space 4
.text

add:
    movl 4(%esp), %eax
    addl 8(%esp), %eax
    movl 12(%esp), %ebx
    movl %eax, 0(%ebx)

    ret

.global main
main:

    pushl $s
    pushl y
    pushl x
    call add
    popl %edx
    popl %edx
    popl %edx

    mov $1, %eax
    xor %ebx, %ebx
    int $0x80

```

Requirement: Run the program with the debugger and observe the evolution of the stack. Explain the access offsets and how a parameter is passed by reference.

Note: The access offsets increase, with the argument order being the one described in the statement.

2.3 Creating the call frame

In the previous example, the `%esp` register was used to access elements on the stack. However, this approach can quickly become inconvenient if several **push** and **pop** instructions are executed during the called procedure. After each such operation, the programmer would have to reconsider the stack layout and change the access method. In practice, the same element might be accessed once as `0(%esp)` and, after another value is added to the stack, as `4(%esp)`. Since we want the way we refer to values on the stack to remain constant, we will use the `%ebp` register, which we know stores the base of the stack.

It was also mentioned earlier that, for each function call, a memory area is allocated on the stack and cleared when the function exits. Such an area is called a **stack frame** and delimits the region belonging to a procedure. For the current procedure, the top of the stack frame is represented by the value of `%esp`. We would also like to mark the base of this area, and the name immediately suggests `%ebp`. Ideally, while executing the current procedure, `%ebp` should point to the base of its own stack memory area rather than to the base of the entire stack, which would not be useful because we are not interested in all the **stack frames** created previously by nested calls.

Therefore, whenever a call frame is created, the `%ebp` register is modified to point to the base of the current **stack frame**. However, this is not sufficient because, upon returning from the procedure, we want `%ebp` to point to the base of the caller's **stack frame**. An obvious solution is to save its initial value when entering the procedure and restore it when exiting, so that `%ebp` always contains the value corresponding to the appropriate frame at any point during execution.

Thus, a correct call frame for the scheme in Subsection 2.1 is the following:

```
.data
.text

proc1:
    pushl %ebp
    movl %esp, %ebp

    ; code in the procedure body

    popl %ebp
    ret

.global main
main:

    call proc1

    mov $1, %eax
    xor %ebx, %ebx
    int $0x80
```

Therefore, we will rewrite the example from Subsection 2.2 using the new concepts. First, let us see what the stack looks like.

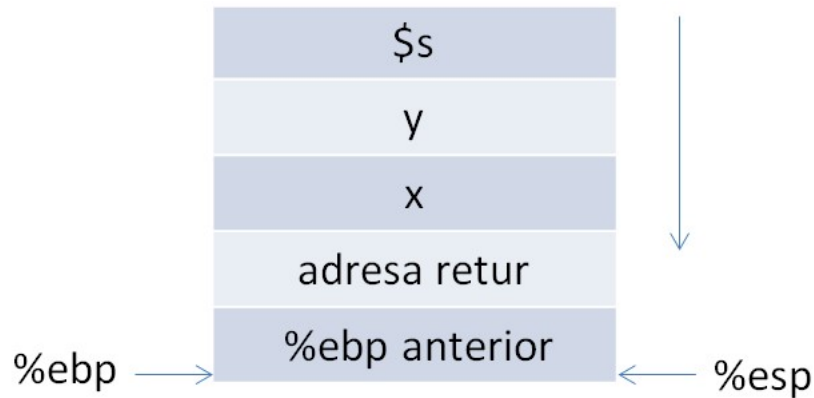


Figure 2: Program stack for Subsection 2.2

In this context, inside the **add** function we can add as many elements as we want downward on the stack because **%ebp** remains fixed; only **%esp** changes its value. Thus, the program is rewritten as in the example below.

Example:

```
.data
x: .long 2
y: .long 3
s: .space 4
.text

add:
    pushl %ebp
    mov %esp, %ebp

    movl 8(%ebp), %eax
    addl 12(%ebp), %eax
    movl 16(%ebp), %ebx
    movl %eax, 0(%ebx)

    popl %ebp
    ret

.global main
main:

    pushl $s
    pushl y
    pushl x

    call add
```

```

popl %edx
popl %edx
popl %edx

mov $1, %eax
xor %ebx, %ebx
int $0x80

```

Requirement: Run the program with the debugger and observe again the evolution of the stack, as well as the new access offsets.

Note: Local variables in procedures are also stored on the stack after the frame described above.

2.4 Callee-saved and caller-saved registers

As we know, the number of registers in the x86 architecture is limited. Until now, we have somewhat loosely assumed, by analogy with C, that registers can be viewed as program variables. Following the same analogy, consider the case in which we have a variable **x** declared in **main** and a variable with the same name in a called procedure. It is well known that the two variables do not affect one another. Modifying the variable **x** in the procedure has no effect on the one in **main**. Similarly, registers modified in a procedure must not produce side effects on their use by the caller. After returning from the procedure, we want all registers in use to contain the values stored before the call.

Again, our main tool for this purpose is the stack. Depending on where register values are saved, there are two types of registers:

- **callee-saved** : **%ebx, %esp, %ebp, %esi, %edi**. The values of these registers are guaranteed to be restored by the procedure, meaning that they must be saved by the procedure in the local-variable area.
- **caller-saved** : **%eax, %ecx, %edx**. Their restoration is not guaranteed. The caller must save these values before loading the function arguments if it wants to recover the initial values after the function returns.

Finally, a call frame will therefore look as shown in Figure 3

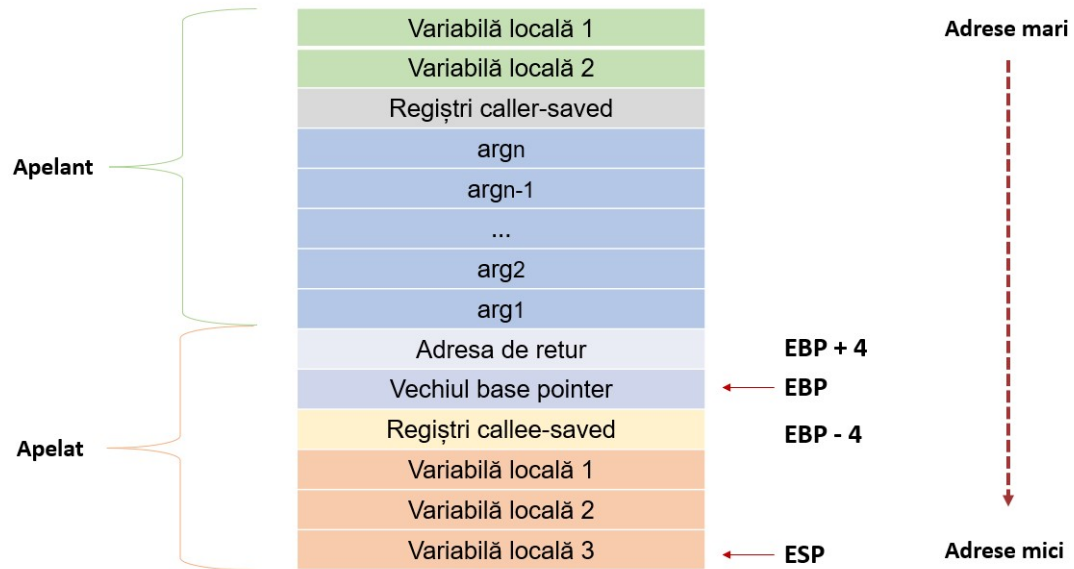


Figure 3: Call frame

Example: Let us take the same code for calling the **add** function and assume that, after the call, there is an additional piece of code that expects to find in the **%eax** and **%ebx** registers the values they had before the call. A rewritten version is shown below.

```
.data
x: .long 2
y: .long 3
s: .space 4
.text

add:
    pushl %ebp
    mov %esp, %ebp

    pushl %ebx

    movl 8(%ebp), %eax
    addl 12(%ebp), %eax
    movl 16(%ebp), %ebx
    movl %eax, 0(%ebx)
```

```

    popl %ebx

    popl %ebp
    ret

.global main
main:

    movl $5, %eax
    movl $6, %ebx

    pushl %eax
    pushl $s
    pushl y
    pushl x
    call add

    popl %edx
    popl %edx
    popl %edx
    popl %eax

    eticheta:

    ; additional code

    mov $1, %eax
    xor %ebx, %ebx
    int $0x80

```

Requirement: Run the program with the debugger and observe again the evolution of the stack, as well as the values of the `%eax` and `%ebx` registers before the additional code begins.

2.5 Returning values

Until now, we have considered only procedures that did not return any value. When values must be returned, they are stored, in this order depending on their number, in `%eax`, `%ecx`, `%edx`, and subsequently through the top of the stack.

Note: The registers used for returning values are exactly the registers that are not restored by the callee. However, they too can be restored at a certain point in the program if their initial values were needed by the caller. Therefore, returned values must be used (or stored) before such an instruction restores the old value.

Requirement: Modify the previous program so that the `add` procedure has only two parameters and returns the sum to the main program. The sum will then be loaded into memory at the address of `s`.

3 Calling C functions

To interact, from a program written in assembly language, with functions written in C, we can use the gcc compiler instead of the *linking* approach presented in the first laboratory material. In this way, we can use functions predefined in the standard library, including those mentioned earlier, namely `printf`, `scanf`, and `fflush`.

3.1 Calling the printf function

In C, the `printf` function is defined as taking a variable number of arguments, the first being the format of the `string` to be displayed. For example, the following calls are valid:

```
printf("The number to display is: %ld\n", x);
printf("The number to display is: %d\n", x);
printf("The sum of %hu and %hu is %hu\n", x, y, sum);
printf("The name of the student with the highest average is %s\n", nume);
```

Two of the useful formats that we will use are

- `%d` - for displaying ints;
- `%ld` - for displaying longs;
- `%hu` - for displaying short ints (words);
- `%s` - for displaying character strings.

Note! C longs and ints on 32 bits are equivalent to a long in assembly (they occupy 4 bytes)

The number of arguments taken by `printf` is determined by how many special formats we use in the first argument – for example, because the string for displaying the sum in the previous example contains three occurrences of `%ld`, the number of arguments is equal to 1 (the format string) + one argument for each value that must be filled in (3) = 4.

To call the `printf` function from a program written in assembly language, we proceed as follows (assume that the following file is named `call_printf.s`):

```
.data
    x: .long 23
    formatString: .asciz "The number to display is: %ld"
.text

.global main

main:
    ; load the arguments in the reverse order from C
    ; for printf("The number to display is: %ld", x);
    ; load x, then the format string
    pushl x
    pushl $formatString
```

```

; call the printf function
call printf

; for each push performed, perform a pop
; choosing a register that we are not using
popl %ebx
popl %ebx

movl $1, %eax
xorl %ebx, %ebx
int $0x80

```

Important notes:

- because `printf` accesses the arguments in their natural order – first the format of the string to be displayed, then the arguments, in the order in which they appear in the format – we must load them onto the stack in **reverse order**;
- the string format is **not** passed as a value, but **as a memory address**; therefore, its name must be prefixed with the `$` symbol;
- to perform a `pop`, we may use any register that does not currently contain a value we want to preserve – `pop` changes the access of the `%esp` register (the *stack pointer*) to the stack and loads the current value at the *top* into the register we specify. In this case, we chose to load it into `%ebx`; any other register would be equivalent;
- the `xorl %ebx, %ebx` instruction is equivalent to `movl $0, %ebx`.

To compile the program, we will use the `gcc` compiler:

```

gcc -m32 call_printf.s -o call_printf
./call_printf

```

Most likely, at this point no actual value will be displayed on **standard output**, because of the way operating systems are implemented. To force the current memory *buffer* to be displayed, we will use the `fflush` function with the `NULL` argument. The call is very simple: we must load `NULL`, i.e. the value 0, onto the stack and remove it again into a register that does not currently contain a useful value – assume again `%ebx`. The call is:

```

pushl $0
call fflush
popl %ebx

```

At this point, the complete example has the following form:

```

.data
    x: .long 23
    formatString: .asciz "The number to display is: %ld"
.text

.global main

```

```

main:
    pushl x
    pushl $formatString
    call printf
    popl %ebx
    popl %ebx

    pushl $0
    call fflush
    popl %ebx

    movl $1, %eax
    xorl %ebx, %ebx
    int $0x80

```

We compile again:

```

gcc -m32 call_printf.s -o call_printf
./call_printf

```

which, this time, correctly displays the message

The number to display is: 23

Note. To standardize the way we make *calls* to `printf`, we will always use a `printf` followed by an `fflush`.

3.2 Calling the `scanf` function

We start again from the way this function is called in C, namely:

```

scanf("%ld", &x);
scanf("%ld %ld", &x, &y);

```

Although the call is very similar to the one used for `printf`, note that this time the arguments are no longer passed by their actual value – we do not pass `x`, but `&x`. This must also be reflected in assembly language. For this purpose, we use the `$` symbol, which gives us access to the memory location of a variable declared in the `.data` section.

We demonstrate this using the code below, assumed to be saved in a file named `call_scanf.s`, for reading an integer from the keyboard.

```

.data
    x: .space 4 ; we do not yet know the value that will be read, but we know how much space it oc
    formatString: .asciz "%ld"
.text

.global main

main:

```

```

; load NOT x onto the stack, but the ADDRESS of x
; that is, not x, but &x
pushl $x
; load the address of the format string
pushl $formatString
call scanf
; remove from the stack what we loaded
popl %ebx
popl %ebx

movl $1, %eax
xorl %ebx, %ebx
int $0x80

```

3.3 An example of reading and displaying an undirected graph

To demonstrate the use of `printf`, `scanf`, and `fflush`, we present the reading of an undirected graph, specified by its number of vertices, number of edges, and its edges.

```

.data
    matrix: .space 1600
    columnIndex: .space 4
    lineIndex: .space 4
    n: .space 4
    nrMuchii: .space 4
    index: .space 4
    left: .space 4
    right: .space 4
    formatScanf: .asciz "%ld"
    formatPrintf: .asciz "%ld "
    newLine: .asciz "\n"

.text

.global main

main:
    pushl $n
    pushl $formatScanf
    call scanf
    popl %ebx
    popl %ebx

    pushl $nrMuchii
    pushl $formatScanf
    call scanf
    popl %ebx
    popl %ebx

```

```

; in fact, we simulate
; for (long index = 0; index < nrMuchii; index++)
; {
;     scanf("%ld", &left);
;     scanf("%ld", &right);
;     matrix[left][right] = 1;
;     matrix[right][left] = 1;
; }
    movl $0, index
et_for:
    movl index, %ecx
    cmp %ecx, nrMuchii
    je et_afis_matr

    pushl $left
    pushl $formatScanf
    call scanf
    popl %ebx
    popl %ebx

    pushl $right
    pushl $formatScanf
    call scanf
    popl %ebx
    popl %ebx

; we need to set
; matrix[left][right] = 1
; the index is left * n + right

    movl left, %eax
    movl $0, %edx
    mull n
; %eax := left * n
    addl right, %eax
; %eax := left * n + right

    lea matrix, %edi
    movl $1, (%edi, %eax, 4)

; we need to set matrix[right][left] = 1
; the index is right * n + left

    movl right, %eax
    movl $0, %edx
    mull n
; %eax := right * n
    addl left, %eax

```

```

; %eax := right * n + left

lea matrix, %edi
movl $1, (%edi, %eax, 4)

incl index
jmp et_for

et_afis_matr:
movl $0, lineIndex
for_lines:
    movl lineIndex, %ecx
    cmp %ecx, n
    je et_exit

    movl $0, columnIndex
    for_columns:
        movl columnIndex, %ecx
        cmp %ecx, n
        je cont

        ; display matrix[lineIndex][columnIndex]
        ; the index is lineIndex * n + columnIndex
        movl lineIndex, %eax
        movl $0, %edx
        mull n
        addl columnIndex, %eax
        ; %eax = lineIndex * n + columnIndex

        lea matrix, %edi
        movl (%edi, %eax, 4), %ebx

        pushl %ebx
        pushl $formatPrintf
        call printf
        popl %ebx
        popl %ebx

        pushl $0
        call fflush
        popl %ebx

        incl columnIndex
        jmp for_columns

cont:
    movl $4, %eax
    movl $1, %ebx

```

```

        movl $newLine, %ecx
        movl $2, %edx
        int $0x80

        incl lineIndex
        jmp for_lines

et_exit:
        movl $1, %eax
        movl $0, %ebx
        int $0x80

```

Note: Test your knowledge so far by taking laboratory tests 6.1, 6.2, and 6.3, which you can find online in the laboratory section.

4 Manipulating floating-point numbers in x86 AT&T

The x86 architecture supports operations on floating-point numbers using two approaches, **FPU** (*Floating Point Unit*) and **SSE** (*Streaming SIMD Extensions*). Historically, FPU was introduced first, and was later extended by the introduction of the SSE convention.

Floating-point numbers are represented according to the *IEEE 754* standard, and in this laboratory we will work only with the 32-bit `float` format (the architecture also supports a `double` format for greater precision, represented on 64 bits). In the `float` format, there is 1 sign bit, 8 exponent bits, and 23 mantissa bits, and a value is encoded according to the formula $\text{Val} := (-1)^S \cdot 2^{E-\text{Bias}} \cdot (1.M)$.

4.1 FPU

FPU was initially implemented as a hardware coprocessor (for example, the 8087 coprocessor for the 8086 processor) and was later integrated directly into processors. It uses an internal stack (stack-based) for operations. We can think of these locations as registers from `st(0)` to `st(7)`, organized as a stack, with **FPU** instructions operating implicitly on this stack.

With respect to the approaches and the registers used, for **FPU** we have registers from `st(0)` to `st(7)`, organized as a stack, on which **FPU** instructions operate implicitly, while for **SSE** we have registers from `xmm0` to `xmm7`.

Below, we present several instructions that can be used to manipulate floating-point numbers.

Instruction	Operands	Brief explanation
<code>flds</code>	<code>mem</code>	Loads a <code>float</code> onto the FPU stack, into <code>st(0)</code>
<code>fldl</code>	<code>mem</code>	Loads a <code>double</code> onto the FPU stack, into <code>st(0)</code>
<code>fstps</code>	<code>mem</code>	Stores a <code>float</code> from <code>st(0)</code> into memory
<code>fstpl</code>	<code>mem</code>	Stores a <code>double</code> from <code>st(0)</code> into memory

We also have mathematical functions available in `libm`, a library that can be specified in the compilation command using the `-lm` flag (for example, `gcc -m32 float.s -o float -no-pie -lm`). Examples of functions we can use include `logf`, `sqrtf`, and `expf`. The calling convention is respected: arguments are loaded onto the stack in reverse order, and the function result is obtained from `st(0)`. These functions are precisely why the FPU is still used, as they are implemented according to this older convention. The FPU is therefore retained for compatibility with libraries implemented according to this standard.

Example for computing the natural logarithm:

```
.data
    number: .float 2.718281828
    logResult: .space 4
.text

.global main

main:
    flds number    // load the value onto the FPU stack
    subl $4, %esp
```

```

    fstps 0(%esp)    // perform a push to respect the calling convention

    call logf

    fstps logResult // we received the result in st(0); pop it and store it in logResult
    addl $4, %esp    // pop from the stack

et_exit:
    movl $1, %eax
    xorl %ebx, %ebx
    int $0x80

```

We run this program (from a file named, for example, `float1.s`) with `gcc -m32 float1.s -o float1 -no-pie -lm`. We use `gdb` to check whether the computation was performed correctly: `b et_exit, run, print (float) logResult`. We observe that the value obtained is 0.99999994.

4.2 SSE

SSE was introduced as a modern extension to address FPU limitations and provide improved performance in certain scenarios. It is intended for operations on vector data (e.g. graphics, audio, video, game physics), allowing the processor to execute the same operations on multiple data values simultaneously.

Vector registers are used (for example, 128-bit `xmm` registers), which can operate on multiple values simultaneously (such as four 32-bit float values at the same time). In this laboratory, we will omit these vector extensions and use the registers from `xmm0` to `xmm7` only to hold a single scalar value.

Again, the calling convention is respected: arguments are loaded onto the stack in reverse order, and this time the function result is obtained from `xmm0`. Newly added procedures (including your own) can follow this newer convention.

For **SSE**, we will consider the following instructions:

Instruction	Operands	Brief explanation
<code>movss</code>	<code>src, dest</code>	Moves a scalar float from src to dest
<code>movsd</code>	<code>src, dest</code>	Moves a scalar double from src to dest
<code>addss</code>	<code>src, dest</code>	Adds the float in src to dest and stores the result in dest
<code>subss</code>	<code>src, dest</code>	Subtracts src from dest and stores the result in dest
<code>mulss</code>	<code>src, dest</code>	Multiplies src by dest and stores the result in dest
<code>divss</code>	<code>src, dest</code>	Divides dest by src and stores the result in dest
<code>sqrtps</code>	<code>src, dest</code>	Computes dest := sqrt(src)
<code>cvtss2ss</code>	<code>src, dest</code>	Converts a long to float and stores the result in dest
<code>cvtss2sd</code>	<code>src, dest</code>	Converts a float to double and stores the result in dest
<code>maxss</code>	<code>src, dest</code>	Computes the maximum in dest
<code>minss</code>	<code>src, dest</code>	Computes the minimum in dest

Although this laboratory will mainly use **float** numbers, the previous table also presented a few instructions for **double**. These are useful when displaying values because, although according to the

conventions the format string "%f" should expect a `float` argument, the implementation of `printf` always requires a `double` argument.

Next, we present a code sequence for displaying the `logResult` value computed in the previous example:

```
.data
    number: .float 2.718281828
    logResult: .space 4
    fs: .asciz "%f\n"
.text
    ...
et_printf:
    cvtss2sd logResult, %xmm0    // convert to double
    sub $12, %esp               // 8 bytes for double + 4 for the format string
    movsd %xmm0, 4(%esp)        // place the double on the stack
    movl $fs, 0(%esp)           // place the address of the format string on the stack
    call printf                  // call printf
    add $12, %esp               // remove the values from the stack
```

We also provide an example for computing the arithmetic mean of the elements of an array of floats:

```
.data
    v: .float 1.0, 2.0, 3.0, 4.0
    n: .long 4
    result: .space 4
    mem: .space 4
.text

.global main

main:
    lea v, %edi                 // address of the array
    xorl %ecx, %ecx             // start from index 0
    movl $0, mem                // use intermediate memory
    movss mem, %xmm0            // initialize the sum to 0

et_loop:
    cmpl n, %ecx
    je avg

    movss (%edi,%ecx,4), %xmm1  // load the array element
    addss %xmm1, %xmm0          // add it to the sum

    incl %ecx
    jmp et_loop
```

```

avg:
    cvtsi2ss %ecx, %xmm1        // convert the number of elements to float
    divss %xmm1, %xmm0          // divide and store the result in xmm0

    movss %xmm0, result

et_exit:
    movl $1, %eax
    xorl %ebx, %ebx
    int $0x80

```

To inspect the value, assume the file is named `float2.s` and run `gcc -m32 float2.s -o float2 -no-pie -lm`; then, in `gdb`, use `b et_exit, run, print (float) result`. We observe that the value obtained is 2.5.

Exercise. Assume that you have an *array* of floats storing the probabilities $\{p_i\}_{i \in \{1, \dots, n\}}$ of n events. Determine the **entropy** of this set. Implement this problem using a procedure with the signature `float entropy(float *probabilities, float n)`.

5 Exercises

- Define the procedure `perfect(x)`, where x is a natural number. A number is perfect if it is equal to the sum of its divisors up to half of the number. Example: $6 = 1 + 2 + 3$; $28 = 1 + 2 + 4 + 7 + 14$;
 - Read an integer `n` and a vector with `n` elements from the keyboard. Display the number of perfect elements.

- Implement an x86 procedure with the same effect as the C procedure `memcpy`:

```
void *memcpy(void *dest, const void *src, size_t n);
```

which copies `n` bytes from address `src` to address `dest` and returns `dest`.

- Write a procedure that implements the C function `atoi`:

```
int atoi(const char *s);
```

which returns the integer whose external decimal representation is contained in the string `s`, or 0 if the string cannot be converted.

- Translate the following C program into x86 assembly language:

```
#include <stdarg.h>

void aduna(long *a, long n, ...){
    register long i;
    va_list l;
    va_start(l,n);
    *a=0;
    for(i=0;i<n;++i) *a+=va_arg(l, long);
    va_end(l);
}

long s, s1;

void main(){
    aduna(&s,3,1,2,3);    /* we obtain s=6 */
    aduna(&s1,2,10,20);   /* we obtain s1=30 */
}
```

- Implement a program that computes the function $f(x) = 2g(x)$, where $g(x) = x + 1$.
- Implement the procedure `proc(x)`, $x > 1$, with the definition:

```
void proc(long x) {
    printf("\%ld ",x);
    if (x != 0)
        proc(x-1);
}
```

- Translate the following into x86 assembly language:

```

long aduna(long a, long b) {
    return a+b;
}

void iteratie(long *a, long *b) {
    long c;
    c=aduna(*a, *b);
    (*a)=(*b); (*b)=c;
}

void main() {
    long n=5,x=1,y=1,z;
    register long i;
    for(i=2;i<n;++i) iteratie(&x,&y);
    z=y;
}

```

8. Implement a procedure that recursively computes the factorial of a number.
9. Write a program that computes the n -th term (t_n) of the Fibonacci sequence (using recursion). At the end, display a text of the form *The n -th element of the Fibonacci sequence is x .* We consider the first two elements to be $t_0 = 0$ and $t_1 = 1$. Example: $n = 5 \Rightarrow$ The 5-th term of the Fibonacci sequence is 5.
10. Read n and k ($k \leq n$) from the keyboard. Write a recursive function $C(n, k)$ that computes combinations of n taken k at a time according to the formula:

$$C(n, k) = \begin{cases} 1 & \text{if } k = 0 \\ 1 & \text{if } k = n \\ C(n, k) = C(n-1, k) + C(n-1, k-1) & \text{otherwise.} \end{cases}$$

6 Additional resources

For more details about GNU Assembly, we recommend Richard Blum's book *Professional Assembly Language*, especially the following chapters:

- Chapter 7: Using numbers, the SIMD Integers and Floating-Point Numbers subsections;
- Chapter 9: Advanced Math Functions;
- Chapter 11: Using Functions;
- Chapter 12: Using Linux System Calls;
- Chapter 13: Using Inline Assembly;
- Chapter 15: Optimizing Routines;
- Chapter 17: Using Advanced IA-32 Features.