

Basic elements of the assembly language

The Intel processor registers

Any processor contains a large number of registers, making use of them for various internal actions. These are of no interest for the programmer. On the other hand there is a special category, namely the general-purpose registers, which can be used explicitly in the programs, similarly to memory variables. Below are listed the general-purpose registers of the processors from the 32-bit Intel x86 family:

	31	15	8	7	0	
EAX		AH		AL		AX
EBX		BH		BL		BX
ECX		CH		CL		CX
EDX		DH		DL		DX
ESI						SI
EDI						DI
EBP						BP
ESP						SP

The general-purpose registers can be used for storing both data (all of them) and memory addresses (only the 32-bit registers).

Attention: the registers in the table above are not completely independent. For example, the register EAX contains AX, which means that any change of the value in AX will automatically alter the value in EAX, and the other way around; the same is true for the 1-byte registers.

Writing the addresses

Syntactically, an address must be enclosed between square brackets. The addressing modes describe how an address can be expressed.

a) The address is a constant value

Example: [100]

b) The address is expressed through a 32-bit general-purpose register

Example: [EAX]

c) The address is expressed as the sum between the value of a 32-bit general-purpose register and a constant value

Example: [EBX+5]

d) The sum between two 32-bit general-purpose registers

Example: [ECX+ESI]

e) The combination between the two previous modes: the sum between two 32-bit general-purpose registers and a constant

Example: [EDX+EBP+14]

f) The sum between two 32-bit general-purpose registers, one of which is multiplied by 2, 4, or 8, to which a constant can be added

Examples: [EAX+EDI*2], [ECX+EDX*4+5]

All 32-bit general-purpose registers can be used equally for addressing, without restrictions. Of course, high level languages use variables for accessing data; accordingly, in assembly language we have to provide the corresponding addresses.

In theory, the first addressing mode seems sufficient. In practice, if we consider the problem of going through the elements of an array, we already see that it is not always possible to work only with constant addresses.

Flags

Beside the result itself, the execution of an instruction may also have additional effects. For example, in the case of addition, it is of interest to know whether the result written into the destination is correct or not; we remind that it may not be possible to represent the result with the number of bits of the destination operand, in which case we get a wrong value.

For such situations, the processor has a series of flags. The flags are bits which are set or reset by the processor, depending on the result yielded by the current instruction, in order to show the occurrence of possible special situations. Their values can be subsequently tested by software, so that the programmer can adapt the execution flow.

The most widely used flags are:

- *Carry*: shows whether, for an addition, the result requires more bits than the destination operand has
- *Overflow*: shows whether, for an addition with signed numbers, the result in the destination is wrong, due to the occurrence of an overflow
- *Zero*: shows whether, for an addition/subtraction, the result has value zero
- *Sign*: shows the sign of the result of an addition/subtraction

The flags are very important, because these are the only conditions that may be tested in assembly language. Consequently, as we will later see, all control structures in the high-level languages can be implemented only by testing those flags.

Assembly language instructions in Visual C++

The purpose of this laboratory is not to learn how to write code exclusively in assembly language, but to write parts of assembly code inserted into C (or C++) programs.

To insert assembly code in a C program, the `_asm` keyword is used:

```
_asm {  
... //comments are the same as in C or C++  
}
```

Assembly instructions can be separated by ";" (as in C) or by newline. For the sake of clarity, it is recommended to write one instruction per line.

Arithmetic instructions

The assignment instruction

```
mov destination,source
```

The operands can be constant values, registers, or memory addresses (corresponding to C variables). The types of the two operands can be combined as in the examples below:

```
mov eax,ebx
mov cx,5
mov bl,[eax]
mov [esi],edx
```

The main rule, for virtually all instructions in assembly language, is that the operands must have the same size.

A special case occurs when one operand is a memory address, while the other is a constant. In the examples above, at least one of the operands is a register, which automatically enforces the size of the other operand. In this case, both the memory address and the constant can require either 1, 2, or 4 bytes. For that reason, the programmer must indicate the size of the operands:

```
mov byte ptr [ecx],5 //1-byte operands
mov word ptr [ecx],5 //2-byte operands
mov dword ptr [ecx],5 //4-byte operands
```

It is not possible for both operands to be memory addresses:

```
mov byte ptr [eax],[ebx] //wrong, the compiler will signal an error
```

Everything that was said here about the operand types and sizes stands true for all other instructions.

The addresses of the variables declared in a C program are not known to the programmer. In order to provide the assembly language with a way of accessing the variables, Visual C++ allows the use of symbolic names. In other words, if a C program contains the following statement:

```
int a;
```

we can write in an assembly block an instruction as below:

```
mov a,3
```

In this case, the compiler will generate an assignment instruction that can be understood and executed by the processor, by replacing the name of variable `a` with its address. Moreover, we notice that in this case it is not necessary to indicate the operand size, although we have a memory address and a constant, because the compiler can see the declaration of variable `a` (the type `int` is 4-byte wide). However, if we wish to access only the least significant byte of `a`, we can write:

```
mov byte ptr a,9
```

On the other hand, the following code sequence is incorrect:

```
int a=8,b=9;
_asm {
mov a,b
}
```

As both operands are memory addresses, the compiler cannot generate a valid processor instruction to do the transfer directly. We have to write instead:

```
_asm {
mov eax,b
mov a,eax
}
```

Test

Write and execute the following program:

```
#include <stdio.h>
void main() {
int i=10;
_asm {
mov i,5
}
printf("%d\n",i);
}
```

By looking at the value displayed for variable `i`, we can confirm that `mov` has the effect described above. Also, imagine some tests for the error case that may occur, in order to see the reaction of the compiler.

Addition

```
add op1,op2
```

Effect: `op1 += op2;`

Operand combinations - as for the assignment:

```
add eax,ebx
```

```
add dl,3
```

```
add si,[...]
```

```
add [...],ebp
```

```
add byte ptr [...],14
```

```
add word ptr [...],14
```

```
add dword ptr [...],14
```

Again, it is not possible for both operands to be memory addresses:

```
add byte ptr [...],[...] // error
```

In the examples above, as in the ones that will come, by `[...]` we denote a memory address operand; any addressing mode can be used.

The sum of two numbers is computed correctly by `add`, whether the numbers are signed or unsigned. In other words, it does not matter if the operands correspond to the C type `int` or `unsigned`.

Addition may yield an incorrect result, if the numbers are too big. For unsigned numbers, the `carry` flag is then set. For signed numbers, the relevant flag is `overflow`.

Test

Write and execute the following program:

```
#include <stdio.h>
void main()
{
    int a=10;
    _asm {
        add a,5
    }
    printf("%d\n",a);
}
```

A case that must be analyzed is the computation of the sum of two variables into a third variable:

```
#include <stdio.h>
void main()
{
    int a=10,b=5,c;
    _asm {
        mov eax,a
        add eax,b
        mov c,eax
    }
    printf("%d\n",c);
}
```

Regarding the flags, we can perform tests on types `int` and `unsigned`:

```
#include <stdio.h>
void main()
{
    int i1=1000000000,i2=2000000000;
    printf("%d\n",i1+i2); // to show that the result is also wrong in C, not a programming error
    _asm {
        mov eax,i2
        add i1,eax
    }
    printf("%d\n",i1);
}
```

```

}
#include <stdio.h>
void main()
{
unsigned i1=1000000000,i2=2000000000;
printf("%u\n",i1+i2); // now the result is correct
i1=3000000000;
printf("%u\n",i1+i2); // now it is wrong
_asm {
    mov eax,i2
    add i1,eax
}
printf("%u\n",i1);
}

```

We remark here that the processor always updates both `carry` and `overflow` (in fact all flags), whether we work with signed or unsigned numbers, because it does not know in which case we are. It is then the programmer's task to test the appropriate flag.

Subtraction

```
sub op1, op2
```

Effect: $op1 -= op2$;

We emphasize the relevance of flag `zero`, which in this case shows the equality of the two initial operands. This flag is also updated by addition, but there its practical usefulness is reduced.

Test

Write and execute step-by-step the following program:

```

#include <stdio.h>
void main()
{
int a=10,b=5;
_asm {
    mov eax,b
    sub a,eax
}
printf("%d\n",a);
_asm {
    mov eax,b
    sub a,eax
}
printf("%d\n",a);
}

```

The goal is to monitor the value of flag `zero` after the two subtractions.

Observation: in the second `_asm` block we cannot assume the value of register `eax` has remained the same as it was at the end of the first block, because there was a call to function `printf` in-between; thus, we cannot discard the second `mov` instruction.

Increment and decrement

```
inc op
dec op
```

Effect:
 $op++$
 $op--$

These instructions have been provided with the single purpose of simulating the addition and subtraction of numbers whose sizes are wider than the processor word. The operand combinations are the same as for `add` and `sub`, and so are the flags.

Naturally, these instructions may be easily replaced by `add` and `sub`, respectively.

Multiplication

`mul op`

Effect: the explicit operand is multiplied by an implicit register, and the result is written to a destination which is also implicit; both the implicit register and the destination depend on the size of the explicit operand. The general rule is that the two operands (explicit and implicit) must have the same size, while the destination must be double in size. More precisely:

explicit operand size	implicit operand	destination
1	<code>al</code>	<code>ax</code>
2	<code>ax</code>	<code>(dx, ax)</code>
4	<code>eax</code>	<code>(edx, eax)</code>

If the operands are 4-bytes wide, the result requires 8 bytes. As there are no registers that big, the register pair `(edx, eax)` is used, where register `edx` contains the most significant part; in other words, the result is equal to $edx \cdot 2^{32} + eax$. For 2-byte operands, although the result would fit into a 4-byte register, the destination is the pair `(dx, ax)`, in order to preserve the compatibility with older processors.

The explicit operand can be a register or a memory location, but not a constant value:

```
mul ebx // eax·ebx→(edx, eax)
mul cx
mul al // compute the square of al
mul byte ptr [...] // eax·[...]→(edx, eax)
mul word ptr [...]
mul dword ptr [...]
mul byte ptr 10 // error
```

When the explicit operand is a memory location, we also must explicitly indicate its size; otherwise, the implicit operand and the destination could not be determined.

Instruction `mul` performs the multiplication of unsigned numbers. For multiplying signed numbers, instruction `imul` is used. This instruction has the same syntax, the same operand types, and the same behavior, but the operands are considered signed numbers and the result is computed accordingly. Unlike addition and subtraction, multiplication and division require different computing algorithms for signed and unsigned numbers, so there are different instructions.

Test

Compute the factorial of an unsigned number. For comparison, the program will be written in C first.

```
#include <stdio.h>
void main()
{
    unsigned n, i, f=1;
    for(i=1; i<=n; i++)
        _asm {
            mov eax, f
            mul i
            mov f, eax
        }
    printf("%u\n", f);
}
```

In C there is no way of getting an 8-byte result when we multiply 4-byte numbers (which is the case for type `unsigned`). On the one hand there is no integer type of such size; on the other hand, the compiler requires for the result to have the same type as the operands. That restriction is also present in the program above, so register `edx` can be ignored.

Division

`div op`

Effect: analogous to instruction `mul`. The explicit operand is the divisor. The dividend is implicit and depends on the size of the divisor, just like the destinations where the quotient and the remainder are written:

divisor size	dividend	quotient	remainder
1	<code>ax</code>	<code>al</code>	<code>ah</code>
2	<code>(dx, ax)</code>	<code>ax</code>	<code>dx</code>

4	(edx, eax)	eax	edx
---	------------	-----	-----

As we can see, in all cases, the quotient is written in the least significant half of the dividend, while the remainder is written in the most significant half. Storing the results in this manner allows repeating the operation in a loop, if necessary, without requiring any additional transfer operations.

By analogy with multiplication, the explicit operand (the divisor) can be a register or a memory location, but not a constant:

```
div ebx
div cx
div dh
div byte ptr [...]
div word ptr [...]
div dword ptr [...]
div byte ptr 10 //error
```

Just as in the case of multiplication, for signed numbers there is instruction `idiv`.

The division operation raises a particular problem: division by 0. For testing reasons, we can write a program as below:

```
#include <stdio.h>
void main()
{
    _asm {
        mov eax,1
        mov edx,1
        mov ebx,0
        div ebx
    }
}
```

The program will signal an runtime error and will be finished forcefully.

Then, we change the program, such that the value assigned to `ebx` is 2, and we resume the execution; the results by itself is not important, it is enough to notice that there is no error any more.

We change the program again, such that `ebx` is assigned the value 1. We get a new runtime error. The cause is that, according to the rules shown above, the quotient must be stored in `eax`, but it is too big to fit. In such a case, the solution is to switch to larger-sized operands, which, unfortunately, is not always possible. All we can do is to detect beforehand when such an error would occur and skip the division operation. This way, the program will no longer be finished forcefully; the way it further handles the situation depends on its goal - and on the programmer.

In order to see how we can determine the occurrence of the error only by looking at the operand values (without performing the division), we consider the case of the program above. The condition for the error to occur is:

$$\frac{edx \cdot 2^{32} + eax}{ebx} \geq 2^{32} \Leftrightarrow edx \cdot 2^{32} + eax \geq ebx \cdot 2^{32} \Leftrightarrow eax \geq (ebx - edx) \cdot 2^{32}$$

As $eax < 2^{32}$ and the values of the registers are natural numbers, the only way to meet the condition above (i.e., to cause an error) is to have $ebx \leq edx$. More generally, we get an error if and only if the most significant half of the dividend is greater than or equal to the divisor.

A straightforward consequence is that the following instructions will always cause an error:

```
div edx
div dx
div ah
```

A last test program:

```
#include <stdio.h>
void main()
{
    unsigned a=500007,b=10,c,d;
    c=a/b;
    d=a%b;
    printf("%u %u\n",c,d);
}
```

For the program above, the division operations must be written in assembly language.

Observations:

- only one division instruction needs to be written in assembly language
- take care with register initialization
- there is a tendency to ignore the initialization of register `edx`; this may lead not only to an incorrect result, but also, as seen before, to the program being finished forcefully (in this case, if the value of `edx` is at least 10)

Bitwise instructions

Boole instructions

The Boole operations implemented by Intel processors are AND, OR, XOR (binary) and NOT (unary). These are also the names of the instructions, while the operands are (just as for assignment or addition) memory variables, registers, constants. As for the previous instructions, the first operand is also the destination where the result is stored:

```
not  eax // complement each bit in eax
and  bx,16 // AND between the bits on the same position in the two operands, for all positions
or   byte ptr [...],100
xor  [...],ecx
```

There are the same limitations regarding the way the operands can be combined: it is not possible for the to differ in size operands or to be both memory addresses. Of course, these restrictions apply to binary operations. For example, the instructions below are incorrect:

```
and word ptr [...],[...]
or  dx,eax
```

The equivalent operators in C are \sim , $\&$, $|$, \wedge ; actually, as the first operand is also the destination, it would be correct to say the operators are $\sim=$, $\&=$, $|=$, $\wedge=$.

As in C, these instructions are rarely used. They are most useful when we work with masks.

Let us consider that we need to test the bit in position 3 from register `ax`. Given what we have learned so far, we could consider using the processor's `Zero` flag; unfortunately, the flag provides information about the whole operand, not just one of its bits. The only choice we have is to alter the register `ax` such that it has the value 0 if and only if the bit in position 3 is 0. To do that, we have to force the value 0 on all bits in `ax`, except for the bit in position 3, which will keep its initial value (that one we want to test). The Boolean function that is capable of performing such processing (forcing to 0 or keeping the previous value) is AND. Thus, we have to perform the AND operation between `ax` and an operand built to have value 1 on position 3 and 0 on all other bits:

```
and ax,8
```

<code>ax</code>	B_{15}	B_{14}	B_{13}	B_{12}	B_{11}	B_{10}	B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
8	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
<code>ax</code>	0	0	0	0	0	0	0	0	0	0	0	0	B_3	0	0	0

In conclusion: if $B_3=0$, in the end we will have `ax=0`; if $B_3=1$, `ax \neq 0`. In both cases the flag `Zero`, which provides information about `ax`, will in fact show the value of bit B_3 in `ax`.

The constant value which serves as the second operand for AND is called a mask; indeed, we see that its task is to "filter" the bits in the position we are not interested in, by forcing a neutral value into them. For the example above, if we wish to test the bit in position i , the mask must have the value 2^i .

Of course, the other Boole functions can be used as well. Also, a mask may leave unchanged more than one bit. Moreover, we can use as a mask a register or a variable (instead of the constant), such that the mask itself may change during the execution; it all depends on what we wish to achieve. Nevertheless, the most widely used operation is AND. Once we got here, we make the observation that, in fact, the instruction above is not really a test, because it changes the value of the operand it works with. No matter how much we need the value of the bit in position 3, we do not wish to lose the values of the other bits. Obviously, there are solutions to that; however, the designers of the processor have decided to solve the problem by providing a special instruction, called `test`. Thus, instead of the instruction above, we can write:

```
test ax,8
```

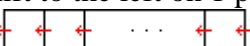
The Boole function that is applied here is still AND, but the value of register `ax` is not changed - in fact, the result is not stored anywhere. On the other hand, the flag `Zero` is still updated in the same way as for instruction `and`. If we wish to test and not process the register `ax`, this is exactly what we need.

There are no similar instructions for the other Boolean functions; this shows, once again, the practical importance of the function AND.

Shift instructions

These are instructions that change the positions of the bits within the operands.

The shift to the left on 1 position has the effect depicted below:

Carry 

A similar action is taken for the shift to the right.

The shift instructions are:

```
shl eax,1 // shift register eax to the left by 1 position
```

```
shl byte ptr [...],3 // it can also be done by multiple positions
```

```
shl dx,c1 /* the number of positions by which the shift is performed can be specified through c1 -  
the only register that can be used for such purpose (even cx or ecx cannot be used) */
```

```
shr bh,1 // shift to the right
```

Of course, if the shift is performed on multiple positions, the flag Carry will store the last bit that got out.

The equivalent C operators are << and >>, respectively (or, more precisely, <=< and >>=).

The shift operations are mostly used for arithmetic purposes. It is well known that a shift to the left on n positions is equivalent to multiplication by 2^n , while a shift to the right represents a division by 2^n . However, using shift operations in order to substitute multiplication and division raises a problem: how to proceed when working with signed numbers? Let us consider the shift to the right. If the most significant bit initially has value 1, and the shift instruction changes it to 0, the sign of the number has changed, which is arithmetically incorrect. For that reason, the processor also provides another shift to the right instruction, especially designed to work on signed numbers:

```
sar eax,1
```

Its effect is mostly the same with the effect of the instruction `shr`, except that the most significant bit keeps its initial value (while being also passed to the right). It can be proven that, with such an operation, we get the correct result of the division by 2^n . In conclusion, we have two shift to the right instructions; one is useful for unsigned data types (`shr`), while the other works for signed types (`sar`).

For the shift to the left, unfortunately, there is no similar solution. The assembly language accepts the instruction `sal`, but this is only another name for `shl`. It is easy to see that we can only work on the bit that "enters" from the outside, which in this case is the least significant, so it has no effect on the sign. The only solution here is through software, by writing a more complex sequence of instructions.

Test

Write and execute the following code sequence:

```
unsigned u;
```

```
for(u=1;u!=0;u<=1)
```

```
    printf("%u\n",u);
```

Then change the code as below:

```
unsigned u=1;
```

```
while(u!=0) {
```

```
    printf("%u\n",u);
```

```
    _asm {
```

```
        shl u,1
```

```
    }
```

```
}
```

We shall notice that, in both cases, the behavior is identical.

Jump instructions

From a practical point of view, a jump instruction changes the value of the program counter; as a result, the next instruction to be executed will no longer be the one following the current instruction in memory. The usefulness of these instructions is clear: any control structure (either test or loop) can be implemented at processor level only through jumps.

Unconditional jump

```
jmp address
```

Although there are also other ways to express the jump address, in the sequel we will only consider the case where the address is a constant value. Obviously, when we write the code, we cannot possibly know the jump address; we have to leave the address management to the compiler, by using labels, just as for the instruction `goto` in C. Visual C++ is quite flexible in this regard: it is possible to jump to labels defined in the same `_asm` block, in another block, or in a region of C code. It is also possible the other way around: a `goto` instruction may jump to a label defined in an `_asm` block.

Test

```
int i;
_asm {
    mov i,16
    jmp et
    mov i,5
et:
    add i,3
}
printf("%d\n",i);
```

When we run the program, we notice that variable `i` ends up with value 19, so the addition was executed, but the second assignment was not.

The unconditional jump by itself is not very useful, because it does not really introduce a branch in the program - we do not have multiple execution paths.

Conditional jumps

In this case we have two possible execution paths:

- the jump condition is true - make the jump to the address given as an operand
- the jump condition is false - continue with the next instruction in memory, as if there was no jump instruction at all

The simplest conditional jump instructions are the ones that test the individual flags. In the sequel we will only consider the most commonly used flags: Carry, Overflow, Zero, and Sign.

For each flag there are two conditional jump instructions: one that jumps when the tested flag has value 1 and one that jumps when the tested flag has value 0.

tested flag	jump on value 1	jump on value 0
Carry	<code>jc</code>	<code>jnc</code>
Overflow	<code>jo</code>	<code>jno</code>
Zero	<code>jz</code>	<code>jnz</code>
Sign	<code>js</code>	<code>jns</code>

Of course, for each of the instructions above, the jump address must be passed as an operand.

In reality, these individual flags are not very useful. As we know, in C, the tests are mainly based on relational operators (`<`, `<=`, `==`, `!=`, `>`, `>=`). With some exceptions, such relations between the operands cannot be determined by testing a single flag; more complex instructions are required.

Let us consider first the comparison instruction:

```
cmp op1, op2
```

The possible combinations of operands are the same as for the previously studied instructions. What we have inside this instruction is a subtraction, which does not change the operands (it does not store the result anywhere); instead, the flags are still updated. This means that we could also use the instruction `sub` for the same purpose, if changing the first operand is not a problem. The most important thing here is the fact that, by analyzing all flags (not just one), it is possible to decide which was the relation between the operands of the comparison instruction. Because it would be too difficult for the programmer to write several conditional jumps

for a single test, the designers of the Intel processors have introduced a set of conditional jumps that perform the corresponding tests:

relation	jump instruction
op1 < op2	jnb
op1 <= op2	jbe
op1 > op2	ja
op1 >= op2	jae
op1 == op2	je
op1 != op2	jne

Problem: the relations between the operands depend on the types of the operands (signed/unsigned). For example, let us consider the following operands:

01001011

10011010

If the operands are unsigned numbers, obviously, the second operand is bigger. If, however, the operands are signed numbers, the first operand is positive, while the second is negative. On the other hand, the comparison instruction is always the same. So, the instructions above are meant for unsigned operands. A different set of instructions has been designed for signed operands:

relation	jump instruction
op1 < op2	jnl
op1 <= op2	jle
op1 > op2	jg
op1 >= op2	jge
op1 == op2	je
op1 != op2	jne

We notice that (as expected) the equality and non-equality are identical for the two sets of instructions.

Example:

Given a variable `a` (unsigned integer), we want to count the bits of value 1. For the sake of simplicity, we consider that we can afford to alter the value of `a`. A possible solution (written entirely in C) could be the following:

```
nr=0;
for(i=0;i<32;i++) {
    if(a&1) nr++;
    a>>=1;
}
printf("%u\n",nr);
```

Now we possess all the knowledge needed to write the code above in assembly language, except for the display operation. For start, we can keep working with memory variables (then, as an exercise, we can consider replacing some of them by registers):

```
_asm {
    mov nr,0
    mov i,0
e1:
    cmp i,32
    jae e3
    test a,1
    jz e2
    add nr,1
e2:
    shr a,1
    jmp e1
e3:
}
printf("%u\n",nr);
```

Control structures

In the sequel we will study the implementation of the C-specific control structures in assembly language.

The if structure

For an `if` structure without the `else` branch, a single jump instruction is sufficient. In assembly language, the jump is made on the opposite condition, compared to the corresponding C code. The reason for this reversal is simple: in C, a condition that is true leads to the execution of a set of instructions; in assembly language, a condition that is true leads to making a jump, that is, avoiding the execution of some instructions.

As an example, let us consider the following C code:

```
if (x>5)
    x--;
```

According to what has already been said, this translates in assembly language as below:

```
cmp x, 5
jle nu
dec x
nu:
```

Observation: Variable `x` is considered to be of type `int` (signed integer), for which reason we used the instruction `jle` instead of `jbe`.

When there is also an `else` branch, two jump instructions must be employed. The second one is an unconditional jump; its role is to make sure that, when the instructions in the first branch have been executed, they will not be followed by the instructions in the second branch.

C code:

```
if (x>5)
    x--;
else
    x++;
```

Assembly code:

```
cmp x, 5
jle nu
dec x      // the branch executed when x>5
jmp _out   // jump over (avoid) the second branch
nu:
inc x      // the branch executed when x≤5
_out:
```

The while structure

A `while` structure also requires the use of two jump instructions: one for deciding whether a new iteration is to be carried out or the loop is terminated; the second, at the end of the loop body, for starting a new iteration. As in the case of the `if` structure, the first jump is made on the opposite condition (compare to the C code), while the second jump is unconditional.

C code:

```
while (x<10)
    x++;
```

Assembly code:

```
_loop:
cmp x, 5
jge _out
inc x      // loop body
jmp _loop
_out:
```

The for structure

The `for` structure is semantically similar to the `while` structure. Consequently, its implementation in assembly language is also similar. The only difference consists in the auxiliary instructions, which separate more clearly both the initialization step and the updating of the variables for the next iteration. Thus, the order of those instructions must be preserved:

1. initialization (executed only once)
2. test - either continue with a new iteration or exit the loop
3. loop body

4. update the variables for the next iteration
5. jump to step 2

The do...while structure

The `do...while` structure resembles the `while` structure, but the test for resume/exit is executed after to loop body. It is the only iterative structure for which the jump condition tested in assembly language is the same as in C (and not the opposite).

C code:

```
do
    x++;
while(x<10);
```

Assembly code:

```
_loop:
inc x // loop body
cmp x,10
jl _loop
```

Working with the stack and function calls

Stack-specific instructions

As a stack-like structure is necessary in many situations, the processor also uses a part of the RAM memory for LIFO accesses. As we know, the only information that is vital for the stack management is the top of the stack (the stack pointer). In this case, the address of the top of the stack is stored in the register ESP.

The instruction responsible for inserting data into the stack is called, obviously, `push` and has a single operand. Examples:

```
push eax
push dx
push dword ptr [...]
push word ptr [...]
push dword ptr 5
push word ptr 14
```

We notice that the stack works only with 2-byte or 4-byte values. In practice it is recommended to work exclusively with 4-byte operands; the 2-byte syntax is maintained to preserve the compatibility with older processors.

What happens when such an instruction is executed? The processor writes the value of the operand at the address pointed by ESP (the stack pointer), after decreasing the value of ESP by the size of the operand (2 or 4); this way, the stack pointer is ready for the next writing operation. In other words, the instruction `push eax` would be equivalent to:

```
sub esp, 4
mov [esp], eax
```

It is better, however, to leave the processor do these all these actions (and thus reduce to risk of making mistakes).

The opposite instruction, which removes data from the stack, is called `pop`. As expected, it also works only with 2-byte or 4-byte operands:

```
pop eax
pop cx
pop dword ptr [...]
pop word ptr [...]
```

When this instruction is executed, operand points to the destination where the data retrieved from the top of the stack (ESP) will be stored; then, the value of ESP is increased by the size of the operand.

The processor stack is used to store temporary information. For example, if we need to use a register for certain operations, but all registers already hold information that we cannot afford to lose, we can proceed as below:

```
push eax
... // use eax
pop eax
```

This way, we could both use register `eax` for a while and then restore its initial value.

Local variables are also stored on the stack. We remind that a local variable is created upon the call of the function where it is declared, and then disposed of upon the termination of the same function, so its temporary nature is straightforward.

Clearly, working with the stack requires a lot of attention; the `push` instructions must be carefully compensated by the `pop` instructions - this is about both the number of such instructions and the sizes of the operands. Any mistake will usually alter more data than we would think. For example, let us consider the following sequence:

```
push eax
push edx
mov eax, [esi]
mov edx, 5
mul edx
mov [esi], eax
pop eax
```

The multiplication by 5 of the variable is done correctly. Then, because a `pop` instruction is missing, the values of both registers involved are affected: `eax` gets another value than initially, while the value of `edx` remains the one received from the multiplication, without the older one being restored. It is just as bad if there are too many `pop` instructions. In most cases, the operand inserted into the stack through a `push` instructions is also the destination of corresponding `pop` instruction (yet there are always exceptions, depending on the nature of the program).

Another error may occur when register `ESP` is directly manipulated. For example, in order to allocate room for a local variable, it is enough to decrease the value of `ESP` with size of that variable. Similarly, when the variable is disposed of, the value of `ESP` is increased. In such cases we usually do not make use of `push` and `pop` instructions, as we are interested not in the values involved, but only in allocating and releasing the memory space. We prefer to increase and decrease directly the register `ESP`; obviously, an error with these operations has the same consequences as described above.

Function calls

At first sight, a function call looks like a jump instruction, because it breaks the linear execution of the program and jumps to an entirely different address. There is, however, a fundamental difference: when the function terminates, the execution is resumed from the point the call was made. Since a function can be called multiple times, from multiple execution points, and the execution is always resumed from the right point, it is clear that the return address is stored and used when necessary. The return address is thus temporary information, so it belongs on the stack.

A function call is performed through the instruction `call`, which has the following syntax:

```
call address
```

As before, in Visual C++ we will use symbolic names to indicate the call address. This times, the symbolic names are not labels, but the names of the called functions.

The effect of the instruction `call`: insert in the stack the address of the next instruction in the program (return address) and jump to the address pointed to by the operand. These actions could also be done through `push` and `jmp` instructions, but again, the processor does all the work alone and thus avoids the possible errors.

The return from a function is done by the instruction `ret`, which reads the return address from the top of the stack (similar to a `pop` instruction) and jumps to that address.

It is now clear that an error in working with the stack has even harsher consequences than we saw previously. If we omit an instruction `pop` (or we write one too much), when an instruction `ret` is executed, it will read from the stack not the correct return add, but something entirely different. In practice, this always leads to the program being jammed or forcefully finished - and we remind that there are always function calls in a program (at least to `main`).

Parameters

The function parameters are also local variables, so they are stored on the stack. The caller has the responsibility to place them on the stack upon the call and remove them from the stack upon return from the called function. Of course, the parameters are placed on the stack by using the instruction `push`, right before the call. Upon return, the parameters must be removed from the stack, as they are no longer necessary. In this particular case we are not interested in reading their values, so the instruction `pop` is not used here; instead, `ESP` is increased by the total number of bytes of the parameters (we remind the recommendation of working with 4-byte values, even when then operands have smaller sizes).

As an example, let us consider the following function:

```
void dif(int a,int b)
{
    int c;
    c=a-b;
    printf("%d\n",c);
}
```

The call `dif(9,5)` is translated by the sequence below:

```
push dword ptr 9
push dword ptr 5
call dif
add esp,8
```


In practice, the parameters can be accessed either by their names, as they are mentioned in the function's header (the simpler way), or by their addresses (less intuitive, but sometimes cannot be avoided). In the second case, in order to express the addresses of the parameters, we use the register EBP:

- The first (leftmost) parameter in the function header is always found at address EBP+8.
- The subsequent parameters, moving to the right in the parameter list, are found at higher addresses: EBP+12, EBP+16, etc.

Returning values

How can a function return a value? The convention used by Visual C++ (and by most other compilers) is to store the result in a certain register, depending on its size:

- for 1-byte data types - in `al`
- for 2-byte data types - in `ax`
- for 4-byte data types - in `eax`
- for 8-byte data types - in the pair `(edx, eax)`

Upon function return, the caller has to read the result from the appropriate register.

Saving the register values

The programs we write are C/C++ programs, so we cannot know how the compiler makes use of the registers. Thus, if the function we write in assembly language changes some of the registers, these must be saved in the stack at the beginning of the function and restored in the end. The only exception is represented by the registers `eax` and `edx`, which, as seen before, are used for returning the function value. Also, when we write assembly code in Visual C++, we never explicitly write the instruction `ret`. The reason is that, when the function is terminating, there are also other actions to be performed, of which we are not aware. The compiler takes care of all these actions, including the instruction `ret`.

Exercises

- Write in assembly language the function `dif`, whose C implementation was discussed above. The variable `c` will no longer be necessary, as we can use registers. On the other hand, the call to `printf` will remain written in C, as we do not currently possess the knowledge required for writing it in assembly language.
- Change the function, such that it will return the difference it computes, instead of displaying it. Upon return the result must be read from `eax` and stored into a variable, because in C we cannot call `printf` with `eax` as a parameter.
- Write in assembly language a recursive function which computes the factorial of a positive integer, passed as a parameter.

Arrays and structures

Arrays and pointers

Let us compute the sum of the elements of an array of integer numbers:

```
int t[]={4,9,12,3,7};
int i,sum=0;
for(i=0;i<5;i++)
    sum+=t[i];
```

All concepts required for translating the code above have already been discussed, except for the way of accessing the elements of an array in assembly language. The first observation is that, obviously, the address of such an element can be determined from two pieces of information: the start address of the array and the position (index) of the element within the array.

For the first component we can use directly the name of the variable, in this case `t`. The compiler always replaces the names of the variables with their addresses in the memory, so that general mechanism is applied here too.

Regarding the index, as it is not possible to use a memory variable (in this case `i`) to access another variable, we have to use a register instead. The code below shows the implementation in assembly language:

```
mov ebx,0 // used instead the variable sum
mov eax,0 // used instead the variable i
_loop:
    cmp eax,5
    jge _out
    add ebx,t[eax*4]
    inc eax
    jmp _loop
_out:
```

We see an important detail: while in the C language the index is expressed as the number of elements with respect to the beginning of the array, in assembly language the offset is expressed in bytes. For this reason, although register `eax` has the same values as variable `i`, in expressing the address of each element its value must be multiplied by 4 (because the elements of the array are of type `int`, whose size is 4 bytes).

Observation: Instead of the syntax `t[eax*4]` we can use the equivalent notation `t+eax*4`, which is more suggestive in some contexts.

Let us now consider a pointer variable, which is initialized with the start address of array `t`:

```
int *p=t;
```

If we rewrite the C code which computes the sum of the elements, this time using variable `p` instead of `t`, the only change is that we will have to write `p[i]` instead of `t[i]`. With the assembly language, however, things are different. Variable `p` cannot be used to express the start address of the array. Instead, its value must be copied into a register, which will be used as a start address:

```
mov ebx,0
mov eax,0
mov ecx,p // get the start address of the array
_loop:
    cmp eax,5
    jge _out
    add ebx,[ecx+eax*4]
    inc eax
    jmp _loop
_out:
```

Passing arrays as function parameters

In C/C++, arrays are never passed directly as parameters to the functions. In order to save memory, the start address of the array is always passed as a parameter, instead of the whole array. Consequently, the following function headers are interchangeable:

```
void f(int x[]);
void f(int *x);
```

In both cases, parameter `x` is a pointer containing the start address of the array. Also, function `f`, with either of the two headers, may receive as a parameter both an array (in which case its start address will be placed on the stack) or a pointer. Using the variables introduced above, the following calls are both valid and yield the same result:

```
f(t);  
f(p);
```

2D arrays and pointers

Let us consider a 2D matrix of size 3×3. This can be allocated either statically (when the arrays is declared) or dynamically. The two cases are depicted below through variables `t` and `p`, respectively:

```
int t[3][3];  
int **p;  
p=new int*[3];  
for(i=0;i<3;i++)  
    p[i]=new int[3];
```

Let us also consider a function which accepts a matrix as a parameter. Similar to the single-dimensional case, there are two syntactic forms of writing the function header:

```
void f(int x[][3]);  
void f(int **x);
```

This time, however, the two forms are not interchangeable. The former will accept only array `t` as a parameter, while the latter accepts only pointer `p`. The reason is that, unlike the single-dimensional case, the ways of accessing the matrix elements are very different in the two situations.

a) In the case of array `t`, while it is declared as a 2D in C, its elements are stored linearly in the memory. Beginning with the start address of the array, we have in order elements `t[0][0]`, `t[0][1]`, `t[0][2]`, `t[1][0]`, etc. In conclusion, in order to determine the address of an element `t[i][j]`, its offset with respect to the beginning of the array is $i \cdot 3 + j$, where 3 is the number of columns of the matrix. We note that, in the case of the first form of function `f`, parameter `x` represents the start address of the matrix, just as for single-dimensional arrays.

b) In the second case, access to element `p[i][j]` is made in two steps. In the first step, the value of pointer `p` is used to read element `p[i]`. In the second step, the value of `p[i]` (which is also a pointer, containing the start address of an array with 3 elements of type `int`) is used for accessing `p[i][j]`.

It is now clear that, for example, an array like `t`, whose elements must be accessed in the way described at point a, cannot be passed as a parameter to the second form of function `f`, which assumes that the elements are accessed in the manner described at point b.

Structures

Structures have one thing in common with arrays: they are all composite types, made up of elements placed into memory at consecutive locations. Unlike an arrays, however, a structure contains fields with different types and thus different sizes; consequently, the fields cannot be accessed based on an index, but only based on the offset within the structure.

Let us consider the example below:

```
struct S {  
    char a,b;  
    int c;  
};
```

At first sight, a variable of type `S` occupies 6 bytes, and fields `a`, `b`, and `c` are placed into memory at offsets 0, 1, and 2, respectively. In practice, most compilers, including Visual C++, use a technique named address alignment, which requires that any field start at an offset which is a multiple of its size. In our example, fields `a` and `b`, which are of type `char` and thus are each 1 byte wide, are still placed at offsets 0 and 1, respectively; on the other hand, field `c`, which is of type `int` (4 bytes wide), is placed at offset 4, so the structure as a whole occupies 8 bytes. Obviously, between fields `b` and `c` there are 2 unused bytes.

Let us consider the structure below, which is used for manipulating points in the plane:

```
struct point {  
    int x;  
    int y;  
};
```

Given two variables `p1` and `p2` of type `point`, we attempt to write the assembly code which determines the middle of the segment between the two points (`p1` and `p2`) and writes it into variable `p1`.

We first make the observation that, within the `point` structure, field `x` is placed at offset 0 (and thus it starts at the same address as the structure as a whole), while field `y` is placed at offset 4. The code mainly consists in the execution of two addition operations, corresponding to fields `x` and `y` of the two variables:

```
mov eax,dword ptr p2
add dword ptr p1,eax
mov eax,dword ptr p2+4
add dword ptr p1+4,eax
```

In this case we have to use the prefix `dword ptr` to indicate that the operands to be added have 4 bytes. Otherwise, when using the names `p1` and `p2`, the compiler will see that they correspond to 8-bytes variables, so that will signal an error (because the other operands are 4-byte registers).

Alternately, it is possible to use the syntax `[. . .]` to indicate the offset of each field within the structure:

```
mov eax,dword ptr p2
add dword ptr p1,eax
mov eax,dword ptr p2[4]
add dword ptr p1[4],eax
```

Exercise

Rewrite the code above in order to implement a function which accepts as parameters the addresses of two structures of type `point`:

```
void middle(point *p1,point *p2);
```

The function writes the result in the structure whose address is indicated by the first operand.

The parameters are pointers because the C language (unlike C++) does not allow passing directly structures as parameters, only their start addresses.