On the assembler

Our today's topic will be the basics of **assembler**, i.e. a very low-level programming language in which single instructions represent single instructions of machine code in a form more readable for the human eye. (On the other hand, this language is still way less intutive than e.g. C++ or Java).

The goal of this lecture is *not* to teach programming in the assembler, as this topic is way too broad for here. We will, however, look at a few snippets of assmebler code and understand what they do. Such passive knowledge is even more important given that snippets of assmebler code can be found in the "most low-level" among the popular programming languages (e.g. C). Therefore, even those who are not strictly assembler programmers may have a need to understand such code.

Introductory remarks

The x86 architecture

The diversity of existing processor architectures leads to diversity of assmeblers. Here, we will focus on the — clearly most popular — **x86 architecture**. For this one, programmers can choose among a few variants of assmebler which differ by various language aspects; in the examples below, we will use the syntax of a popular open-source assembler, **NASM**.

Before describing the language itself, let us discuss what is the *x86 family* of processors and why it is so important.

The first model from this family, Intel 8086 from year 1978, was used in the first generally available personal computers. Comparing to earlier processor models, it included multiple innovations (more registers, changes in memory management, increased efficiency). All of this made Intel 8086 a very important model. This, in turn, created a demand for subsequent processor models to have a **backwards compatible** architecture, i.e. that their instruction set should support all programs which used to work on Intel 8086. Thanks to this, the assmeblers for the x86 architecture generate programs which can run on various processors. This involves not only Intel models, but also those from other manufacturers, including the second important market player, AMD. However, not everything is unified. For example, the sets of available registers may differ across x86 processors, which also leads to some differences in supported assmebler instructions. Moreover, even the operating system (Windows/Linux/another) can influence certain assembler instructions. Still, the general philosophy of the language and most of the instructions remain common.

Distinctive language properties

The structure of an assembler program has much in common with programs known from more popular languages: basically, a program is a sequence of **instructions**, often taking **arguments** which can be e.g. constants, registers (the simplest counterpart of variables), or more complex memory references (somewhat analogous to arrays, pointers etc.).

On the other hand, the assembler syntax does not contain some constructs which are fundamental for higher-level languages (while, in assembler, the necessary functionality must be "built piece by piece" from instructions with more simplistic meaning). Two examples of such "painful omissions" are:

- an almost complete lack of arithmetic expressions — instead of writing e.g. `a * (b + c)`, we will have to achieve the necessary value through individual arithmetic *instructions*;
- lack of basic flow control instructions like `if`, `for`, `while` — in assembler, the necessary flow control is achieved with various *jump* instructions (including *conditional* and *unconditional* ones).

Simple examples

An arithmetic expression

```
mov  eax, 1
mov  ebx, 2
mov  ecx, 3
imul ebx, ecx
sub  eax, ebx
imul eax, eax
```

The above code contains registers `eax`, `ebx`, `ecx`, and the following instructions:

- `mov` X Y: stores into register X the value described by Y (e.g. a numeric constant, or the value of another register);
- `sub` X Y: subtracts from the value of X the value described by Y, and stores the result back into X;
- `imul` X Y: mutliples the values of X and Y, and stores the result back into X.

Generally, the convention in NASM syntax is an arithmetic operation stores its result in the register given as its first argument. (Caution — in some other assmeblers, that role is played by the last argument).

This means that the above code corresponds to the following code in Java (or C++):

```
int a = 1;
int b = 2;
int c = 3;
b = b * c;
a = a - b;
a = a * a;
```

in which the variables `a`, `b`, `c` correspond, respectively, to registers `eax`, `ebx`, `ecx`. (For clarity: as a result of running that code, the final value of variable `a`, or register `eax`, will be `(1-2*3)*(1-2*3)`, which is 25).

Notably, while in languages like Java/C++ the programmer can use nearly arbitrary variable names (of course assuming they have been properly declared), in assembler the available registers are restricted to a fixed set (which we already described in the lecture "Processor and programs"). If these registers do not suffice to hold all necessary data (which happens very often), the data must be somehow stored in the RAM memory, outside registers — of which some examples will be shown later.

A loop

Loops in assembler must be built using conditional jump instructions. Here is an example (explained below):

```
     mov eax, 0
     mov ecx, 10
start_of_the_loop:
     add eax, ecx
     dec ecx
     cmp ecx, 0
     jne start_of_the_loop
```

This contains the following new instructions:

- `add`: adds two registers; stores the result in the first of them;
- `dec`: decreases the value of the given register by 1;
- `cmp`: compares the two given values; stores the information about results in the appropriate flag registers:
  - in ZF: 1, if the compared values were equal; 0, if they were different;
  - in CF: 1, if the first value was lower than the second one; 0 otherwise.
- `jne` X: ***j**ump if **n**ot **e**qual*: makes a conditional jump to instruction X (here described with the **label** `start_of_loop`), on the condition that the previously executed comparison instruction resulted in "not equal" (which can be checked via the value of the flag ZF).

This means that the above code corresponds to the following code in Java (or C++):

```
int sum = 0;
int i = 10;
do {
  sum += i;
  i--;
} while (i != 0);
```

in which the variable `sum` corresponds to the register `eax`, and `i` corresponds to `ecx`.

A conditional instruction

Now, consider the following code in Java (or C++):

```
if (eax > ebx)
  ecx = ebx;
else
  ecx = eax - ebx;
```

Although the assembler does not contain instructions like `if`, still, similarly as for loops, such behavior can be obtained with a conditional jump instruction. Below, we show a sample "translation" of this code to the assembler language:

```
    cmp eax,  ebx
    jle another_case
    mov ecx,  ebx
    jmp end_of_this
another_case:
    mov ecx,  eax
    sub ecx,  ebx
end_of_this:
```

The new instructions used here have the following meaning:

- `jle` X: ***j**ump if **l**ess than or **e**qual*: a conditional jump to instruction X, on the condition that one of the flags CF, ZF contains value 1 (which means: in the previously executed comparison, the first value was lower than or equal to the second one);
- `jmp` X: an unconditional jump to instruction X.

Subroutines

A **subroutine** (also called: *subprogram*, *procedure*, *function*) is a piece of program code which may be **invoked** many times during the execution of a program, and moreover — importantly — these invocations may happen **from various locations** in the code, and yet, after each invocation, the execution will *return* to the place from which the invocation was made.

Thus, a *non-example* of a subroutine is the content of a `while` loop (even though it can be executed many times). *Examples* of subroutines are functions in higher-level programming languages (which, in Java and other object oriented-languages, are often called *methods*).

Similarly as for loops and conditional branches, there is no assembler syntax for defining subroutines straightforwardly; instead, the necessary functionality can be essentially built of simpler elements: uncodintional **jumps** to an instruction specified by its address, and existence of an instruction pointer register (`eip`). However, since the x86 architecture does not offer a direct programmer access to register `eip`, instead one has to use special instructions which hide the details from the programmer:

- `call` X: make a **subroutine call** to the label X, which means:
  - jump (unconditionally) to X
  - also, store in a special place (specifically — on the *stack* which will be described later in this lecture) the **return address**, i.e. the address to which the execution should jump back, after executing the subroutine is finished. (It is the address of the instruction *directly following* the `call` instruction).
- `ret`: make the return jump, that is: jump (unconditionally) to the most recently stored return address. (Also, clean up the just-used information about the return address).

Let us consider an example. Suppose that we have some code resulting in printing out the value of `ecx` to the screen. Then, the following code will print, in order, the values 1, 2, 2, 3:

```
 1      jmp start_there
 2
 3    our_printer:
 4      // some code (details omitted)
 5      // printing out the value of ecx
 6      // to the screen
 7      ret
 8
 9    our_double_printer:
10      call our_printer
11      call our_printer
12      ret
13
14    start_there:
15      mov  ecx, 1
16      call our_printer
17      mov  ecx, 2
18      call our_double_printer
19      mov  ecx, 3
20      call our_printer
```

Executing this code will proceed as follows:

- We run line 1, which causes a jump to line 14 (`start_there`). No subroutines touched so far.
- In line 15, we set `ecx` to 1, and then in line 16 we call `our_printer`. The return address points to the instruction which we will want to run after the subroutine ends — in this case, it is line 17. The processor stores that address (on that stack), and then jumps to the label `our_printer` (line 3).
- Lines 4, 5, 6 print out "1" on the screen.
- Line 7 jumps to the return address, which is line 17. Now, we finished our first subroutine call.
- Line 17 sets `ecx` to 2, and then line 18 calls `our_double_printer` (with return address 19). There, in turn, in line 10, we make another call to `our_printer` (with return address 11). Thus, we have here a subroutine call **inside** another subroutine! This means that we need to remember **multiple return addressess** at once: "after this subroutine, return to line 11; after *the preceding one*, return to line 19". Hence, the return addresses naturally form a **stack** structure — and that's why they must be stored in such way.
- After printing "2" for the first time, the `ret` from line 7 jumps to the return address (11), at the same time *pops* that value from the stack, to *uncover* the previously pushed return address associated with calling `our_double_printer` (19).
  Executing line 11 leads to printing "2" once again, returning to line 12 and popping the return address 12 from the stack.
- Now, line 12 (`ret`) will make the return jump to the address on the top of the stack, which is currently 19.
- Finally, lines 19 and 20 will lead to printing "3" and terminating the program.

The strength of the mechanism described above lies in its **full generality**: thanks to using a stack, we can support nearly unboundedly long chains of subroutine calls inside each other, an in particular, **recursive** calls (that is, situations in which a subroutine calls itself — either directly or indirectly).

Using RAM memory

Until now, all our examples only involved manipulating on data stored in registers. These, however, are limited in number and in total capacity. On the good side, the operating system equips every process with some amount of RAM memory (up to 4 GiB). This area can be imagined as what we'd call an array in higher-level langauges: a sequence of memory cells to which we can refer using consecutive indices (here called **addresses**).

The address space of a process

While the *physical* localization of the area of RAM granted to a particular process may vary (in particular, it does not have to be contiguous!), the machine code of the process does not need to "care" about this, because it operates on **logical addresses** which are usually between 0 and 4 GiB, and are translated by the processor's MMU to physical addresses on every memory access. (The details of this translating mechanism will be discussed on a later lecture). The layout of memory granted to a process — organized according to logical addresses — is called the **address space** of that process.

In such address space, we distinguish several areas with a specified purpose, size and access mode. (These areas are often called **segments**, although that term happens to have different interpretations in various contexts).

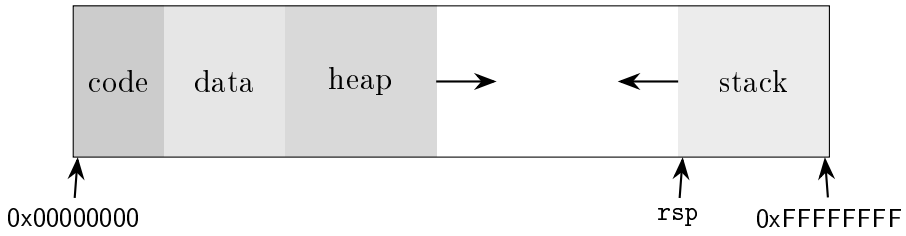

Figure 1. The typical layout of the address space of a process in the x86 architecture.
As it dates back to the times of 32-bit addressing, the space of available addresses has size 2^{32} B = 4 GiB.
The code and data segments (whose size is known upfront at the start of program execution) are located in the lowest address ranges; the stack and heap segments (which grow dynamically as the process executes) are placed on the opposite sides of the remaining free place — which leaves maximum elasticity in memory management. By convention, the heap takes the lower addresses and grows upwards, while the stack takes the highest addresses and grows downwards.

- The **code segment** contains the machine code of the process (loaded into RAM to let the processor fetch consecutive instructions efficiently). This segment is available to the process in a read-only fashion (which is enforced by the MMU); this protects programs against accidentally modifying their own code. (That is a trace of Harvard architecture; see the lecture "Computer structure").
- The **data segment** stores data (constans as well as variables) which are **statically allocated**, i.e. those data for which we want to reserve place in memory *for the whole timespan* of process execution. (In particular, this means that the size of this segment is known upfront when starting the process). For example, it will contain global variables in a C++ program, as well as *static variables* in a Java program.

- The **heap** and the **stack** provide place for the **dynamically allocated** data, i.e. those data which *turn out* to require storing only *during* program execution. The difference between them can be summarized as follows:
  - The **stack** stores data which can be organized in the so-called **LIFO** manner (*last in, first out*), that is: whenever a new value is *pushed* to the *top* of the stack, it will be *popped* (i.e. removed) from there earlier than any other values already currently present on the stack.
    Although such description might sound a bit enigmatic, in practice — due to the natural process of organizing the code into subroutines — the stack structure is fit for storing all kinds of **subroutine local data**, that is: data whose lifetime ends at the moment of exiting a subroutine.
  - The **heap** stores those dynamically allocated data whose lifetime can extend beyond the context in which they have been created. For example, in languages like Java and C++, the heap will host data (e.g. objects, arrays) created with the `new` keyword: even if a usage of `new` happens inside some function, the created object must be often stored also after returning from that function (e.g. because the function has returned a pointer or a reference to it). In such case, the moment of removing the object from the memory can be determined directly by the programmer (in C++) or by the language runtime environment (in Java); anyway, in both cases, there is no LIFO-like rule here, which makes the stack not suitable for such data.

Memory references

In NASM, the basic way of referring to RAM cells is to put their **addresses** (of course, the logical ones[1]) in **square brackets**. For example:

- `mov [`N`], ecx`: stores the value of register `ecx` under address N;
- `mov ecx, [`N`]`: fetches into register `ecx` the value from address N.

However, we must watch out for pitfalls of various kinds:

- Since the size of `ecx` is 4 bytes, the above instructions will actually operate on four memory cells (each of them one-byte), with addresses n, N+1, N+2, N+3.
- The exact value of N will be usually not known for us, at least not as a numeric constant (which is e.g. because the placement of segments in the address space is usually not known upfront). Hence, in practice, we will specify it by using available langauge identifiers (see descriptions of arrays and stack below).

Arrays

An **array** — whether in assembler or some higher-level langauges (e.g. C/C++) — is a *contiguous* area of memory, storing a sequence of values of the same type (in particular: occupying the same number of bytes). In NASM, a **byte array** can be created with the following instructions:

[1] More strictly, it may happen that the number N will not be interpreted directly as a logical address (the cell index in the process address space) but as a **relative address** to some base address (e.g. the beginning of the data segment). This, however, has no impact on our considerations in the following part of this lecture.

- `db` *val1*, *val2*, ...: creates a **read-only array** with the specified numeric values *val1*, *val2* etc. The NASM syntax also allows string constants here, e.g. `db "ABC", 10` will create an array of four bytes, with values 65, 66, 67 (ASCII codes of characters A, B, C) and 10 (which happens to be the ASCII code of the line-feed special character).
- `resb` *N*: creates an array of size *N*, whose content will be then modifiable by our code. (Such arrays are commonly called **uninitialized**, though in practice it often happens that they are initialized with all zeroes).

Whenever the instruction defining an array is preceded by a **label**, that label — used anywhere else in the code — will denote the address of the first element of that array. Moreover, the NASM syntax allows some amount of explicit arithmetic on memory addresses, so we can write for example:

- `mov ecx, [tab+4]`: store into `ecx` the value from the address "the first element of the array whose label is `tab`, plus 4 bytes".
  (If we adopt the convention that the cells of that array are numbered starting from zero, this can be expressed in short as: "store into `ecx` the value from the address of the fourth cell of the array described by `tab`". Here, however, we reach the byte-size pitfall again: as `ecx` is four-byte, this will result in actually rewriting the fourth, fifth, sixth and seventh byte of the array!)

In practice, programmers often need **arrays of multi-byte types**. For example, in Java, the type `int` has 4 bytes, so declaring `int tab[3] = {10, 1000, 100000};` will result in creating an array of total size of 12 bytes. An analogous code can be also written in the assembler:

- `dd 10, 1000, 100000`: creates a read-only array, containing three 4-byte numbers (of total size 12 B).

The ending "`d`" here comes from "*double word*", as a **word** traditionally means "2 bytes" in this context. Analogously, `dw` (`w` for *word*) declares an array of 2-byte numbers, and `dq` (`q` for *quad word*) — an array of 8-byte numbers. Similarly, the instructions `resw 10`, `resd 10`, `resq 10` will allocate a memory area of size — respectively — 20, 40 or 80 bytes.

However, it must be remembered that — unlike in higher-level languages — the unit of memory addressing (for *all* kinds of arrays) in the assembler is **always a single byte**, regardless of the "declared content type". Therefore, even if we have a multi-byte-typed declaration like

```
tab:
dd 10, 1000, 100000
```

then, still, referring to the value 100000 requires writing `[tab+8]`, *not* `[tab+2]`! Luckily, to make the code more intuitive, the language allows a more explicit notation: `[tab+4*2]`. Moreover, if we want to refer to the i-the element of the array (counting from 0), and the value of i is stored in `eax`, the language allows this: `[tab+4*eax]`.

The stack

The basic access to the stack is enabled by instructions which we know back from an earlier lecture:

- `push rax`: puts the value of `rax` on the top of the stack;

- `pop rax`: takes (and removes) the value from the top of the stack, and stores it in `rax`.

In the x86 architecture, the **stack grows downwards** (i.e. towards lower addresses): the stack base has the highest address, and pushing a value boils down to storing it in the memory *directly before* the current stack content. The **stack pointer register** `rsp` always stores the address of the current top element of the stack, and is updated by the processor on every use of the above instructions.

The above conventions allow treating the stack as an array, with the starting address stored in `rsp`. Therefore, to read the value of the top element of the stack, it suffices to write `[rsp]`. As the stack is often composed of 8-byte entries, `[rsp+8]` will often mean the "next top" element, etc.

For an assembler programmer, the stack is a convenient **temporary storage** for intermediate computation results in case when we run out of available registers. However, its **fundamental application** is storing all the relevant register and flag values in case of a **subroutine call**.

In an earlier part of this lecture, we described how the stack is used to store the "obscured" return addresses — however, an analogous problem (and a solution to it offered by the stack) involves the general-purpose registers. For example, consider the following Java code:

```
int our_function (int a, int b) {
    return a + b;
}

// ...

void start() {
    a = 2;
    b = 5;
    c = our_function(3, 4);
    return a + b + c;
}
```

In this code, the values of variables `a` and `b` in function `start` are respectively 2 and 5. However, during execution of `our_function`, these are temporarily **obscured** by respective 3 and 4 — until we exit the call to `our_function`. After returning to `start`, the previous values of `a` and `b` will be restored — thanks to the fact that the compiler was able to associate the names `a`, `b` with distinct memory addresses, depedning on which function is currently executed.

A similar problem occurs in the assmebler: here, also, a subroutine that has been just called can e.g. freely play with registers content, which often creates a need for restoring their previous values after exiting that subroutine. However, in the assmebler, names like `ebx` always refer to *the same* register, so — unless we care to "back up" the original value before calling a subroutine — that value could become prematurely and irreversibly lost. And here — just like for return addresses — the possibility of long call chains (and particularly recursion) makes the stack the only reasonable place for backing up the previous register values.

Moreover, while the abovementioned stack management of return addresses happens "automatically", as a side effect of the `call` and `ret` instructions, it is generally a **programmer's responsibility** to back up the general-purpose registers! As a result, the assembler code often contains — depending on the convention followed — whole "back-up sections" appearing either *just before* or *just after* making a subroutine call:

back-up just before the call

```
push eax
push ecx
push edx
call my_function
pop edx
pop ecx
pop eax
```

back-up just after the call

```
my_function:
push eax
push ecx
push edx
// proper subroutine code
pop edx
pop ecx
pop eax
ret
```

The key things here are to ensure that:

- backing up (along at least one of the above strategies) altogether covers *every* register whose obscuring by a called subroutine could affect the correctness of the whole program;
- in case when back-up involves multiple values (like `eax`, `ecx`, `edx` in the above example) — the order of restoring them from the stack should be reverse to the order of pushing them there (following the LIFO rule);
- the stack content at the end of a subroutine (just before `ret`) should be **indentical** with its content at the start of that subroutine (just after `call`) — which is necessary for properly retrieving the return address from the top of the stack.
  (For that reason, it would be unacceptable e.g. to push `eax` to the stack on the calling side but pop it back in the called subroutine code).

Other instructions

System calls

The examples discussed so far did not include a typical initial example featured in programming courses — a program printing out "Hello World!" to the screen. That has a reason: there is no assmebler program which would achieve this on an arbitrary operating system. Printing to the screen (and generally — the whole input/output communication) is realized with **system calls** (*syscalls*), i.e. special kinds of processor **interrupts** (described in the lecture "Processor and programs") for which a handler has been defined to pass the control to the operating system code, to let the system take the necessary actions.

Similarly, the responsibility of operating system also includes managing memory allocations on the **heap** (which is significantly more difficult than for the stack, as the necessary memory size is not known upfront, and moreover, the unpredictable order of freeing up allocated data leads to "holes" of unused memory appearing, and operating system support is needed to "fill" them). Therefore, reserving and freeing up memory on the heap also requires making syscalls, and consequently, it looks significantly differently e.g. between Windows and Unix platforms.

Bit-level operations

Last, we will discuss instructions which operate on variables, treating them as sequences of single bits.

On the lecture "Logic", we met logic operations (like AND, OR, XOR), though their arguments were single bits, e.g. "1 AND 0", "0 OR 1" etc. Such operations can be extended to multi-bit numbers, and perform them **bitwise**, i.e. separately on each position. For example:

$$10011010 \text{ AND } 11010001 = 10010000, \qquad 10011010 \text{ OR } 11010001 = 11011011.$$

In the assembler, the standard Boolean operator names typically represent bitwise operations, for example:

```
and ax, 1111111111011111
```

will result in clearing (i.e. setting to zero) the sixth rightmost bit in register `eax` (and leaving all the other bits unchanged). The same action can be written even more concisely:

```
and ax, ~32
```

where `~` denotes bitwise negation applied to the binary representation of the given number (as the number 32 has representation `0...00100000`, its bitwise negation gives exactly `1...11011111`).

The above code may have a practical application. In the ASCII encoding (described on the lecture "Encoding data"), the uppercase charaters differ from their lowercase counterparts only by the sixth rightmost bit value (e.g. A = 65, a = 97). Therefore, the above code just converts characters to their uppercase versions.

A standard application of bitwise operations is zeroing a register by the use of `xor`, for example:

```
xor eax, eax
```

clears the value of `eax`. That way of coding it is obvously less human-readable than `mov eax, 0`; however, it turns out to be more efficient (or, simply put, faster). This is the main reason for using bitwise operations in the assembler code.

Another type of operations worth mentioning here are the **bit shifts**, which append a given number of zeros to the binary representation of a number, either from the left or from the right. For example:

```
shl eax, 3
```

results in "shifting the value of `eax` by 3 bits to the left", that is, appending three zeros on the right: for example, $101_{(2)} = 5$ will turn into $101000_{(2)} = 40$. And conversely, shifting 40 by 3 bits to the right will produce 5. Hence, at least as long as we don't face any arithmetic overflows, these operations are equivalent to multiplication or (integer) division by 2^3. In practice, though, **several variants** of bit shifts have been defined which differ exactly in the treatment of arithmetic overflows.

The operations described above also have counterparts in higher-level languages (typically denoted by `&`, `|`, `~`, `<<` and `>>`). While in older times they were often used for optimizing programs, currently their importance in this regard has decreased — as optimizations of this kind have become a task for compilers, or even for the processor itself. Yet, bitwise operations and bit shifts still appear in high-level language code, for several reasons:

- historical grounds: some programmers are used to exploiting them;
- code clarity: `1 << 17` may look more legibly than `131072`, or `(int) Math.pow(2, 17)`;
- in case of C/C++ — ease of handling bit masks: e.g. many functions from the operating system library accept numeric arguments in which each bit carries its own meaning; in such case, bitwise operations significantly simplify managing such information.