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).

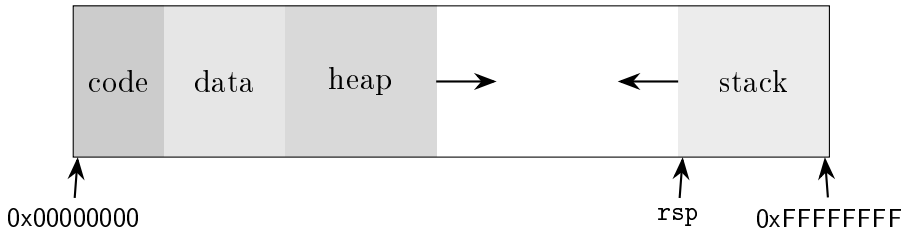

Figure 1. The typical layout of the address space of a process in the x86 architecture.
As it dates back to the times of 32-bit addressing, the space of available addresses has size 2^{32} B = 4 GiB.
The code and data segments (whose size is known upfront at the start of program execution) are located 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.

Primary memory

Computer memory is not uniform. On this lecture, we will discuss these types of memory which are fundamental from the viewpoint of the modern architecture (x86); they are used while executing *every* program. Besides the processor registers (already well known to us), these include: cache and RAM. Altogether, they're called **primary storage** (or *primary memory*), though both these terms are also used to describe the RAM layer alone.

Of course, there also exist other memory types, e.g. SSD and HDD drives, pendrives, DVD discs etc. These form the so-called **secondary** (or *external*) **memory**, which will be discussed on a later lecture. Unlike the primary memory, it is not available to the processor directly, but via the input/output interfaces. It's also not strictly *necessary* for executing *every* program, though it's an important component of every personal computer, as all kinds of **non-volatile memory** (keeping its content even after losing electric power) belong to the secondary memory.

Types of primary memory

The main two memory properties, from the user viewpoint, are **capacity** and **speed** (understood as the time needed to reaching the data, but also the amount of data transferred in 1 second). Sadly, these two features are hard to combine, which is caused both by economy (faster memory is more expensive, so available in smaller amounts) and technology (e.g. to have a very quickly reachable cache, we need to place it very close to the MMU, and that puts a limit on its size).

With respect to these two properties, the types of primary memory look as follows:

- **Main memory** (typically called simply **RAM**, or *primary memory*) — largest and slowest.
  It defines the area of memory available directly to the processor (as we already described in the lectures "Computer structure" and "Processor and programs"). On this lecture, we will discuss in more detail the process of translating between logical and physical addresses, and in particular, the important technique of *memory virtualization*.
- **Cache** — intermediate capacity and speed.
  Cache is physically placed inside the processor and acts as a buffer for exchanging data between the RAM and the registers. On this lecture, we will describe its purpose, structure and rules of operation.
- **Processor registers** — fastest and smallest; already discussed in detail in the lecture "Processor and programs".

Unfortunately, in practice, we face an unavoidable ambiguity of naming, with *primary memory*, *main memory* and *internal memory* used interchangeably to denote either the whole primary memory or just its largest and slowest layer. Also, the term *RAM*, while literally expanding to "random access memory (of any kind)", is commonly used to describe the main memory — and so will happen in the remaining part of this lecture.

Hardware properties

Static RAM (SRAM)

The most effective memory type is **static memory** (**SRAM**), called so because it does not need periodic refreshing of its content (unlike the dynamic RAM, DRAM).

On the other hand, SRAM is more expensive than DRAM, as a result of using more transistors per a single memory cell. This also makes SRAM less *dense* (i.e. taking physically more space). For these reasons, SRAM is used in processor registers and relatively small cache memories, but not in much larger RAM modules in personal computers.

Dynamic RAM (DRAM)

The RAM memory, clearly the largest component of primary memory, is manufactured as **dynamic RAM** (**DRAM**), and more specifically — since the early 21st century — as **DDR SDRAM**. That acronym stands for: *double data rate synchronous dynamic random-access memory*. Let us discuss the individual parts of this long name:

- *random-access memory* — this means that data can be accessed in an arbitrary (*random*) order; the opposite to this is memory with *sequential access*, which requires that data are read in some specific order;
- *dynamic* — this means that memory needs periodic refreshing its content (i.e. reading each memory cell and writing it back to the same place);
- *synchronous* — this means that the access to memory is regulated with the system clock, as opposed to *asynchronous* memory in which the access is regulated by internal memory signals;
- *double data rate* — this means that data are sent twice as frequently as the nominal clock rate would suggest — that is, on the *rising slope* as well as on the *falling slope* of the clock signal (see the lecture "Integrated circuits").

Other important properties of the SDRAM type memory are:

- *interleaving* — allows more than one read/write operation at a time;
- returning data in the form of multi-byte chunks (which is desirable because, as soon as a particular memory cell is fetched, its neighbors are also likely to be soon needed).

During its evolution, DDR SDRAM has existed in the following configurations:

- DDR SDRAM (bandwidth: from 1,600 MB/s to 3,200 MB/s);
- DDR2 SDRAM (bandwidth: from 3,200 MB/s to 8,533 MB/s);
- DDR3 SDRAM (bandwidth: from 6,400 MB/s to 19,200 MB/s);
- DDR4 SDRAM (bandwidth: from 12,800 MB/s to 25,600 MB/s);

- DDR5 SDRAM (bandwidth: from 25,600 MB/s to 57,600 MB/s).

Unlike cache, DDR SDRAM is a separate integrated circuit, which makes it possible to take it out from a computer and replace by another; often it's also possible to extend the existing RAM modules with additional ones. However, there are some technical limitations here. Physically, the memory is inserted into the *motherboard* which ensures communication between the computer components. Clearly, one cannot insert more RAM modules than the number of available memory ports. Another limitation comes from the processor, which is designed with an upper bound of supportable memory size. Also, not all processors / motherboards can support DDR4. Otherwise, a DDR4 module can be placed in a computer, though it will behave as DDR3 (which matters e.g. for the voltage input; DDR4 is capable of operating on a lower voltage, leading to reduced heat and increased performance).

The size of available RAM memory varies greatly across computers: in typical modern personal devices, it can range from a few up to hundreds of gigabytes.

An important characteristic of the processor and the motherboard is the number of **channels**. A *single-channel* architecture means that RAM has only one connection (*channel*) with the processor. *Dual-channel* means there are 2 such connections; the number of channels can be also 3, 4, or 8. Having more channels allows the processor to communicate simultaneously with more memory modules. Therefore, using e.g. two 4 GB modules will be more efficient than one 8 GB module. Benefitting from multi-channel transmission may, however, require ensuring that the modules used have the same manufacturer and/or capacity.

Cache

Generally, the concept of *cache* in computer science denotes the following situation:

- The starting point is storing data in a large though not so fast memory M — and wishing to accelerate access to these data.
- To achieve that, we introduce **cache** C, that is: a smaller buffer made of memory which allows faster access, containing those data from M which we expect to be relevant in the near future. Thus, we replicate some data stored in the original location M — though, in reward, we ensure faster reachability for them. Typically, writing to the memory is also done via this buffer.

In the case of **CPU cache**, the role of M will be played by RAM. However, it's worth noting that similar problems and solutions appear also elsewhere, including the software layer. (For example: for a large, distributed database, it often pays off to replicate some data and store them locally).

Rules of operation

Caches can generally work in various ways, depending on how we answer the following questions:

- Where in the buffer can we store the given value?
- If the buffer is filled up, how should we choose an old value to replaced by the new one?
- How long does it take to find a requested value in the cache?

- How much time do we lose in case of a cache **miss**?
  (That is related to the difference between access times for M and for C)
- In case when we do fetch a value from M, should we store in the cache just that value, or also some other values related to it?
  (An example from the database world: after retrieving a customer's last name, we may reasonably expect a need to fetch their first name — so, whenever we fetch the former, it may pay off to proactively also fetch the latter).

In the case of CPU cache, **speed** is a decision factor of utmost importance: checking for a value in the cache should take as little clock cycles as possible (and, to achieve that, we can even accept an increased risk of a cache miss). This leads to the following *simplified* rules of operation (which we will below amend to a somewhat more precise form):

- The cache is an array of 2^N **entries**, each of which contains some value from the RAM together with its source address.
- Whenever the CPU needs a value from an address A, it checks the cache only at the **index** A `%` 2^N (where `%` is the *modulo* operation, i.e. the division remainder).
- If we do fetch the value from an address A in RAM, we store it always at the index A `%` 2^N. The previous cache entry at that index is overwritten.

Such an approach is very fast (we check only one cache entry, which in addition requires very light computations, as A `%` 2^N is just the last N bits of A). However, it has its drawbacks: the risk of a miss is significantly increased (e.g. two frequently used addresses pointing to the same index would constantly expel each other from the cache); also, any cross-data relationships are disregarded.

For these reasons, in practice, CPU cache introduces the following enhancements:

- We allow an increased **set-associativity** D (typically: 2, 4, or 8), which means that each index stores D entries instead of one.
  Now, on every RAM access, we check *all* entries at the appropriate index. If any of them matches the deisred address, we have a cache hit (and can stop searching). If none of them was a hit, we claim a miss, fetch the value from RAM, and choose one of the D entries to be replaced.
  By doing this, we may slow down cache access by up to D times, in exchange for increasing the cache hit rate.
- We increase the **value size** of cache entries to 2^K bytes (typically, K is 6 or 7).
  This trick increases the chance of related data coexisting in the cache (as programmers often use data arrays, or sets of variables that had been allocated next to each other, etc.). It also decreases the fraction of cache capacity "wasted" on storing the source addresses.

  In practice: if we want to get the A-th byte of RAM, we transform A to the appropriate index of a 2^K-byte memory cell (result: "the $(A/2^K)$-th cell"), determine the appropriate index in the cache (result: $(A/2^K)$ `%` 2^N), and — if a cache hit occurred — pick the (A `%` 2^K)-th byte of the cached value.
  Note that all these address operations still boil down to very simple bit shifts or masks.
- Finally, to reduce memory footprint, we equip each cache entry no longer with the full source address but rather with its **non-obvious prefix** (omitting the last $N + K$ bits, which in practice makes up about a half of the total address length).

Cache levels

The CPU cache is split into (typically) 3 parts, called **levels** and denoted L1, L2, L3.

The **L1 cache** was already featured in the lecture about processor orders. It's here where the orders are initially decoded, and their arguments fetched. L1 cache observes a split into data part and instruction part (so that half of it can only store the actual data, and half of it — the code). Its size is modest — typically 256 kB. The set-associativity is typically 2 or 4. Each processor code has its own L1 cache. The transmission bandwidth is between 25 and 700 GB/s, and the access time is about 0.5 ns (reminder: $1\,\text{ns} = 10^{-9}\,s$).

L1 cache does not fetch data directly from RAM, but from another level, **L2 cache**. There, we lose the split into data part and instruction part, though we usually retain the split per processor core. The sizes vary greatly but usually fall between 64 kB and 12 MB. The set-associativity is typically 2, 4 or 8. The typical bandwidth is 15-200 GB/s, and typical access time is 5-10 ns.

The last level, **L3 cache**, is largest and slowest. Its size is typically 4-64 MB, and set-associativity is 16 or more. The bandwidth is 10-100 GB/s, and access time is typically 20-50 ns. This type of cache is shared across processor cores, which makes it crucial for synchronizing their work.

In some processors, there is also **L4 cache**, manufactured in the technology called **eDRAM** (*embedded DRAM*). That type has a higher memory density than SRAM, and is almost equally expensive. Despite being a dynamic memory (so requiring periodic refreshing), the technical details (integration with the processor and equipping with a dedicated controller) make the refreshing less visible for the user, to the point where eDRAM can be considered as if it was some simple kind of SRAM. Moreover, placing L4 cache inside the processor allows to use a more efficient transfer bus.

Addressing RAM memory

We will now discuss the topic of memory addressing. As explained on the lecture "On the assembler", processes are equipped with their private pools of RAM, to which they refer with the so-called **logical addresses**. The programmer (or the compiler) uses the logical addresses as if the first memory cell had address 0, and the whole area of RAM assigned to the process was contiguous. We will now discuss two techniques of address translation: **segmentation** and **paging**.

Segmentation

We have already came across segmentation on the lecture "On the assembler".

The memory assigned to a process is split into a few **segments**, whose specifications (starting address, size, access mode) are stored in the **segment descriptor table**. The **virtual address** — the one which can be used by an assmebler programmer — consists of two parts:

- the **segment selector** (specifying that the desired value belongs e.g. to the code segment)
- the **offset**, i.e. the index of the desired memory cell within that segment.

To obtain the **logical address** corresponding to a given virtual address, we need to look up the base address of the selected segment in the segment descriptor table, and then add the offset to it.

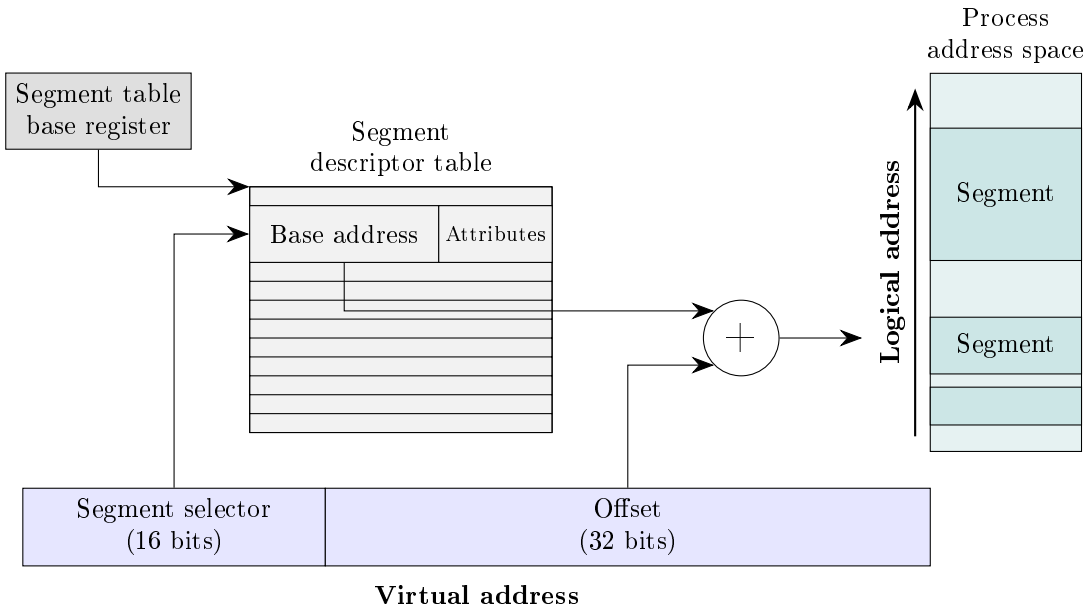

Figure 1. Segmentation in the Intel 80386 processor.

At this point, however, we need to make three disclaimers:

- Translating a virtaul address to a logical one used to happen in the way described above in the 32-bit processors from the x86 family. In the newer 64-bit version known as **x86-64**, in the 64-bit modes, segmentation becomes reduced (though still present to some extent).
- In the assembler code, the segment selector is often implicit as the compiler can deduce it from the context. (For example: the target addresses for jump instructions are interpreted within code segment; the data pointers within data segment, etc.)
  If the programmer needs to specify the segment selector explicitly, they can use an appropriate extended syntax, e.g. `cs:eax` means "an address with offset specified by the value of `eax`, and the code segment register value (`cs`) used as the segment selector".
- Again, the terminology commonly used to describe segmentation contains many divergences. For example:
  - the phrases *logical address* and *virtual address* happen to be used interchangeably; the logical address is also sometimes called the *line address*;
  - the word *segment* itself can mean an area of data described by a segment descriptor (like above), but also a functional subset of a process address space (e.g. the heap, stack etc.), or even an area of memory (with specific access mode) assigned to a process by the operating system (which does not need to coincide with a "segment" from the processor viewpoint).

Paging

Let us recall that the logical address should not be confused with the **physical address**, which specifies the actual placement of data in RAM. After all, as every process uses its own logical addresses between 0 and 4 GiB, various processes will generally operate on the same logical addresses, corresponding to different physical addresses. To translate the logical address into the physical one, we use a **page table** for the current process, maintained and stored by the operating system. The basic idea of translation using paging looks as follows:

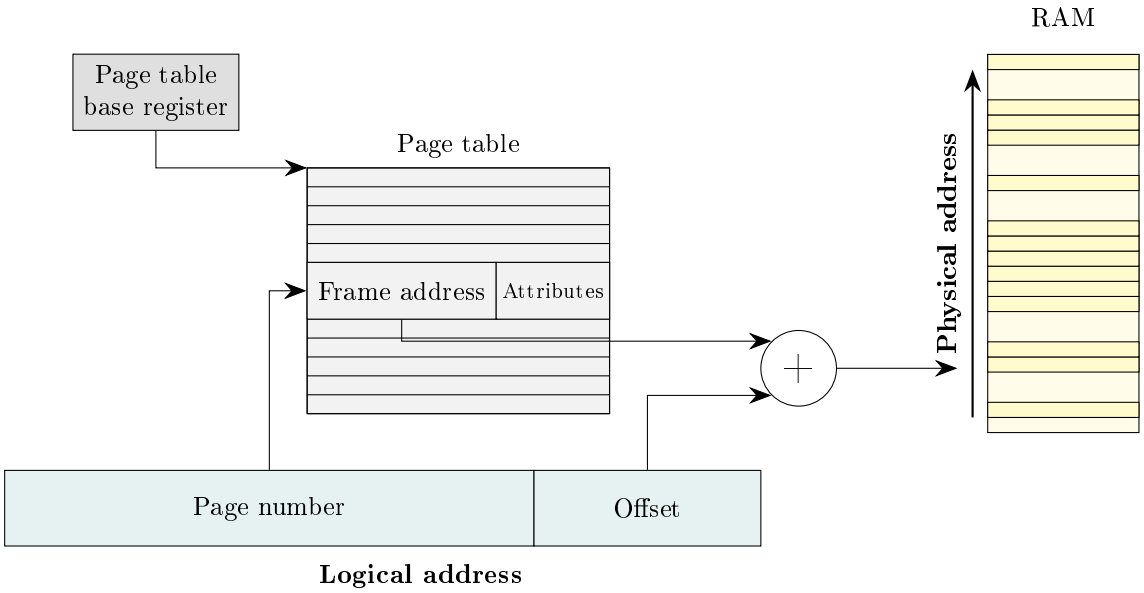

Figure 2. The overall idea of paging.
Although the general principle seems very similar to the picture for segmentation (see Figure 1), the main difference is in the equal sizes of all RAM frames, which allows unconstrained bookkeeping. (Another important difference, which we will discuss later in this lecture, is the *virtualization* technique).

What can be seen here is a split of the process address space into **pages** of a fixed size (typically 4 kiB), and the split of RAM into **frames** of the same size.

By introducing a fixed page size, the operating system can grant memory to processes according to their changing needs, assigning individual frames to individual processes (and to some logical addresses in their address spaces) in a *completely unconstrained* manner. This removes the problem known as **external memory fragmentation**, i.e. the scenario in which the free space in RAM memory gets split into so many smaller parts that some requests for a contiguous memory block cannot be served even though the request block size is smaller than the total amount of free memory. To the contrary, paging ensures that, as long as free memory exists at all, it is available in the form of free frames, and moreover, free frames have a suitable format for serving memory requests.

On the other hand, paging brings another kind of problem, the **internal fragmentation**: the amount of memory granted to a process is equal to its actual need but rounded up (to a whole number of pages); therefore, the rounding margin is effectively wasted. Therefore, choosing the page size involves a trade-off between reducing internal fragmentation (for which we'd prefer smaller pages) and reducing memory footprint of the page table (for which larger pages are better).

Multi-level paging

A single-level page table turns out to be an overly simplistic technique, especially when we look at its memory consumption. If the address space of a process (4 GiB) is split into pages of size 4 kiB, and each page descriptor takes e.g. 24 bits, then the page table for a single process will take 3 MiB, and all the tables for all currently running processes — often over 1 GiB! In such case, a common solution is to apply **multi-level paging**, in which the page table itself is subject to additional paging. For example, two-level paging proceeds as shown below:

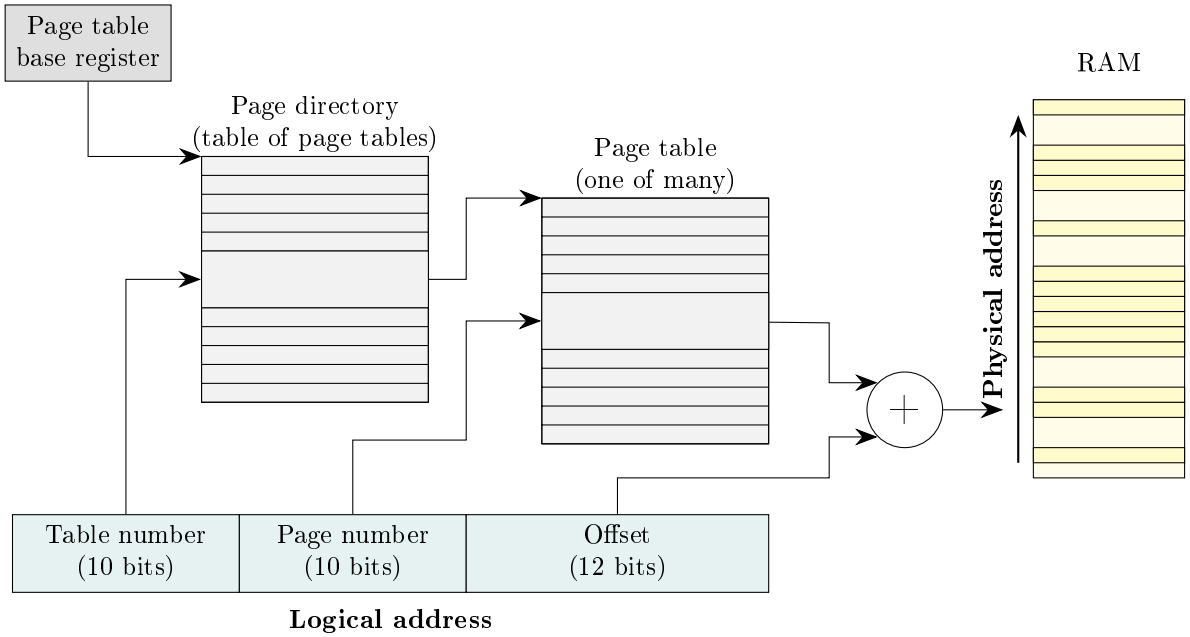

Figure 3. Two-level paging in the Intel 80386 processor.

A disadvantage of this solution is the increase of access time: by using n-level paging, reaching the desired RAM address may require altogether $n + 1$ read operations from that memory. (Generally, page tables are placed in RAM; however, due to how frequently they are referenced, processors contain a dedicated cache for them, called **TLB**, *translation lookaside buffers*). However, the main advantage of multi-level paging is more efficient management of memory used for page tables: those not used might not exist at all, and those used rarely can be safely sent back to secondary memory (see below).

The modern x86 processors use **4 or 5 levels** of paging.

Virtualization

One advantage of paging already known to us is solving the problem of external memory fragmentation. However, paging also brings additional opportunities, among which the crucial one is the so-called **memory virtualization**. This technique allows *referring to* a total amount of RAM memory (by which we mean the total size of address spaces of all processes) **larger than** its physical capacity! Such "cheating" is possible thanks to the fact that the frames of data which do not fit into RAM are in fact stored in the secondary memory (e.g. on the hard drive) and brought back to RAM when needed.

In the virtualized setup, fetching data from RAM engages the processor MMU as well as the operating system. To decide where the data should be fetched from (RAM or hard drive?), we use additional information stored in page descriptors. These contain in particular the **present bit**, specifying whether the page is available in RAM. If not (which means it should be retrieved from the hard drive), the processor generates a **page fault interrupt**, bringing control to the operating system, which manages fetching the appropriate frame from the hard drive to RAM. After such fetching, the page is now placed in the main memory, so we need to update its descriptor in the page table (in particular, store the new physical address, and flip the present bit).

This leads to another problem: when fetching a new page to the RAM, *where* should it be placed? It must replace another page, and clearly, the best one to replace would be one that is not going to be used anymore, which may exist for various reasons (e.g. because another process has just finished

executing, or because its local data are no longer live, e.g. as a result of freeing previously allocated memory). To allow detecting such cases, page descriptors contain the **use bit** controlled by the operating system.

However, the operating system will not always find an unused location in the RAM. If there's no free space while we fetch a new page, we must pick another *least necessary* page to be sent back to the hard drive. The problem of choosing a *least necessary* page is complex and can be solved with various techniques. The following are the simplest strategies for this:

- **FIFO** (*First In First Out*) — always send back the page which has been fetched longest time ago;
- **LRU** (*Least Recently Used*) — send back the page which has been used longest time ago;
- **LFU** (*Least Frequently Used*) — equip each page with a *reference counter*, initialized when the page is fetched and incremented on every reference to that page; send back the page for which the reference counter has the lowest value.

The above paragraph contains multiple simplifications but it (hopefully) shows the main idea behind memory virtualization. Unfortunately, describing the details of choosing pages and exchanging them between the hard drive and RAM is out of scope of this course.

Virtualization vs. caching

Note that the main idea of caching — moving data between a larger slower memory (M) and a smaller faster one (C) — does take place also in case of virtualization: this time, RAM plays the role of C, and M is the secondary memory. Yet, despite this apparent similarity, virtualization *is not* a special case of caching.

The purpose of caching is *accelerating* the access to memory M, by introducing C as a buffer. This affects just the efficiency; no cache would mean working just with M, which would be slower but would still work. The case of virtualization is different: the data in M (on a hard drive) are *useless* unless they are fetched to smaller memory C (RAM), and the goal of the whole mechanism is to improve *scalability* of the system (i.e. enable executing more processes at a time, and assigning a larger total amount of memory to those processess) at the cost of efficiency.