Odpovědi pište na zvláštní odpovědní list s vaším jménem a fotografií. Pokud budete odevzdávat více než jeden list s řešením, tak se na 2. a další listy nezapomeňte podepsat. Do zápatí všech listů vždy napište i/N (kde i je číslo listu, N je celkový počet odevzdaných listů).

Společná část pro otázky označené X

V příloze najdete specifikaci bitmapového obrazového formátu TARGA (soubory s příponou `.TGA`) – specifikaci formátu máte v příloze k dispozici ve dvou komplementárních variantách: 1) z Wikipedie, která je stručnější a přehlednější, a 2) detailnější specifikaci pro dohledání detailů chybějících na Wikipedii.
Dále **v příloze** najdete obsah souboru `AmazingPicture.tga` uloženého právě v uvedeném formátu `.TGA`, který jsme si zobrazili v hex vieweru.

Otázka č. 1 (X)

Nakreslete, jaký obrázek je v přiloženém `AmazingPicture.tga` souboru uložen – tj. nakreslete pixelovou mřížku (obdélníkovou matici), a pro každý uložený pixel určete jeho barvu (čtvereček pixelu touto barvou vybarvěte, nebo do něj napište označení barvy: K = black, R = red, G = green, B = blue, W = white). Vysvětlete, jak jste k vašemu výsledku došli.

Otázka č. 2 (X)

Pro přiložený `AmazingPicture.tga` soubor napište v desítkové soustavě datum a čas jeho uložení. Vysvětlete, jak jste k danému výsledku došli.

Otázka č. 3 (X)

Napište v Pythonu implementaci funkce:

```
def PrintInfo(targaFileName : str):
```

která jako svůj argument bere jméno souboru s obrázkem ve formátu `.TGA`. Pro jednoduchost nerealisticky předpokládejte, že funkci jsou předávány pouze korektní hodnoty argumentů, tj. není třeba kontrolovat jejich správnost (a tedy i že např. vstupní soubor je opravdu ve správném formátu).
Funkce má zobrazit rozměr obrázku v pixelech, a datum jeho uložení (první řádek popisuje rozměry jako šířka x výška, druhý datum uložení) – výpis má tedy vypadat následujícím způsobem:

```
1024 x 768
26.4.1986
```

Pokud byste potřebovali zjistit velikost nějakého souboru v bytech, tak je možné ji v Pythonu získat např. jako výsledek volání funkce:

```
os.path.getsize(fileName)
```

Otázka č. 4 (X)

Přiložený `AmazingPicture.tga` soubor uložíme na disk naformátovaný nějakým reálným souborovým systémem. Odhadněte a svůj odhad detailně vysvětlete, kolik bytů data souboru na disku opravdu zaberou (tj. o kolik se uložením souboru zmenší kapacita volného místa na disku).

Otázka č. 5

Pro reprezentaci znaménkových celých čísel můžeme použít např. reprezentaci *s explicitním znaménkovým bitem* (sign–magnitude), *dvojkový doplněk* (two's complement) nebo reprezentaci *s posunem* (bias). Detailně srovnejte tyto 3 různé reprezentace a uveďte jejich hlavní výhody a nevýhody.
V každé z těchto reprezentací v 8-bitové variantě zapište čísla:

```
125
-3
0
```

Otázka č. 6

Předpokládejte, že chceme reálné číslo `-65539,375` uložit do paměti od adresy `0x0C0244F0` jako typ `single`, tj. 32-bitové floating-point číslo dle standardu IEEE 754, tj. mantisa je normalizována se skrytou 1 a zabírá spodních 23 bitů, pak následuje 8-bitový exponent uložený ve formátu s posunem [bias] +127, a poslední bit, tedy `MSb`, je znaménkový bit. Napište v šestnáctkové soustavě hodnotu každého bytu paměti, ve kterém bude uložená nějaká část této hodnoty – předpokládejte, že pracujeme na 32-bitovém **little-endian** CPU.

Otázka č. 7

Napište v Pythonu implementaci funkce `Conv` využívající standardní typy z balíčku `numpy`:

```
def Conv(utf32 : List[uint32]) -> List[uint16]:
```

Funkce `Conv` má převést vstupní textový řetězec daný parametrem `utf32` z kódování UTF-32 (reprezentovaný jako standardní Python list instancí `uint32`) do kódování UTF-16 a výsledné znaky UTF-16 vrátit opět jako standardní Python list, ale typů `uint16`.
Pozor, předpokládejte, že funkci programujeme jako součást standardní Python knihovny, tedy **nesmíte** používat žádné Pythonové **funkce pro převod textu** mezi různými kódováními – **smíte** použít jen standardní **bitové operace**.
Unicode znaky z rozsahu `$010000` až `$10FFFF` se v UTF-16 kódují následujícím způsobem:

(1) Od kódu znaku se odečte hodnota `$010000`, a výsledné 20-bitové číslo se rozdělí na dvě 10-bitové části, které se zakódují dle následujících pravidel.

(2) Nejvyšších 10-bitů 20-bitové hodnoty se uloží do nejnižších 10 bitů prvního 16-bitového surrogate znaku (leží na nižší adrese, tj. více k začátku textového řetězce). Horních 6 bitů první surrogate má být nastaveno na (vlevo je hodnota bitu 15, vpravo bitu 10):

```
1101 10
```

(3) Nejnižších 10-bitů 20-bitové hodnoty se uloží do nejnižších 10 bitů druhého 16-bitového surrogate znaku (leží na vyšší adrese). Horních 6 bitů druhé surrogate má být nastaveno na (od první surrogate se liší pouze 10. bitem):

```
1101 11
```

Společná část pro otázky označené Y

Předpokládejte níže popsaný **32-bitový little-endian** CPU s obecnou registrovou architekturou vycházející z architektury x86 (IA-32) – jedná se o procesor s 32-bitovým adresovým prostorem. Procesor má mimo jiné obecné registry `EAX`, `EBX`, `ECX`, `EDX` (pro každý registr existuje i pohled na jeho spodních 16 bitů pod jmény registrů `AX`, `BX`, `CX`, `DX`, a stejně tak i pohled na jeho spodních 8 bitů pod jmény `AL`, `BL`, `CL`, `DL`), příznakový registr `EFLAGS` s běžnými příznaky, a registr `EIP` (instruction pointer). V instrukční sadě jsou mimo jiné i následující instrukce (příznakový registr modifikují pouze aritmetické operace, ale instrukce přenosu dat nikoliv) – každá z uvedených instrukcí má 32-bit, 16-bit a 8-bitovou variantu, pro konkrétní variantu instrukce jsou **oba** její **operandy vždy** o **stejné bitové přesnosti** (tedy např. 32-bitová varianta má všechny operandy 32-bitové, apod.; variantu instrukce volíme druhem zdrojového/cílového registru):

`MOV reg,imm/[addr]`	(load register)
`MOV [addr],reg`	(store register)
`MOV reg0,reg1`	(copy from `reg1` to `reg0`)
`ADD reg,imm/[addr]/reg`	(add without carry)
`ADC reg,imm/[addr]/reg`	(add with carry)
`SUB reg,imm/[addr]/reg`	(subtract without borrow)
`SBB reg,imm/[addr]/reg`	(subtract with borrow)
`CLC`	(clear carry)
`STC`	(set carry)
`OR reg,imm/[addr]/reg`	(bitwise OR)
`AND reg,imm/[addr]/reg`	(bitwise AND)
`SHR reg,imm/CL`	(logical shift right)
`SHL reg,imm/CL`	(logical shift left)

Všechny výše uvedené instrukce se dvěma operandy mají vždy **vlevo cílový** a **vpravo zdrojový** operand. Instrukce mohu mít pro každou „bitovost“ ještě následující varianty operandů (povolené varianty viz definice konkrétní instrukce):

- immediate hodnota `imm`
- absolutní adresa `[addr]`, kde `[addr]` může být:
  - `[imm]` adresa daná konstantou `imm`
  - `[reg32]` adresa daná obsahem 32-bit registru `reg32`
  - `[reg32 +/- imm]` adresa daná součtem/rozdílem obsahu 32-bit registru `reg32` a konstanty `imm`
- libovolný registr `reg`

Otázka č. 8 (Y)

Předpokládejte, že v programu v jazyce C# máme deklarované 4 proměnné `a`, `b`, `c`, `d` typu `ulong`, který je ekvivalentní typu `uint64` z balíčku `numpy`. Víme, že pro proměnné překladač vyhradil místo hned za sebou v pořadí `a`, `b`, `c`, `d` směrem k vyšším adresám, kde první proměnná `a` leží na adrese `0x00051000`. Dále víme, že pro dočasné proměnné můžeme využít libovolnou paměť mezi adresami `0x7F000000` a `0x7F00FFFF`. Zapište v assembleru výše uvedeného procesoru, jak by se přeložil níže uvedený C# výraz (neprovádějte žádné optimalizace a algebraická zjednodušení [jako odstraňování závorek]):

```
a = b - (c + d)
```

Otázka č. 9 (Y)

Navrhněte, jak by mohl vypadat strojový kód uvedené varianty procesorů x86 – samozřejmě není třeba, aby váš návrh odpovídal reálné procesorové architektuře x86, nicméně by měl být přiměřeně realistický a principálně odpovídat tomu, jak strojový kód běžných CPU vypadá. Předpokládejte, že bychom samozřejmě nakonec chtěli podporovat všechny výše uvedené instrukce, nicméně ve svém řešení detailně popište pouze, jak bude vypadat strojový kód pro pouze 32-bitové varianty instrukcí `ADD` a `SUB`. Pro instrukce `ADD` a `SUB` uvažujte všechny výše uvedené varianty operandů, nicméně předpokládejte, že jako registr může být v operandech instrukce použit pouze některý z výše uvedených 4 obecných registrů.

Zapište hodnotu každého bytu paměti, kde by byla uložena nějaká část strojového kódu dle vašeho návrhu, pokud bychom v něm chtěli zapsat níže uvedenou posloupnost instrukcí (kód budeme ukládat od adresy `0x00010000`):

```
ADD EAX, [12345678h]
ADD EAX, DEADBEEFh
ADD EBX, EAX
SUB EBX, ECX
```

Otázka č. 10

Předpokládejte, že navrhujeme komunikační protokol pro SRAM paměti připojitelné přes I^2C sběrnici. Víme, že budeme chtít vyrábět paměťové čipy s kapacitami 4 kB, 64 kB, 128 kB, a 1 MB. Chceme, aby všechny naše paměťové čipy mely stejný komunikační protokol.

Napište, jak budou obecně vypadat požadavky na čtení a zápis pomocí vámi navrženého protokolu – to pak ilustrujte na příkladu, jak bude vypadat požadavek na zápis bytu `0x5F` na adresu `0x123`, a jak požadavek na čtení z adresy `0x456`, na který paměť odpoví hodnotou bytu `0x71`.