1) Můžeme při stavbě počítače k jeho CPU s Harvardskou architekturou připojit datovou a kódovou paměť s libovolnou kapacitou? Pokud ano, tak proč? Pokud ne, tak čím je kapacita pamětí omezená?

2) Může CPU s Harvardskou architekturou principiálně stanovovat různé limity na kapacitu připojené datové a kódové paměti?

3) Co je *instrukce* procesoru?

4) Co je *instrukční sada* procesoru?

5) Co je *strojový kód*?

6) Má každé CPU registr *program counter* nebo *instruction pointer* nebo podobný? Co je obsahem takových registrů?

7) Co je *opcode instrukce* procesoru?

8) Jak na Harvardské architektuře bude koncepčně fungovat *překladač* nějakého vyššího programovacího jazyka (např. jazyka C, nebo C#)? Jaká data jsou vstupem a jaká výstupem překladače?

9) Co je *interpret* jazyka Python? Jaký je rozdíl mezi interpretováním Pythonového programu a jeho překladem (pokud bychom hypoteticky měli překladač Pythonu)?

10) Čím na úrovni strojového kódu identifikujeme jednotlivé proměnné programu?

Definice: Pokud chceme zobrazit obsah paměti, tak ho typicky zobrazujeme jako posloupnost 8-bitových čísel zapsaných v šestnáctkové soustavě – každé takové číslo reprezentuje hodnotu 1 bytu. Pokud bychom čísla zapisovali v jednom řádku, tak typická konvence je, že adresy rostou zleva doprava, tj.: a) nejvíce vlevo je hodnota bytu na nejnižší adrese ze zobrazovaných, b) hodnota o 1 vpravo je hodnota bytu na adrese o 1 větší.

Definice: Jelikož často chceme zobrazit obsah větší části paměti, tak pokud bychom v zobrazení striktně reflektovali, že je paměť pouze lineální posloupnost bytů, tak by nám dle definice výše při zobrazení vycházely velmi dlouhé řádky, což by bylo nepraktické. Proto typicky obsah paměti zobrazujeme jako tzv. *hexview*, nebo *hexdump*, nebo *hex* zobrazení – zde pouze kvůli zobrazení zavedeme umělé rozdělení dat do více řádků, tak že na každý řádek (případně kromě posledního, případně kromě prvního) zobrazíme konstantní počet bytů (typicky 16, nicméně relativně běžné jsou i jiné mocniny 2). Typická konvence je rozšířením konvence z definice výše:

a) adresy rostou zleva doprava, tj. na každém řádku byte v nejlevějším sloupci leží na nejnižší adrese bytů na řádku, byte v nejpravějším sloupci leží na nejvyšší adrese bytů na řádku;
b) poté adresy rostou shora dolů, tj. byty na řádcích zobrazených výše leží na nižších adresách, než byty zobrazené na řádcích zobrazených níže;
c) pro přehlednost bývá v záhlaví každého řádku uvedena adresa (typicky zapsaná v šestnáctkové soustavě), na které leží nejlevější byte na řádku;
d) *někdy* bývá pro přehlednost v záhlaví každého sloupce uvedený tzv. offset (jako číslo zapsané v šestnáctkové soustavě), který pro každý byte daného sloupce určuje, jakou hodnotu musíme přičíst k adrese řádku (viz bod c), abychom dostali celou adresu daného bytu;
e) jelikož se očekává použití výše uvedených konvencí, kdy hodnota bytu, adresa řádku, i offset na řádku jsou zapsány v šestnáctkové soustavě, tak je běžné, že se u žádné z těchto hodnot explicitní uvedení zápisu v šestnáctkové soustavě neuvádí (tj. hodnoty nemají prefix `0x` nebo `$`, apod.).

Poznámka: *Hexview*/*hexdump* budeme probírat **na 12. přednášce** (tj. **je třeba** ho znát a chápat **ke zkoušce**), nicméně jako spoiler si ho představujeme již v zde v zadání této sady self-assessment úloh, protože nám zpřehlední zadání a řešení úloh v této sadě.

Pozor: Toto „dvourozměrné“ rozdělení paměti na řádky a sloupce je zavedeno pouze uměle pro „hezké zobrazení“ jejího obsahu, ale samotná paměť je z pohledu CPU vždy pouze lineální jednorozměrná posloupnost bytů.

Příklad: Z hexdumpu uvedeného dále na straně 2 můžeme vyčíst např. následující informace:

a) zobrazuje obsah paměti od adresy `0x0010FC80` do adresy `0x0010FCB6`,
b) z bodu a) vyplývá, že uvedený hexdump nezobrazuje kompletní obsah paměti, protože hodnoty bytů na adresách `0x00000000` až `0x0010FC7F` nejsou ve výpisu uvedené,

c) na `0x0010FC80` leží byte s hodnotou `0xFF`,
d) na `0x0010FC8C` leží byte s hodnotou `0x4D`,
e) na `0x0010FC95` leží byte s hodnotou `0x0A`,
f) jediný byte s hodnotou `0x86` leží na adrese `0x0010FCA1`,
g) z bodu a) může, ale nemusí, vyplývat i následující: dá se očekávat (nicméně to nemusí být pravidlo), že kapacita paměti bude nejspíše nějaký násobek základní jednotky kapacity jako kB, MB, apod. – jelikož je adresa `0x0010FCB6` větší než `0x00100000` (= 1 MB), tak bychom si mohli tipnout, že kapacita paměti bude nejméně nejbližší násobek 1 MB, tj. 2 MB, tedy že hexdump nezobrazuje hodnoty bytů minimálně na adresách `0x0010FCB6` až `0x001FFFFF`.

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0010FC80: FF D8 FF E1 2F FE 45 78 69 66 00 00 4D 40 00 2A
0010FC90: 00 00 00 00 00 0A 01 0F 00 02 00 00 00 06 FC FF
0010FCA0: 00 86 01 10 00 02 00 00 00 0B 01 00 00 8C 01 12
0010FCB0: 00 03 00 00 00 01 1B
```

11) Pro výše uvedený hexdump nedokážeme určit přesnou kapacitu paměti, část jejíhož obsahu je výše uvedena. V bodě g) jsme uvedli tip, že kapacita paměti by mohla být nejméně 2 MB. Můžeme si z informací v hexdumpu tipnout největší možnou kapacitu, kterou uvedená paměť může mít? Pokud ano, tak jaká tato maximální kapacita je?

12) Co to je endianita dat? Jakých dat se týká?

13) Srovnejte pojem byte order v komunikačním protokolu s pojmem endianita dat. Je mezi nimi nějaký rozdíl, resp. spojitost?

14) Předpokládejte, že všechny proměnné jsou v paměti uložené **v LE pořadí**. Pro výše uvedený hexdump napište v desítkové soustavě hodnotu níže uvedených proměnných:

a) 32-bitová unsigned proměnná A na adrese `0x0010FCA9`,
b) 8-bitová unsigned proměnná B na adrese `0x0010FC85`,
c) 16-bitová signed proměnná C na adrese `0x0010FC9E`.

15) Předpokládejte, že všechny proměnné jsou v paměti uložené **v BE pořadí**. Pro výše uvedený hexdump napište v desítkové soustavě hodnotu níže uvedených proměnných:

a) 32-bitová unsigned proměnná A na adrese `0x0010FC93`,
b) 8-bitová unsigned proměnná B na adrese `0x0010FC8F`,
c) 16-bitová signed proměnná C na adrese `0x0010FCB5`.

16) Jaký je rozdíl mezi *volatile* a *non-volatile* pamětí? Jakou bychom použili pro datovou a jakou pro kódovou paměť?

17) Jaký je rozdíl mezi *Harvardskou* a *von Neumann* architekturou?

18) Jak na von Neumann architektuře bude koncepčně fungovat *překladač* nějakého vyššího programovacího jazyka (např. jazyka C)? Je možné pak přeložený program spustit rovnou na místě, kde byl vygenerován?

19) Co je *assembler*? Jaký je jeho vztah ke strojovému kódu?

20) Co dělá *instrukce (nepodmíněného) skoku*?

21) K čemu slouží *obecné registry* procesoru?

22) Co dělá instrukce *load*?

23) Co dělá instrukce *store*?

Poznámka: V dále uvedených úlohách použijte slidy 12 až 28 z prezentace 7. přednášky jako referenci k základním instrukcím nepodmíněného skoku, load, a store na procesoru 6502. Pro zápis v assembleru využívejte konvenci assembleru pro 6502 uvedenou na slidech. Na slidech též najdete popis uvedených instrukcí na úrovni strojového kódu – nad rámec informací uvedených ve slidech předpokládejte, že instrukce `STA` s absolutní adresou má opcode `0x8D` (tato instrukce je store ekvivalentem k loadu `LDA` s absolutní adresou a s opcodem `0xAD`, tedy předpokládejte i

ekvivalentní strukturu celé instrukce na úrovni strojového kódu).

24) Máme níže uvedené příkazy v jazyce C, které používají 32-bitové proměnné `one` (leží na adrese `0xA404`) a `two` (leží na adrese `0xA410`). Tyto příkazy přímočaře přeložíme do strojového kódu procesoru 6502 – strojový kód příkazu a) uložíme od adresy `0x1400`, strojový kód příkazu b) uložíme od adresy `0x1500`, strojový kód příkazu c) uložíme od adresy `0x15FC`. Zapište, jak bude daný příkaz vypadat i) v assembleru 6502, ii) ve strojovém kódu 6502 (jako posloupnost bytů zapsaných v šestnáctkové soustavě, ideálně ve formě hexdumpu):

a) `two = one;`
b) `one = 1277;`
c) `two = -2;`

25) Přepokládejte, že CPU 6502 má v registru `PC` uloženou hodnotou `0x2000`, a začne zpracovávat níže uvedený strojový kód – viz hexdump paměti před začátkem provádění programu. Sledujte běh programu, dokud `PC` nedosáhne hodnoty větší něž `0x201F`. Pro každý byte v paměti, který procesor při vykonávání tohoto kódu změnil, tak napište jeho adresu a jeho konečnou hodnotu (tj. hodnotu, která je v daném bytu paměti uložená po dokončení programu). Na straně X je hexdump paměti z této otázky uvedený ještě jednou větším písmem, pokud byste si ho chtěli vytisknout a dělat do něj poznámky.

```
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2000: A9 03 8D 00 A9 4C 0A 20 A9 AB 8D 01 A9 EA AD 09
2010: 20 8D 02 A9 4C 00 50 EA EA EA EA EA EA EA EA EA
```

26) Přepokládejte, že CPU 6502 má v registru `PC` uloženou hodnotou `0x2000`, a začne zpracovávat níže uvedený strojový kód. Sledujte běh programu, dokud `PC` nedosáhne hodnoty větší něž `0x201F`. Napište hexdump paměti mezi adresami `0x8000` a `0x800F` po dokončení běhu programu. Níže je uvedený hexdump paměti před začátkem provádění programu. Na straně X je hexdump paměti z této otázky uvedený ještě jednou větším písmem, pokud byste si ho chtěli vytisknout a dělat do něj poznámky.

```
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
2000: A9 FF 4C 10 20 A9 EA 8D 0A 20 8D 0B 20 8D 0C 20
2010: A9 00 8D 00 80 A9 12 8D 11 20 4C 05 20 8D 01 80
...
8000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Sada Python programovacích úloh (propojení 6-té a 6 ½-té „přednášky“ a ilustrace rozdílu mezi reprezentací a zobrazením čísel**) – společné poznámky:** Vezměte si vždy jeden z projektů `PySimulatedI2c-Gpio7SegmetDisplays` (viz níže specifická varianta u každé úlohy), který v souboru `PySimulatedI2cMainProgram.py` obsahuje kostru programu, který máte doplnit v řešení úloh níže. Součástí projektu je stejná implementace simulovaných příkazů CPU pro komunikaci po I^2C sběrnici jako v sadě self-assessment úloh k 5. přednášce – v zadání 5. sady úloh také najdete detailní popis fungování těchto funkcí, a poznámky k jejich používání. Navíc jsou v projektech doplněny nadstavbové funkce ze „vzorového řešení“ úloh k 6. přednášce – ty můžete ve svém řešení použít (může vám to řešení úlohy zjednodušit), ale nemusíte.

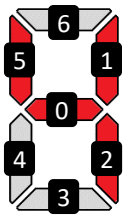
Na simulované sběrnici jsou tentokrát připojeny čtyři 16-bitové tzv. GPIO řadiče, viz níže, typu *Texas Instruments TCA9555*. Při řešení úloh je třeba mít k dispozici datasheet tohoto slave zařízení, viz přiložený soubor „`06-SA-tca9555 - I2C 16-bit GPIO controller.pdf`“[1] – z datasheetu je pro nás relevantní pouze strana 1, a strany 19 až 24.

N-bitový GPIO řadič je zařízení, které nám umožňuje nezávisle ovládat N nezávislých digitálních signálů (výstupních linek), resp. zjišťovat stav (hodnotu) N nezávislých digitálních signálů (vstupních linek). N-bitový GPIO řadič bude mít tedy pro tuto funkci vyhrazeno N disjunktních pinů – každý z těchto N pinů může fungovat buď jako vstupní nebo jako výstupní (ale nemůže mít obě funkce současně). To, které piny slouží jako vstupní a které jako výstupní, se typicky dá nastavit hodnotou v tzv. „konfiguračního registru“ (resp. „směrového registru“) GPIO řadiče – to platí i pro použitý řadič *Texas Instruments TCA9555*. My budeme využívat GPIO řadič pouze jako výstupní zařízení – to, zda pak GPIO řadič na konkrétních pinech generuje signál logické `0` nebo logické `1`, lze typicky nastavit hodnotou tzv.

[1] Původní zdroj (listopad 2020): https://www.ti.com/lit/ds/symlink/tca9555.pdf

„výstupního registru“ – opět platí i pro použitý *Texas Instruments TCA9555*: výstupy na pinech `P00` až `P07` pak odpovídají hodnotám `00.0` až `00.7`, a výstupy na pinech `P10` až `P17` pak odpovídají hodnotám `01.0` až `01.7`.

Připojené sedmi-segmentové displeje: Uvedený GPIO řadič *Texas Instruments TCA9555* má 2 osmi-bitové porty (skupiny pinů ovládané jedním registrem) – ke každému portu je připojený jeden klasický „sedmi-segmentový“ displej pro zobrazení jedné číslice (takové displeje můžete znát např. z různých digitálních hodin). Ke každému GPIO řadiči jsou tedy připojeny 2 nezávislé displeje – jelikož jsou k simulované I^2C sběrnici připojeny 4 GPIO řadiče, tak můžeme ovládat celkem 8 displejů pro zobrazení 8 číslic – pokud si spustíte kostry programů z úloh níže, tak se vám zobrazí simulovaný výstup na (pro zatím) zhasnuté displeje. Na obrázku níže je uvedené mapování jednotlivých segmentů displeje na piny GPIO řadiče – „sudé displeje“ (počítáno od 0 zprava) mají piny s prefixem `P0`, „liché displeje“ mají piny s prefixem `P1`. Za předpokladu, že na displeji chceme rozsvítit symbol např. `4`, viz obrázek (červená = svítící segmenty = výstup na `1`, šedá = zhasnuté segmenty = výstup na `0`), tak musíme zařídit, aby GPIO řadič generoval výstupní hodnotu `0` na pinech `P03`, `P04`, `P06` (resp. `P13`, `P14`, `P16` pro „lichý displej“), a výstupní hodnotu `1` na pinech `P00`, `P01`, `P02`, `P05` (resp. `P10`, `P11`, `P12`, `P15` pro „lichý displej“):



Mapování hodnoty výstupního registru GPIO řadiče na rozsvícené segmenty displeje si můžete vyzkoušet v testovacím projektu `Simulated7SegmentDisplayTester` – příklad použití najdete na obrázku `Simulated7SegmentDisplayTester-UsageExample-(without-segment-IDs).png` z příloh.

Poznámka: V řešení níže uvedených úloh pro jednoduchost neřešte a neošetřujte chybové stavy. Předpokládejte, že jsou všechna zařízení na I^2C sběrnici správně připojena, a komunikace s nimi probíhá vždy bez chyb a dle specifikace.

27) Doplňte program v `PySimulatedI2cMainProgram.py` v projektu `PySimulatedI2c-Gpio7SegmetDisplays-1-DisplayPJ-ASSIGNMENT` tak, aby na připojených displejích zobrazil dvoupísmenný text „`PJ`“ – viz `TODO` komentář, a obrázek `DisplayPJ-expected-output` s referenčním výstupem.

28) **Pro lehce pokročilé** (je třeba napsat trochu komplexnější program, nicméně veškeré potřebné znalosti k tomu máte již všichni): Doplňte program v `PySimulatedI2cMainProgram.py` v projektu `PySimulatedI2c-Gpio7SegmetDisplays-2-DisplayHexNumber-ASSIGNMENT` tak, aby na konzoli (se standardního vstupu, tj. např. pomocí funkce `input()`) četl 32-bitová bezznaménková čísla zapsaná v desítkové soustavě, a každé takové číslo zobrazil na připojených displejích jako číslo v šestnáctkové soustavě – viz `TODO` komentář, a obrázek `DisplayHexNumber-output-example` s referenčními příklady výstupů programu pro různé vstupní hodnoty.

29) **Pro lehce pokročilé** (je třeba napsat trochu komplexnější program, nicméně veškeré potřebné znalosti k tomu máte již všichni): Upravte program z řešení 15 tak (resp. upravte program dle kostry v projektu `PySimulatedI2c-Gpio7SegmetDisplays-3-DisplayHexOrDecNumber-ASSIGNMENT`), aby si uživatel mohl zvolit, zda se číslo zadané v desítkové soustavě bude zobrazovat v soustavě desítkové nebo šestnáctkové – viz komentáře a text v programu a obrázek `DisplayHexOrDecNumber-output-example` s referenčním příkladem výstupu programu.

Poznámka: Cílem těchto úloh je, abyste si doma v rámci opakování látky z přednášky mohli sami ověřit, jak jste látce porozuměli. Úlohy jsou koncipovány víceméně přímočaře, a po projití poznámek a případně video záznamů z přednášek by jejich řešení mělo být zřejmé. Pro řešení úloh tedy není potřeba studium látky nad rámec probraný na přednáškách (nicméně je třeba mít i znalosti a pochopení látky z paralelně probíhajících přednášek a cvičení z předmětu Programování I). Pokud i po detailním a opakovaném projití látky z přednášek máte s řešením těchto úloh problém, tak se na nejasnosti co nejdříve ptejte na on-line konzultaci k přednášce, případně zvažte se mnou domluvit na konzultaci (zvlášť pokud tento stav u vás přetrvává i po dalších přednáškách).

Upozornění: Úlohy jsou vybrány a postaveny tak, abyste si po přednášce a před přednáškou následující mohli rychle připomenout hlavní části probrané látky a ověřit si její pochopení. Nicméně úlohy nejsou vyčerpávajícím přehledem látky z přednášek a tady rozhodně nepokrývají kompletní látku přednášek, která bude vyžadována u zkoušky. Pokud tedy u každé úlohy víte, jak by se měla řešit, tak to ještě neznamená, že jste dostatečně připraveni na zkoušku – nicméně jste jistě na velmi dobré cestě. Každopádně nezapomeňte, že na zkoušce se vyžaduje pochopení a porozumění právě všem konceptům ze všech přednášek, a navíc jsou zkouškové příklady postaveny komplexněji tak, aby ověřily také vaši schopnost přemýšlet nad látkou napříč jednotlivými přednáškami.

Hexdump paměti se strojovým kódem procesoru 6502 **z otázky 25:**

```
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

2000: A9 03 8D 00 A9 4C 0A 20 A9 AB 8D 01 A9 EA AD 09

2010: 20 8D 02 A9 4C 00 50 EA EA EA EA EA EA EA EA EA
```

Hexdump paměti se strojovým kódem procesoru 6502 **z otázky 26:**

```
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

2000: A9 FF 4C 10 20 A9 EA 8D 0A 20 8D 0B 20 8D 0C 20

2010: A9 00 8D 00 80 A9 12 8D 11 20 4C 05 20 8D 01 80

...
8000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```