ZADÁNÍ SEMESTRÁLNÍ PRÁCE
EMULÁTOR POČÍTAČE S CPU ARCHITEKTURY MISC

Zadání

Naprogramujte v ANSI C přenositelnou[1] **konzolovou aplikaci**, v podstatě jednoduchý, ale plně funkční, *emulátor* teoretického počítače – nazvěme ho třeba K's Machine (KM) – s procesorem architektury MISC (= Minimal Instruction Set Computer), který bude na hostitelském počítači vykonávat programy (v binární podobě, tj. ve strojovém kódu) pro níže popsaný minimalistický teoretický počítač[2].

Emulátor se bude spouštět příkazem

`X:\>kmemu.exe`[3] ⟨*program_ve_strojovém_kódu*`.kmx`⟩ [⟨*výstupní soubor*[`.txt`]⟩]↵

Symbol ⟨*program_ve_strojovém_kódu*⟩ zastupuje povinný parametr – název vstupního binárního souboru s programem ve strojovém kódu pro procesor KM. Přípona `.kmx` (= K's Machine Executable; obdoba windowsovského `.exe`, = Executable), označující spustitelný soubor pro náš teoretický procesor, musí být vždy uvedena.

Symbol ⟨*výstupní_soubor*⟩ pak představuje nepovinný parametr, kterým je jméno textového souboru, do kterého se bude ukládat výstup při vykonávání programu, tj. informace/hodnoty, které emulátor vypisuje na obrazovku při vykonávání instrukcí programu. Není-li druhý, nepovinný, parametr uveden, směrujte výstup emulátoru do konzole (do standardního streamu `stdout`).

Úkolem Vámi vyvinutého programu tedy bude:

1. Načíst prvním parametrem určený spustitelný soubor (program) ve strojovém kódu ve formátu KMX pro níže popsaný teoretický procesor.
2. Vytvořit (podle údajů uložených ve spustitelném souboru) kódový a datový segment pro běh programu.
3. Vykonat program uložený v kódovém segmentu provedením všech jeho instrukcí, včetně smyček, volání podprogramů, atd.
4. Provádí-li se právě instrukce, jejímž úkolem je vypsat nějaký údaj na obrazovku, vypsat tento údaj do konzole, případně (je-li při spuštění programu uveden nepovinný druhý parametr) uložit do specifikovaného výstupního textového souboru.
5. Po dokončení vykonávání programu uvolnit paměť obsazenou datovým a kódovým segmentem vykonávaného programu a následně emulátor ukončit.

Váš program může být během testování spuštěn například takto (v operačním systému Windows):

```
X:\>kmemu.exe "E:\My Work\Test\mersenne.kmx" mersenne.txt↵
```

nebo takto:

[1] Je třeba, aby bylo možné Váš program přeložit a spustit na PC s operačním prostředím Win32/64 (tj. operační systémy Microsoft Windows NT/2000/XP/Vista/7/8/10/11) a s běžnými distribucemi Linuxu (např. Ubuntu, Debian, Red Hat, atp.). Server, na který budete Vaši práci odevzdávat a který ji otestuje, má nainstalovaný operační systém Debian GNU/Linux 11 (bullseye) s jádrem verze 5.10.0-28-amd64 a s překladačem gcc 10.2.1-6.

[2] Návrh tohoto teoretického počítače vychází konceptuálně z Turingova teoretického počítače *Universal Computing Machine* a následně prvních realizovaných počítačů *Manchester Baby*, *Manchester Mark 1*, *EDSAC*, *ENIAC*, apod. Ovlivněn byl částečně samozřejmě i „současností" v podobě procesoru Intel 8086.

[3] Přípona `.exe` je povinná i při sestavení v Linuxu, zejm. při automatické kontrole validačním systémem.

```
X:\>kmemu.exe "E:\My Work\Test\primes.kmx" output\primes.out↵
```

nebo pouze takto:

```
X:\>kmemu.exe bubble.kmx↵
```

První z uvedených příkladů spuštění by měl vést k vytvoření výstupního textového souboru `mersenne.txt` v tomtéž adresáři, kde se nachází spustitelný soubor `mersenne.kmx`. Druhý příklad vytvoří výstupní textový soubor `primes.out` v adresáři `output`, který je podadresářem adresáře, ve kterém se nachází spustitelný soubor `primes.kmx`. Při vykonání třetího příkladu bude výstup emulátoru směřován do konzole.

Spuštění v UNIXových operačních systémech (GNU/Linux, FreeBSD, macOS, atp.) může vypadat např. takto:

```
$./kmemu.exe /home/user/work/primes.kmx primes.txt↵
```

Uvažujte všechny možné specifikace (uvedení úplné cesty, relativní cesty, atp.) jak vstupního, tak výstupního souboru, které jsou přípustné v daném operačním systému. Pokud nebude na příkazové řádce uveden alespoň jeden parametr (tj. jméno binárního spustitelného souboru ve formátu KMX), vypište chybové hlášení a stručný návod k použití programu (v angličtině). Návratová hodnota funkce `int main(...)` bude v tomto případě `1`.

Hotovou práci odevzdejte v jediném archivu typu ZIP prostřednictvím automatického odevzdávacího a validačního systému. Postupujte podle instrukcí uvedených na webu předmětu. Archiv nechť obsahuje všechny zdrojové soubory potřebné k přeložení programu, **makefile** pro Windows i Linux (pro překlad v Linuxu připravte soubor pojmenovaný `makefile` a pro Windows `makefile.win`) a dokumentaci ve formátu PDF vytvořenou v typografickém systému TEX, resp. LATEX. **Bude-li některá z částí chybět, validační systém Vaši práci odmítne.**

Podrobné informace k odevzdání práce najdete na webové stránce předmětu Programování v jazyce C na adrese URL `https://www.kiv.zcu.cz/studies/predmety/pc/index.php#work`.

Popis činnosti programu

Program funguje jako *emulátor*[4] níže popsaného teoretického počítače KM s procesorem architektury MISC. Načítá parametrem předaný binární spustitelný soubor pro počítač KM ve formátu KMX, dekóduje jej a získané informace (obsah datového a kódového segmentu) uloží do svých interních struktur. Následně vykoná instrukce z kódového segmentu (za případného využití dat z datového segmentu) tak, jako by je vykonal teoretický počítač KM (kdyby existoval).

Během vykonávání strojového kódu počítače KM emulátor upravuje hodnoty v interních strukturách (registrech, segmentech paměti, apod.) tak, aby vnitřní stav emulátoru neustále odpovídal stavu teoretického počítače v průběhu provádění předmětného programu.

Na konzoli emulátor v průběhu vykonávání strojového kódu počítače KM vypisuje pouze takové údaje (hodnoty), které jsou výstupem instrukcí, jež provádějí výstup na konzoli (tj. pouze instrukce `OUT`x, viz tabulka 2) – **nic jiného emulátor během své činnosti nevypisuje!** Pokud potřebujete během vývoje a ladění programu vypisovat nějaké diagnostické informace, čiňte tak do samostatného logovacího souboru na disku, nikoliv do konzole (kvůli správnému fungování automatického validačního systému[5]).

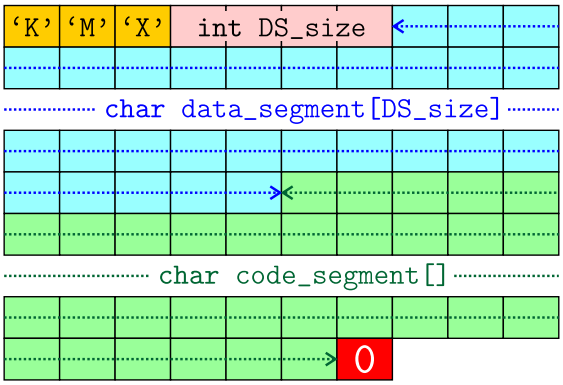
Implementace vnitřních struktur emulátoru není zadáním specifikována; je na libovůli programátora, jak bude emulátor realizovat (nicméně musí to samozřejmě fungovat).

[4] `https://cs.wikipedia.org/wiki/EmulÃĄtor`

[5] Validátor kontroluje výstup emulátoru na konzoli. Bude-li tedy váš program vypisovat něco, „co tam nepatří“, validátor bude zmatený a program nejspíš kontrolou neprojde. Logovací soubor (ve stejném adresáři, kde je spustitelný soubor emulátoru) validátoru nijak nevadí.

Popis formátu KMX spustitelného souboru pro emulátor

Soubor ve formátu KMX je určen k uložení programu ve strojovém kódu pro teoretický počítač KM. Struktura souboru je velmi jednoduchá a zachycuje ji obr. 1. Soubor je sekvence 8-bitových



Obrázek 1: Grafické znázornění uspořádání údajů v binárním souboru formátu KMX.

bytů (datový typ `char` jazyka C). Na začátku souboru je signatura 3 znaků '`K`', '`M`' a '`X`'. Tuto signaturu by měl emulátor při spuštění zkontrolovat – pokud v souboru není, hlásí emulátor chybu (anglický text chybového hlášení není předepsán, ale musí to být srozumitelné), ukončí se a předává návratovou hodnotu `2`.

Bezprostředně po signatuře následuje 32-bitové celé číslo (datový typ `int` jazyka C), které představuje délku datového segmentu v bytech. Poté je v souboru KMX uložen obsah datového segmentu programu. Za posledním bytem datového segmentu následuje první byte kódového segmentu. Kódový segment je ukončen bytem s hodnotou `0` – ale **pozor:** byte s hodnotou `0` se může vyskytovat i kdekoliv v datovém nebo kódovém segmentu (může tam být jako argument některé z instrukcí).

Popis teoretického počítače KM s procesorem MISC

Teoretický počítač KM, resp. jeho procesor, je 32-bitový, tzn. šířka registrů (všech bez výjimky) je 32 bitů. Všechna čísla jsou celá, 32-bitová v kódu s dvojkovým doplňkem, tj. odpovídají typu `signed long int` jazyka C. Jiné nativní datové typy tento procesor nezná. Endian procesoru je malý, tzn. nižší řády čísel jsou uloženy na nižších adresách v paměti (stejně jako u procesorů Intel 80x86).

Procesor KM nabízí programátorovi 2 všeobecné střadače (*Accumulators*) `A` a `B`, a jeden čítač `C` (= *Counter*). Dále jsou k dispozici dva indexové registry `S` (= *Source*) a `D` (= *Destination*), které slouží zejména k indexování dat v datovém segmentu (ale lze je použít i jako střadače).

Virtuální počítač je harvardské architektury, tzn. instrukce vykonávaného kódu jsou v samostatném kódovém segmentu (*Code Segment*, CS), zatímco data jsou v samostatném datovém segmentu (*Data Segment*, DS) paměti. Velikost CS i DS je 256 KB. Počítač má dále pro jednoduchost samostatný 16 KB segment pro zásobník (*Stack Segment*, SS). Pozice právě vykonávané instrukce v kódovém segmentu je určena obsahem registru `IP` (= *Instruction Pointer*), který ale není programátorovi přístupný (tedy neexistuje žádná instrukce, která by jeho obsah mohla přímo změnit, a tak nemůže být ani parametrem jakékoliv instrukce pro přesun dat). Pozici vrcholu zásobníku jednoznačně určuje obsah registru `SP` (= *Stack Pointer*). Ten lze měnit jako kterýkoliv jiný registr pomocí instrukcí `MOV`, ovšem s tím, že nesprávně provedená změna může mít katastrofální následky pro další vykonávání programu.

Každá instrukce strojového kódu našeho virtuálního počítače zabírá v paměti (v CS) právě jeden byte. Za tímto bytem, který specifikuje instrukci (podle tabulky 2), může (ale nemusí, podle

druhu instrukce) následovat jeden či více bytů parametrů dané instrukce. Typicky např. registr, který příslušná instrukce modifikuje, je specifikován dalším bytem v kódovém segmentu podle tabulky 1. Tzn. např. zápis instrukce v assembleru '`MOV B, 16`' bude převeden na strojový kód v podobě sekvence bytů (hexadecimálně) '`10 02 10 00 00 00`'. Byte '`10`' představuje tzv. *op-code* instrukce `MOV` (= *Move*, přesun; viz tabulka 2), byte '`02`' pak parametrické určení dotčeného registru, tedy `B`. Následují čtyři byty argumentu, tedy 32-bitového čísla s hodnotou 16.

Tabulka 1: Určení registru, je-li parametrem instrukce.

Registr	**Kód** (hexadecimálně)
`A`	`01`
`B`	`02`
`C`	`03`
`D`	`04`
`S`	`05`
`SP`	`06`

Instrukční sada

Instrukční sada procesoru teoretického počítače KM je úplně popsána tabulkou 2. V popisu každé instrukce může být užito následujících symbolů: (i) *im32* (= *immediate*) znamená 32-bitový argument bezprostředně následující v kódovém segmentu za příslušnou instrukcí; (ii) *reg* (= *register*) znamená, že za instrukcí následuje 1 byte určující, se kterým registrem bude instrukce provedena podle tabulky kódů registrů, tab. 1.

Tabulka 2: Instrukční sada teoretického procesoru KM architektury MISC.

Instrukce (zápis v assembleru)	**Kód** (byty hexa)	**Popis činnosti počítače při vykonání instrukce**
Pozastavení vykonávání a prázdná instrukce		
`HALT`	`00`	Procesor ukončí svoji činnost. Každý korektní program ve strojovém kódu by měl končit touto instrukcí. Pokud ale není poslední instrukcí kódu, nemělo by to vést k havárii.
`NOP`	`90`	Procesor vykoná tuto instrukci tím, že neprovede nic (kromě zvýšení `IP` o 1).
Instrukce pro přesun dat		
`MOV` *reg*`,` *im32*	`10` *reg im32*	Vloží bezprostředně následující 32-bitové číslo do registru *reg*.
`MOV` reg_d`,` reg_s	`11` reg_d reg_s	Vloží obsah registru reg_s do registru reg_d.
`MOVSD`	`12`	Načte 32-bitovou hodnotu z datového segmentu na adrese dané obsahem registru `S` a uloží ji do datového segmentu na adresu danou obsahem registru `D`, tj. `DS[D]` ← `DS[S]`.
`LOAD` *reg*`,` *im32*	`13` *reg im32*	Do registru *reg* uloží 32-bitové číslo, které se nachází v datovém segmentu na adrese dané operandem instrukce, tedy 32-bitovou hodnotou bezprostředně následující instrukci, tj. *reg* ← `DS[`*im32*`]`.
`LOAD` reg_d`,` reg_s	`14` reg_d reg_s	Do registru reg_d uloží 32-bitové číslo, které se nachází v datovém segmentu na adrese dané obsahem registru reg_s, tj. reg_d ← `DS[`reg_s`]`.
`STOR` *reg*`,` *im32*	`15` *reg im32*	Z registru *reg* vezme 32-bitové číslo a uloží ho do datového segmentu na adresu danou operandem instrukce, tedy 32-bitovou hodnotou bezprostředně následující instrukci, tj. `DS[`*im32*`]` ← *reg*.
`STOR` reg_s`,` reg_d	`16` reg_s reg_d	Z registru reg_s vezme 32-bitové číslo a uloží ho do datového segmentu na adresu danou obsahem registru reg_d, tj. `DS[`reg_d`]` ← reg_s.

Instrukce pro práci se zásobníkem		
`PUSH` *reg*	`20` *reg*	Zvýší hodnotu uloženou v registru `SP` (= *Stack Pointer*, ukazatel na pozici vrcholu zásobníku) o 4 a následně na nový vrchol uloží obsah registru *reg*, tj. `SP` ← `SP + 4`; `SS[SP]` ← *reg*.
`POP` *reg*	`21` *reg*	Přečte z vrcholu zásobníku 32-bitovou hodnotu a vloží ji do registru *reg*, načež sníží hodnotu registru `SP` o 4, tj. *reg* ← `SS[SP]`; `SP` ← `SP - 4`.
Aritmetické instrukce		
`ADD` *reg*, *im32*	`30` *reg im32*	Přičte k obsahu registru *reg* 32-bitovou hodnotu bezprostředně následující instrukci, tj. *reg* ← *reg* + *im32*.
`ADD` reg_d, reg_s	`31` reg_d reg_s	Přičte k obsahu registru reg_d obsah registru reg_s, tj. $reg_d \leftarrow reg_d + reg_s$.
`SUB` *reg*, *im32*	`32` *reg im32*	Odečte od obsahu registru *reg* 32-bitovou hodnotu bezprostředně následující instrukci, tj. *reg* ← *reg* − *im32*.
`SUB` reg_d, reg_s	`33` reg_d reg_s	Přičte k obsahu registru reg_d obsah registru reg_s, tj. $reg_d \leftarrow reg_d - reg_s$.
`MUL` *reg*, *im32*	`34` *reg im32*	Vynásobí obsah registru *reg* 32-bitovou hodnotou bezprostředně následující instrukci, tj. *reg* ← *reg* × *im32*.
`MUL` reg_d, reg_s	`35` reg_d reg_s	Vynásobí obsah registru reg_d obsahem registru reg_s, tj. $reg_d \leftarrow reg_d \times reg_s$.
`DIV` *reg*, *im32*	`36` *reg im32*	Vydělí obsah registru *reg* 32-bitovou hodnotou bezprostředně následující instrukci, tj. *reg* ← *reg* / *im32*.
`DIV` reg_d, reg_s	`37` reg_d reg_s	Vydělí obsah registru reg_d obsahem registru reg_s, tj. $reg_d \leftarrow reg_d / reg_s$.
`INC` *reg*	`38` *reg*	Zvýší obsah registru *reg* o 1, tj. *reg* ← *reg* + 1. Může dojít k přetečení (považuje se za běžný stav, nikoliv chybu).
`DEC` *reg*	`39` *reg*	Sníží obsah registru *reg* o 1, tj. *reg* ← *reg* − 1. Může dojít k podtečení (dtto).
Logické instrukce		
`AND` *reg*, *im32*	`40` *reg im32*	Provede logický součin (konjunkci) obsah registru *reg* s 32-bitovou hodnotou bezprostředně následující instrukci, tj. *reg* ← *reg* ∧ *im32*.
`AND` reg_d, reg_s	`41` reg_d reg_s	Provede logický součin (konjunkci) obsah registru reg_d s obsahem registru reg_s, tj. $reg_d \leftarrow reg_d \wedge reg_s$.
`OR` *reg*, *im32*	`42` *reg im32*	Provede logický součet (disjunkci) obsah registru *reg* s 32-bitovou hodnotou bezprostředně následující instrukci, tj. *reg* ← *reg* ∨ *im32*.
`OR` reg_d, reg_s	`43` reg_d reg_s	Provede logický součet (disjunkci) obsah registru reg_d s obsahem registru reg_s, tj. $reg_d \leftarrow reg_d \vee reg_s$.
`XOR` *reg*, *im32*	`44` *reg im32*	Provede exkluzivní logický součet (nonekvivalenci) obsah registru *reg* s 32-bitovou hodnotou bezprostředně následující instrukci, tj. *reg* ← *reg* ⊕ *im32*.
`XOR` reg_d, reg_s	`45` reg_d reg_s	Provede exkluzivní logický součet (nonekvivalenci) obsah registru reg_d s obsahem registru reg_s, tj. $reg_d \leftarrow reg_d \oplus reg_s$.
`NOT` *reg*	`46` *reg*	Nahradí obsah registru *reg* jeho jedničkovým doplňkem (čili inverzí všech bitů), tj. *reg* ← ¬*reg*.
Instrukce pro bitové manipulace		
`SHL` *reg*, *im32*	`50` *reg im32*	Posune obsah registru *reg* doleva o počet bitů daný 32-bitovou hodnotou bezprostředně následující instrukci (ovšem z této 32-bitové hodnoty se pro posun bere v potaz pouze 5 nejméně významných bitů), tj. *reg* ← *reg* `<<` *im32*.
`SHL` reg_d, reg_s	`51` reg_d reg_s	Posune obsah registru reg_d doleva o počet bitů daný hodnotou registru reg_s (ovšem z této 32-bitové hodnoty se pro posun bere v potaz pouze 5 nejméně významných bitů), tj. $reg_d \leftarrow reg_d$ `<<` reg_s.

`SHR` *reg*`,` *im32*	`52` *reg im32*	Posune obsah registru *reg* doprava o počet bitů daný 32-bitovou hodnotou bezprostředně následující instrukci (ovšem z této 32-bitové hodnoty se pro posun bere v potaz pouze 5 nejméně významných bitů), tj. *reg* ← *reg* `>>` *im32*.
`SHR` reg_d`,` reg_s	`53` reg_d reg_s	Posune obsah registru reg_d doprava o počet bitů daný hodnotou registru reg_s (ovšem z této 32-bitové hodnoty se pro posun bere v potaz pouze 5 nejméně významných bitů), tj. $reg_d \leftarrow reg_d$ `>>` reg_s.
Porovnávací instrukce		
`CMP` *reg*`,` *im32*	`60` *reg im32*	Porovná obsah registru *reg* s 32-bitovou hodnotou bezprostředně následující instrukci a nastaví vnitřní příznaky procesoru tak, aby šlo podle výsledku porovnání provést skok, tj. `if (`*reg* = *im32*`)` ...
`CMP` reg_1`,` reg_2	`61` reg_1 reg_2	Porovná obsah registru reg_1 s obsahem registru reg_2 a nastaví vnitřní příznaky procesoru tak, aby šlo podle výsledku porovnání provést skok, tj. `if (`$reg_1 = reg_2$`)` ...
Instrukce skoků		
`JMP` *im32*	`70` *im32*	Provede skok v kódovém segmentu na adresu danou 32-bitovou hodnotou bezprostředně následující instrukci, tj. `IP` ← *im32*.
`JMP` *reg*	`71` *reg*	Provede skok v kódovém segmentu na adresu danou obsahem registru *reg*, tj. `IP` ← *reg*.
`JE` *im32*	`72` *im32*	Provede **relativní** skok v kódovém segmentu směrem k vyšším (při kladné hodnotě *im32*) či nižším (při záporné hodnotě *im32*) adresám o 32-bitovou hodnotou bezprostředně následující instrukci tehdy, **je-li výsledkem předchozí instrukce `CMP` rovnost** (= *Jump if Equal*), tj. `if (...) IP` ← `IP` + *im32*.
`JNE` *im32*	`73` *im32*	Provede relativní skok v kódovém segmentu o 32-bitovou hodnotou bezprostředně následující instrukci tehdy, **je-li výsledkem předchozí instrukce `CMP` nerovnost** (= *Jump if Not Equal*), tj. `if (...) IP` ← `IP` + *im32*.
`JG` *im32*	`74` *im32*	Provede relativní skok v kódovém segmentu o 32-bitovou hodnotou bezprostředně následující instrukci tehdy, **byl-li při provedení předchozí instrukce `CMP` první operand větší než druhý** (= *Jump if Greater*), tj. `if (...) IP` ← `IP` + *im32*.
`JGE` *im32*	`75` *im32*	Provede relativní skok v kódovém segmentu o 32-bitovou hodnotou bezprostředně následující instrukci tehdy, **byl-li při provedení předchozí instrukce `CMP` první operand větší nebo roven druhému** (= *Jump if Greater or Equal*), tj. `if (...) IP` ← `IP` + *im32*.
`JNG` *im32*	`76` *im32*	Provede relativní skok v kódovém segmentu o 32-bitovou hodnotou bezprostředně následující instrukci tehdy, **nebyl-li při provedení předchozí instrukce `CMP` první operand větší než druhý** (= *Jump if Not Greater*), tj. `if (...) IP` ← `IP` + *im32*.
`JL` *im32*	`77` *im32*	Provede relativní skok v kódovém segmentu o 32-bitovou hodnotou bezprostředně následující instrukci tehdy, **byl-li při provedení předchozí instrukce `CMP` první operand menší než druhý** (= *Jump if Less*), tj. `if (...) IP` ← `IP` + *im32*.
`JLE` *im32*	`78` *im32*	Provede relativní skok v kódovém segmentu o 32-bitovou hodnotou bezprostředně následující instrukci tehdy, **byl-li při provedení předchozí instrukce `CMP` první operand menší nebo roven druhému** (= *Jump if Less or Equal*), tj. `if (...) IP` ← `IP` + *im32*.
`JNL` *im32*	`79` *im32*	Provede relativní skok v kódovém segmentu o 32-bitovou hodnotou bezprostředně následující instrukci tehdy, **nebyl-li při provedení předchozí instrukce `CMP` první operand menší než druhý** (= *Jump if Not Less*), tj. `if (...) IP` ← `IP` + *im32*.

Instrukce na podporu podprogramů		
`CALL` *im32*	`80` *im32*	Předá řízení podprogramu, jehož vstupní bod (počáteční adresa v CS, tj. adresa první instrukce tohoto podprogramu) je dán 32-bitovou hodnotou bezprostředně následující instrukci. Oproti instrukci `JMP` musí `CALL` před skokem na danou adresu uložit do zásobníku tzv. návratovou adresu, čili adresu instrukce, která bezprostředně následuje po této instrukci `CALL`, tj. `PUSH (IP + 5)`; `IP` ← *im32*.
`CALL` *reg*	`81` *reg*	Předá řízení podprogramu, jehož vstupní bod je dán hodnotou v registru *reg*. Oproti instrukci `JMP` musí `CALL` před skokem na danou adresu uložit do zásobníku tzv. návratovou adresu, čili adresu instrukce, která bezprostředně následuje po této instrukci `CALL`, tj. `PUSH (IP + 2)`; `IP` ← *reg*.
`RET`	`82`	Vrátí řízení z podprogramu zavolaného instrukcí `CALL` volajícímu (pod)programu. Adresu pro návrat získá instrukce `RET` ze zásobníku, kam ji uložila instrukce `CALL`, tj. `POP` *val*; `IP` ← *val*.
Emulátorové instrukce pro vstup a výstup údajů		
`OUTD` *reg*	`F0` *reg*	Vypíše na konzoli emulátoru jako celé dekadické číslo hodnotu uloženou v registru *reg*.
`OUTC` *reg*	`F1` *reg*	Vypíše na konzoli emulátoru jeden znak, jehož ASCII hodnota je umístěna v registru *reg* (pro výpis znaku se bere v úvahu pouze nejméně významných 8 bitů).
`OUTS` *reg*	`F2` *reg*	Vypíše na konzoli emulátoru řetězec znaků (sekvence ASCII znaků ukončený znakem s ASCII hodnotou `0`), jehož adresa (počátku) v DS je umístěna v registru *reg*.
`INPD` *reg*	`F3` *reg*	Načte z konzole emulátoru celé dekadické číslo a to uloží do registru *reg*.
`INPC` *reg*	`F4` *reg*	Načte z konzole emulátoru jeden znak a ten uloží (jeho ASCII hodnotu) do registru *reg*.
`INPS` *reg*	`F5` *reg*	Načte z konzole emulátoru řetězec znaků (sekvence ASCII znaků ukončený znakem s ASCII hodnotou `0`) a ten uloží do DS na adresu určenou obsahem registru *reg*.

Pokud si u některé instrukce nejste z popisu jisti, jak se přesně má chovat, prozkoumejte chování takové (nebo podobné instrukce) u některého existujícího procesoru, např. Intel 8086 (kterým je instrukční sada procesoru KM silně inspirovaná).

Specifikace výstupu programu

Program je konzolová aplikace, tj. ovládá se pouze parametry předanými při spuštění v příkazové řádce (konzoli/terminálu). V průběhu vykonávání strojového kódu emulovaného teoretického počítače KM se žádná interakce s uživatelem nepředpokládá, s výjimkou situace, kdy emulátor provádí některou z instrukcí `OUT`x nebo `INP`x.

V případě provádění instrukce `OUT`x je výstup směřován do konzole a ukončen vždy znakem (resp. sekvencí) pro novou řádku (‘`\n`’). Pokud byl při spuštění emulátoru druhým nepovinným parametrem specifikován výstupní soubor, pak je výstup ukládán do tohoto souboru a také je za každý vypsaný údaj (ať je to číslo, znak či řetězec znaků) připojen znak pro novou řádku.

Při zpracování instrukce `INP`x je potřebný údaj od uživatele načten z konzole např. prostředky funkce `int scanf(...)`. Načítá-li se číslo (instrukce `INPD`) nebo řetězec (instrukce `INPS`), očekává emulátor ukončení uživatelského vstupu stisknutím klávesy ENTER. Pokud se načítá pouze jediný znak instrukcí `INPC`, je uživatelský vstup ukončen ihned po zadání příslušného znaku (tedy po stisknutí jakékoliv jedné klávesy odpovídající tisknutelnému znaku).

Pokud nastane závažný chybový stav, který nedovoluje emulátoru pokračovat v činnosti, oznamuje to uživateli srozumitelným chybovým hlášením v anglickém jazyce vypsaným na konzoli

(do standardního proudu `stderr`) a následně vynuceným ukončením programu s předáním návratové hodnoty podle tabulky 3. Jiným než výše popsaným způsobem emulátor během své činnosti s uživatelem neinteraguje.

Tabulka 3: Návratové kódy programu v případě chyby.

Popis chybového stavu	**Návratová hodnota funkce `int main()`**
Bez chyby/korektní ukončení činnosti emulátoru.	`0`
Nebyly předány správné parametry na příkazové řádce při spuštění emulátoru.	`1`
Soubor určený prvním parametrem při spuštění emulátoru není ve formátu KMX.	`2`
Emulátor narazil při vykonávání strojového kódu na neplatnou instrukci (op-code, který není uveden v tab. 2).	`3`
Emulátor provedl operaci s katastrofálním následkem (dělení nulou, skok na adresu mimo rozsah adresního prostoru, atp.).	`4`
Ostatním chybovým stavům můžete přiřadit návratové kódy dle svého uvážení.	

Užitečné informace

Níže uvedené zdroje informací je možné (ale nikoliv nezbytně nutné) využít při řešení úlohy. Protože emulace je postup v informatice víceméně velice běžný, lze k němu nalézt velké množství různé dokumentace např. za pomoci vyhledávače Google:

1. Informace o emulátorech na Wikipedii – `https://en.wikipedia.org/wiki/Emulator`
2. Emulator 101 – `http://www.emulator101.com`
3. Fantasy CPU Emulator – `https://codeguppy.com/blog/fantasy-cpu-emulator/index.html`
4. Instrukční sada CPU Intel 8086 – `https://en.wikipedia.org/wiki/X86_instruction_listings`
5. Michal Brandejs: *Mikroprocesory Intel 8086 – 80486*. Grada, 1991. ISBN 80-85424-27-4.

Řešení úlohy je zcela ve vaší kompetenci – zvolte takové algoritmy a techniky, které podle vás nejlépe povedou k cíli.

Příloha: Ukázka assembleru a strojového kódu počítače KM

V níže uvedené ukázce je naprogramován v assembleru počítače KM algoritmus řazení *Bubble Sort*. V komenáři za každou instrukcí je její podoba ve strojovém kódu (po bytech, hexadecimálně).

```
.KMA
.DATA
    swapped DWORD ?             ; items swapped in the inner loop, DS:[0]
    array DWORD 20 DUP(?)       ; the data to be sorted, DS:[4]
.CODE
    MOV   A, 1                  ; 10 01 01 00 00 00
    STOR  A, OFFSET swapped     ; 15 01 00 00 00 00
@LBubbleWhileBeg:               ; <--- CS:[12]
    LOAD  A, OFFSET swapped     ; 13 01 00 00 00 00
    CMP   A, 1                  ; 60 01 01 00 00 00
    JNE   @LBubbleWhileEnd      ; 73 62 00 00 00
@LBubbleForInit:                ; <--- CS:[29]
    MOV   C, 19                 ; 10 03 13 00 00 00
    MOV   A, 0                  ; 10 01 00 00 00 00
    STOR  A, OFFSET swapped     ; 15 01 00 00 00 00
    MOV   S, OFFSET array       ; 10 05 04 00 00 00
    MOV   D, OFFSET array       ; 10 04 04 00 00 00
    ADD   D, 4                  ; 30 04 04 00 00 00
@LBubbleForBeg:                 ; <--- CS:[65]
    LOAD  A, S                  ; 14 01 05
    LOAD  B, D                  ; 14 02 04
    CMP   A, B                  ; 61 01 02
    JNG   @L01                  ; 76 12 00 00 00
    STOR  B, S                  ; 16 02 05
    STOR  A, D                  ; 16 01 04
    MOV   A, 1                  ; 10 01 01 00 00 00
    STOR  A, OFFSET swapped     ; 15 01 00 00 00 00
@L01:                           ; <--- CS:[97]
    DEC   C                     ; 39 03
    ADD   S, 4                  ; 30 05 04 00 00 00
    ADD   D, 4                  ; 30 04 04 00 00 00
    CMP   C, 0                  ; 60 03 00 00 00 00
    JG    @LBubbleForBeg        ; 74 C7 FF FF FF
@LBubbleForEnd:                 ; <--- CS:[122]
    JMP   @LBubbleWhileBeg      ; 70 0C 00 00 00
@LBubbleWhileEnd:               ; <--- CS:[127]

@LOutputForInit:                ; <--- CS:[127]
    MOV   C, 20                 ; 10 03 14 00 00 00
    MOV   S, OFFSET array       ; 10 05 04 00 00 00
@LOutputForBeg:                 ; <--- CS:[139]
    LOAD  A, S                  ; 14 01 05
    OUTD  A                     ; F0 01
    ADD   S, 4                  ; 30 05 04 00 00 00
    DEC   C                     ; 39 03
    CMP   C, 0                  ; 60 03 00 00 00 00
    JG    @LOutputForBeg        ; 74 E8 FF FF FF
@LOutputForEnd:                 ; <--- CS:[157]
    HALT                        ; 00
```