


ZADÁNÍ SEMESTRÁLNÍ PRÁCE

ASSEMBLER PROCESORU ARCHITEKTURY MISC

Zadání

Naprogramujte v ANSI C přenositelnou¹ **konzolovou aplikaci**, která bude zajišťovat funkcionality velmi jednoduchého (ale plně funkčního) **překladače jazyka symbolických adres (JSA) teoretického/virtuálního počítače** – nazvěme ho třeba K’s Machine (KM) – s procesorem architektury MISC (= Minimal Instruction Set Computer). Vámi vyvinutý program zpracuje zdrojový soubor zapsaný v JSA (nebo krátce a zjednodušeně *assembly*) procesoru KM a vygeneruje binární spustitelný soubor ve formátu KMX (K’s Machine eXecutable) obsahující odpovídající sekvenci instrukcí strojového kódu níže popsaného minimalistického teoretického/virtuálního počítače².

Assembler se bude spouštět příkazem

```
X:\>kmas.exe3 <program_v_jsa.kas>[<spustitelný_soubor[.kmx]>] [-v] [-i] 
```

Symbol `<program_v_jsa>` představuje povinný parametr – název vstupního zdrojového souboru v jazyce symbolických adres (assembly) procesoru KM. Přípona `.kas` (= K’s Machine Assembly), identifikující soubor se zdrojovým kódem v JSA procesoru KM, musí být vždy uvedena.

Symbol `<spustitelný_soubor>` pak představuje nepovinný parametr, kterým je jméno výstupního binárního spustitelného souboru, do kterého se bude ukládat přeložená sekvence strojových instrukcí procesoru KM. Není-li druhý, nepovinný, parametr uveden, nechť program směřuje výstup do souboru pojmenovaného stejně jako vstupní soubor, ovšem s příponou `.kmx` (místo přípony `.kas` vstupního souboru).

Nepovinný přepínač ‘-v’ (= verbose) slouží k aktivaci režimu intenzivního výpisu diagnostických informací. Co přesně bude assembler v tomto režimu vypisovat, je zcela na programátorovi a není to touto specifikací předepsáno.

Nepovinný přepínač ‘-i’ (= instructions) slouží k aktivaci režimu výpisu každé jednotlivé přeložené instrukce ze zdrojového textu programu. Formát výpisu není předepsán a slouží zejména k tomu, aby si programátor mohl zkontrolovat, že jím vyvíjený assembler pracuje správně. Doporučený formát výpisu vypadá takto: ‘L50: DEC C at CS:150’. Znamená, že na řádce 50 našel assembler instrukci ‘DEC’, jejímž parametrem je registr ‘C’ a opkód této instrukce uložil do kódového segmentu výsledného spustitelného programu na adresu 150. Můžete ale zvolit i jiný formát, validátor přesnou podobu výpisu nekontroluje.

Úkolem Vámi vyvinutého programu tedy bude:

1. Načíst prvním parametrem určený vstupní soubor se zdrojovým kódem programu v JSA procesoru KM.
2. Zpracovat všechny deklarační příkazy a podle jejich konkrétní podoby vyhradit odpovídající počet bytů (tedy provést alokaci paměti) v datovém segmentu spustitelného souboru formátu KMX.

¹Je 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 5.10.0-32-amd64 a s překladačem gcc 10.2.1-6.

²Návrh tohoto teoretického/virtuálního 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.

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

3. Uložit adresy všech objektů v datovém segmentu do vnitřních struktur assembleru tak, aby bylo později možné nahradit symbolické vyjádření adres (návěští) objektů/cílových pozic skoků v parametrech strojových instrukcí konkrétními adresami v datovém/kódovém segmentu.
4. Zpracovat všechny instrukce programu, přeložit je na sekvence bytů s odpovídajícími opkódy následované případnými parametry (absolutními hodnotami či signaturami registrů).
5. Uložit sekvence strojových instrukcí do kódového segmentu.
6. Po sestavení obou segmentů (kódového i datového) vypočítat absolutní adresy skoků v kódovém a objektů v datovém segmentu a všechny symbolické adresy (návěští) nahradit konkrétními absolutními adresami.
7. Sestavené segmenty uložit do výstupního souboru ve formátu KMX a doplnit údaje do jeho hlavičky, tak aby vzniknul korektní, emulátorem počítače KM vykonatelný spustitelný soubor.

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

```
X:\>kmas.exe "E:\My Work\Test\mersenne.kas" mersenne.kmx ↵
```

nebo takto:

```
X:\>kmas.exe "E:\My Work\Test\primes.kas" build\primes.kmx ↵
```

nebo pouze takto:

```
X:\>kmas.exe bubble.kas ↵
```

První z uvedených příkladů spuštění by měl vést k vytvoření výstupního spustitelného souboru `mersenne.kmx` v tomtéž adresáři, kde se nachází zdrojový soubor `mersenne.kas`. Druhý příklad vytvoří výstupní spustitelný soubor `primes.kmx` v adresáři `build`, který je podadresářem adresáře, ve kterém se nachází zdrojový soubor `primes.kas`. Při vykonání třetího příkladu bude výstup směřován do souboru `bubble.kmx`, umístěného ve stejném adresáři jako zdrojový soubor.

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

```
$/kmas.exe /home/user/work/primes.kas primes.kmx ↵
```

Uvažujte všechny možné specifikace (uvedení úplné cesty, relativní cesty, atp.) jak vstupního, tak nepovinného výstupního souboru, které jsou přípustné v daném operačním systému. Pokud nebude na příkazové řádce uveden alespoň jeden parametr (tj. jméno vstupního souboru se zdrojovým kódem programu v assembleru procesoru KM), vypíše chybové hlášení a stručný návod k použití programu (v angličtině). Návrátová 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 nechtě 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 $\text{T}_{\text{E}}\text{X}$, resp. $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$. **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 *assembler*⁴ níže popsaného teoretického počítače KM s procesorem architektury MISC. Načítá parametrem předaný vstupní soubor se zdrojovým kódem v JSA počítače KM, provede jeho lexikálně-syntaktickou analýzu a tou získané informace (potřebné pro sestavení obsahu datového a kódového segmentu budoucího spustitelného souboru) převede na sekvenci bytů datového segmentu a sekvenci bytů odpovídající instrukcím strojového kódu a jejich parametrům. Obě binární sekvence následně uloží do výstupního souboru ve formátu KMX, popsaném v sekci Popis formátu KMX na str. 5, který je vykonatelný teoretickým počítačem KM (kdyby existoval) nebo jeho emulátorem.

Assembler při překladu analyzuje jednotlivé příkazy (v částech⁵ definujících obsah datového segmentu začínajících klíčovým slovem `‘.DATA’`) a strojové instrukce (v částech definujících obsah kódového segmentu začínajících klíčovým slovem `‘.CODE’`⁶). Komentáře se mohou vyskytovat kdekoli ve zdrojovém textu, uvozeny jsou znakem `‘;’` (středník) a končí vždy koncem řádky, tj. nemohou být vloženy „dovnitř“ kódu. Assembler komentáře při překladu ignoruje.

Příkazy definující obsah datového segmentu assembler přímo **provádí**, a tím modifikuje velikost a obsah datového segmentu budoucího spustitelného souboru, který je v průběhu překladu udržován v interních strukturách assembleru.

Příkazy pro definici obsahu datového segmentu

V popisu datového segmentu (sekce začíná vždy klíčovým slovem `‘.DATA’`, končí začátkem jiné sekce, a v programu se může vyskytnout vícekrát) lze opakovaně použít pouze následující příkazy definující objekty:

Tabulka 1: Příkazy pro definici objektů v datovém segmentu.

Příkaz	Popis
DWORD	Definuje proměnnou typu <i>double word</i> , tj. 4-bytové/32-bitové celé číslo v kódu s dvojkovým doplňkem, tj. odpovídající typu <code>signed long int</code> jazyka C.
DW	Zkrácený zápis příkazu DWORD.
BYTE	Definuje proměnnou typu <i>byte</i> , tj. 1-bytové/8-bitové celé číslo v kódu s dvojkovým doplňkem, tj. odpovídající typu <code>char</code> jazyka C.
DB	Zkrácený zápis příkazu BYTE.
DUP	Příkaz pro opakované (duplicate) vložení hodnoty do pole prvků daného typu.

Jiné primitivní datové typy než 8-bitový byte deklarovaný příkazem `‘BYTE’` a 32-bitové znaménkové celé číslo deklarované příkazem `‘DWORD’` procesor KM nepodporuje (viz sekce Popis počítače KM na str. 5)!

Příklady a jejich vysvětlení:

- (i) `var1 DWORD ?` — vytváří v datovém segmentu proměnnou s názvem `var1` o šířce 32 bitů, hodnota není určena, tj. proměnná není inicializovaná. Znak `?` za deklarátorem datového typu znamená, že proměnná není inicializovaná a může obsahovat libovolnou hodnotu.
- (ii) `arr1 DW 0, 1, 2, 3, 4` — vytváří v datovém segmentu pole 32-bitových celých čísel s názvem `arr1`, prvky pole jsou čísla 0, 1, 2, 3 a 4. Za deklarátorem datového typu může následovat výčet hodnot (které odpovídají danému datovému typu) oddělených čárkami, které inicializují jednotlivé prvky deklarovaného pole.

⁴https://en.wikipedia.org/wiki/Assembly_language

⁵V jednom zdrojovém kódu jich může být více a také tam nemusí být žádná.

⁶V jednom zdrojovém kódu jich může být více a vždy tam musí být alespoň jedna.

(iii) `arr2 DWORD 100 DUP(?)` — vytváří v datovém segmentu pole 32-bitových celých čísel o 100 prvcích s názvem `arr2`, prvky pole nejsou inicializované. Příkaz *počet-opakování* `DUP(hodnota)` určuje velikost pole daného bazového typu a hodnotu, kterou budou všechny prvky pole inicializovány. Význam je tedy vlastně *počet-opakování* × **zDUP**likuj zadanou hodnotu.

(iv) `str1 DB "Hello, world!", 0` — vytváří v datovém segmentu pole znaků inicializované jako znakový řetězec ukončený znakem s ASCII hodnotou 0 (tj. tzv. *null-terminated string*, který slouží k ukládání znakových řetězců v jazyce C). Inicializace může být provedena buď čárkami odděleným seznamem hodnot nebo v uvozovkách uzavřenými řetězci znaků (každý z nich reprezentuje jeden byte výsledného pole). Obě techniky inicializace lze libovolně kombinovat (viz ukázka níže).

(v) `str2 BYTE 20 DUP(0), 1` — vytváří v datovém segmentu pole 20 znaků inicializovaných hodnotou 0, za těmito 20 byty bude bezprostředně následovat jeden byte s hodnotou 1. Inicializátory lze tedy různě kombinovat, každý inicializátor je pak od následujícího oddělen čárkou.

Ukázka definice datového segmentu:

```
1 .KMA
2 .DATA
3     x DWORD ?           ; proměnná 'x' typu 32-bitové číslo, neinicializovaná
4     y DW 10             ; proměnná 'y' typu 32-bitové číslo, inicializovaná na hodnotu 10
5     z DW 100 DUP(65)    ; pole 100 čísel, inicializované na hodnotu 65
6     z2 DW 1, 2, 3, 4, 5 ; pole 5 čísel s hodnotami 1, 2, 3, 4 a 5
7     arr DWORD 20 DUP(?) ; pole 20 čísel, neinicializované
8     msg DB "Hello, world!", 13, 10, 0 ; řetězec následovaný znaky CR+LF a #0
9     grt DB "Hi!", 13, 10, "Hello!", 13, 10, 0 ; řetězec se dvěma řádkami
10 .CODE
11     MOV     A, 1
12     STOR    A, OFFSET x
13     ...
```

Popis datového segmentu končí začátkem jiného segmentu (tedy zřejmě kódového). Jakmile se tedy na následující řádce vyskytne příkaz `‘.CODE’`, definice datového segmentu končí a začíná definice kódového segmentu. Definic obou segmentů (datového i kódového) může být ve zdrojovém textu programu v JSA více, mohou se libovolně střídát, při sestavení výstupního spustitelného programu assemblerem jsou všechny případně oddělené definice jednotlivých segmentů spojeny v jediný datový a jediný kódový segment. Ty jsou následně uloženy do výstupního souboru ve formátu KMX.

Příkazy pro definici obsahu kódového segmentu

V sekcích definujících obsah kódového segmentu⁷ uvozených klíčovým slovem `‘.CODE’` se nacházejí zejména instrukce procesoru KM následované svými případnými parametry. Přehled všech instrukcí teoretického/virtuálního procesoru KM a jejich možných parametrů najdete v sekci Popis teoretického počítače KM na str. 5.

Kromě instrukcí se ve zdrojovém textu kódového segmentu mohou vyskytovat tzv. *návěští*, tedy označení míst, na která je možno skákat prostřednictvím instrukcí podmíněných a nepodmíněných skoků. Návěští mají pevně daný formát: Vždy začínají znakem `‘@’` (zavináč), za kterým následuje identifikátor, který splňuje požadavky, kladené na identifikátory v jazyce C, tj. může obsahovat velká i malá písmena, číslice (přičemž číslicí nesmí začínat) a znak `‘_’` (podtržítko).

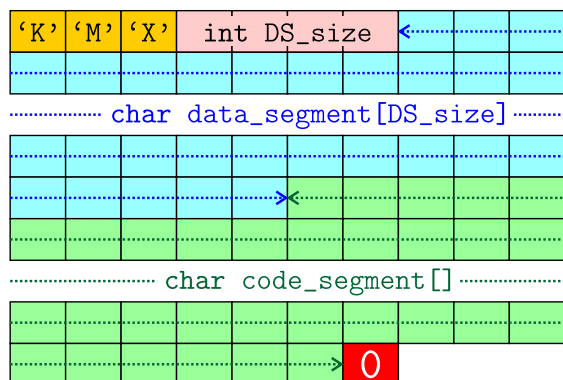
Jediným dalším speciálním příkazem, který se může vyskytnout ve zdrojovém textu kódového segmentu, je `‘OFFSET’`: Tento příkaz zajišťuje dodání adresy svého argumentu, kterým musí být objekt z datového segmentu. Např. příkaz JSA ve tvaru `‘MOV A, OFFSET var1’` znamená, že se do registru `‘A’` vloží adresa proměnné `‘var1’` v datovém segmentu, tj. počet bytů od počátku datového segmentu k pozici, kde je uložen první z bytů kódujících tuto proměnnou.

⁷Opět platí, že ve zdrojovém textu programu může být těchto sekcí více, na různých místech.

Ukázku rozsáhlejšího kódu v JSA procesoru KM najdete v příloze Ukázka JSA a strojového kódu počítače KM na str. 11.

Popis formátu KMX spustitelného souboru počítače KM

Soubor ve formátu KMX (= K's Machine Executable) je určen k uložení vykonatelného programu ve strojovém kódu pro teoretický počítač KM. Struktura souboru je velmi jednoduchá a zachycuje ji obr. 1. Soubor je sekvenční 8-bitových bytů (datový typ `char` jazyka C). Na začátku souboru je



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

signatura 3 znaků 'K', 'M' a 'X'. Přítomnost této signatury kontroluje emulátor počítače KM při spuštění programu – pokud v souboru není, hlásí emulátor chybu a ukončí se.

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 bezprostředně následuje první byte kódového segmentu. Kódový segment je ukončen bytem s hodnotou 0 – ale **pozor**: byte s hodnotou 0 se může vyskytovat i kdekoli v datovém nebo kódovém segmentu (může tam být jako argument některé z instrukcí).

Popis teoretického/virtuálního počítače KM s procesorem architektury MISC

Teoretický počítač KM, resp. jeho procesor, je 32-bitový, tzn. šířka registrů (všech bez výjimky) je 32 bitů. Všechna čísla, která lze ukládat do registrů a zpracovávat instrukcemi procesoru, 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 jakékoli 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 3), 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 2. 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. *opcode* instrukce **MOV** (= *Move*, přesun; viz tabulka 3), byte **02** pak parametrické určení dotčeného registru, tedy **B**. Následují čtyři byty argumentu, tedy 32-bitového čísla s hodnotou 16.

Tabulka 2: 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 3. 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. 2.

Tabulka 3: Instrukční sada teoretického procesoru KM architektury MISC.

Instrukce (zápis v JSA)	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 <i>reg, im32</i>	10 <i>reg im32</i>	Vloží bezprostředně následující 32-bitové číslo do registru <i>reg</i> .
MOV <i>reg_d, reg_s</i>	11 <i>reg_d reg_s</i>	Vloží obsah registru <i>reg_s</i> do registru <i>reg_d</i> .
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. $DS[D] \leftarrow DS[S]$.
LOAD <i>reg, im32</i>	13 <i>reg im32</i>	Do registru <i>reg</i> uloží 32-bitové číslo, které se nachází v datovém segmentu na adrese dané operandem instrukce, tedy 32-bitovou hodnotou bezprostředně následující instrukci, tj. $reg \leftarrow DS[im32]$.
LOAD <i>reg_d, reg_s</i>	14 <i>reg_d reg_s</i>	Do registru <i>reg_d</i> uloží 32-bitové číslo, které se nachází v datovém segmentu na adrese dané obsahem registru <i>reg_s</i> , tj. $reg_d \leftarrow DS[reg_s]$.
STOR <i>reg, im32</i>	15 <i>reg im32</i>	Z registru <i>reg</i> vezme 32-bitové číslo a uloží ho do datového segmentu na adresu danou operandem instrukce, tedy 32-bitovou hodnotou bezprostředně následující instrukci, tj. $DS[im32] \leftarrow reg$.

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. $DS[reg_d] \leftarrow reg_s$.
Instrukce pro práci se zásobníkem		
PUSH reg	20 reg	Zvýší hodnotu uloženou v registru SP (= <i>Stack Pointer</i> , ukazatel na pozici vrcholu zásobníku) o 4 a následně na nový vrchol uloží obsah registru reg , tj. $SP \leftarrow SP + 4$; $SS[SP] \leftarrow reg$.
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. $reg \leftarrow SS[SP]$; $SP \leftarrow SP - 4$.
Aritmetické instrukce		
ADD $reg, im32$	30 $reg im32$	Přičte k obsahu registru reg 32-bitovou hodnotu bezprostředně následující instrukci, tj. $reg \leftarrow reg + im32$.
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. $reg \leftarrow reg - im32$.
SUB reg_d, reg_s	33 $reg_d reg_s$	Odečte od 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. $reg \leftarrow reg \times im32$.
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. $reg \leftarrow reg / im32$.
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. $reg \leftarrow reg + 1$. Může dojít k přetečení (považuje se za běžný stav, nikoliv chybu).
DEC reg	39 reg	Sníží obsah registru reg o 1, tj. $reg \leftarrow reg - 1$. 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. $reg \leftarrow reg \wedge im32$.
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. $reg \leftarrow reg \vee im32$.
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. $reg \leftarrow reg \oplus im32$.
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. $reg \leftarrow \neg reg$.
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. $reg \leftarrow reg \ll im32$.

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. $reg_d \leftarrow reg_d \ll reg_s$.
SHR $reg, im32$	52 $reg im32$	Posune obsah registru reg doprava o počet bitů daný 32-bitovou hodnotou bezprostředně následující instrukcí (ovšem z této 32-bitové hodnoty se pro posun bere v potaz pouze 5 nejméně významných bitů), tj. $reg \leftarrow reg \gg im32$.
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. $reg_d \leftarrow reg_d \gg reg_s$.
Porovnávací instrukce		
CMP $reg, im32$	60 $reg im32$	Porovná obsah registru reg s 32-bitovou hodnotou bezprostředně následující instrukcí 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 instrukcí, tj. $IP \leftarrow im32$.
JMP reg	71 reg	Provede skok v kódovém segmentu na adresu danou obsahem registru reg , tj. $IP \leftarrow reg$.
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 ($= \text{Jump if Equal}$), tj. if (...) $IP \leftarrow IP + im32$.
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 ($= \text{Jump if Not Equal}$), tj. if (...) $IP \leftarrow IP + im32$.
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 ($= \text{Jump if Greater}$), tj. if (...) $IP \leftarrow IP + im32$.
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 ($= \text{Jump if Greater or Equal}$), tj. if (...) $IP \leftarrow IP + im32$.
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 ($= \text{Jump if Not Greater}$), tj. if (...) $IP \leftarrow IP + im32$.
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 ($= \text{Jump if Less}$), tj. if (...) $IP \leftarrow IP + im32$.
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 ($= \text{Jump if Less or Equal}$), tj. if (...) $IP \leftarrow IP + im32$.

JNL <i>im32</i>	79 <i>im32</i>	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 (= <i>Jump if Not Less</i>), tj. if (...) $IP \leftarrow IP + im32$.
Instrukce na podporu podprogramů		
CALL <i>im32</i>	80 <i>im32</i>	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. $PUSH (IP + 5)$; $IP \leftarrow im32$.
CALL <i>reg</i>	81 <i>reg</i>	Předá řízení podprogramu, jehož vstupní bod je dán hodnotou v registru <i>reg</i> . 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. $PUSH (IP + 2)$; $IP \leftarrow reg$.
RET	82	Vrátí řízení z podprogramu zvaného instrukcí CALL volajícím (pod)programu. Adresu pro návrat získá instrukce RET ze zásobníku, kam ji uložila instrukce CALL, tj. $POP val$; $IP \leftarrow val$.
Emulátorové instrukce pro vstup a výstup údajů		
OUTD <i>reg</i>	F0 <i>reg</i>	Vypíše na konzoli emulátoru jako celé dekadické číslo hodnotu uloženou v registru <i>reg</i> .
OUTC <i>reg</i>	F1 <i>reg</i>	Vypíše na konzoli emulátoru jeden znak, jehož ASCII hodnota je umístěna v registru <i>reg</i> (pro výpis znaku se bere v úvahu pouze nejméně významných 8 bitů).
OUTS <i>reg</i>	F2 <i>reg</i>	Vypíše na konzoli emulátoru řetězec znaků (sekvence ASCII znaků ukončený znakem s ASCII hodnotou 0), jehož adresa (počátku) v DS je umístěna v registru <i>reg</i> .
INPD <i>reg</i>	F3 <i>reg</i>	Načte z konzole emulátoru celé dekadické číslo a to uloží do registru <i>reg</i> .
INPC <i>reg</i>	F4 <i>reg</i>	Načte z konzole emulátoru jeden znak a ten uloží (jeho ASCII hodnotu) do registru <i>reg</i> .
INPS <i>reg</i>	F5 <i>reg</i>	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 <i>reg</i> .

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 překladač zdrojového kódu v JSA do strojového kódu emulovaného teoretického počítače KM se žádná interakce s uživatelem nepředpokládá, kromě případných diagnostických výpisů, pokud je uživatel požaduje (použije přepínač ‘-v’ (= verbose) či ‘-i’ (= instructions) na příkazové řádce).

Pokud nastane závažný chybový stav, který nedovoluje assembleru pokračovat v činnosti, oznamuje to uživateli srozumitelným chybovým hlášením v anglickém jazyce vypsáním na konzoli (do standardního proudu `stderr`) a následně vynuceným ukončením programu s předáním návratové hodnoty podle tabulky 4. Jiným než výše popsáním způsobem assembler během své činnosti s uživatelem neinteraguje.

Tabulka 4: Návrátové kódy programu v případě chyby.

Popis chybového stavu	Návrátová hodnota funkce int main()
ERR_NO_ERROR: Bez chyby/korektní ukončení činnosti assembleru.	0
ERR_INVALID_INPUT_FILE: Neplatný název vstupního souboru, nebo soubor neexistuje či nelze otevřít pro čtení.	1
ERR_INVALID_OUTPUT_FILE: Neplatný název výstupního souboru, nebo soubor nelze vytvořit či otevřít pro zápis.	2
ERR_SYNTAX_ERROR: Syntaktická chyba ve zdrojovém textu programu v JSA. Assembler narazil např. na instrukci, která není uvedena v tab. 3 nebo u instrukce chybí parametr nebo je naopak uveden parametr u instrukce, která žádný nevyžaduje.	3
ERR_FILE_ACCESS_FAILURE: Soubor (vstupní či výstupní) je nepřístupný, nelze s ním pracovat (protože je např. otevřen ve výhradním režimu jiným procesem).	4
ERR_OUT_OF_MEMORY: Assembleru došla operační paměť. To by se stát nemělo, ale pokud ano, bude program reagovat tímto návratovým kódem.	5
ERR_UNRESOLVED_REFERENCE: Assembler narazil na situaci, kdy je v kódu použit symbol (např. identifikátor návěští, funkce, objektu v datovém segmentu, apod.), který nelze nalézt a převést na adresu.	6
ERR_CODE_SEGMENT_TOO_LARGE: Kódový segment je příliš velký a nevejde se do limitu 256 KB daného specifikací počítače KM.	7
ERR_DATA_SEGMENT_TOO_LARGE: Datový segment je příliš velký a nevejde se do limitu 256 KB daného specifikací počítače KM.	8
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 konstrukce assembleru je věc v informatice celkem běžná, lze k tomu nalézt velké množství různých dokumentace např. za pomoci vyhledávače Google:

1. Informace o JSA a assembleru na Wikipedii – <https://en.wikipedia.org/wiki/Assembly>
2. Instrukční sada CPU Intel 8086 – https://en.wikipedia.org/wiki/X86_instruction_listings
3. Syntaktická analýza rekurzivním sestupem – https://en.wikipedia.org/wiki/Recursive_descent_parser
4. Brian Callahan: Starting an assembler – <https://briancallahan.net/blog/20210408.html>
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. Ačkoli vypadá zadání na první pohled velmi složité, lze řešení implementovat programem o přibližně 2000 řádcích zdrojového kódu (přednášející to zvládl na 1847 řádek).

Příloha: Ukázka JSA a strojového kódu počítače KM

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

```
1 .KMA
2 .DATA
3     swapped DWORD ?           ; items swapped in the inner loop, DS:[0]
4     array  DWORD 20 DUP(?)    ; the data to be sorted, DS:[4]
5 .CODE
6     MOV     A, 1               ; 10 01 01 00 00 00
7     STOR    A, OFFSET swapped ; 15 01 00 00 00 00
8 @LBubbleWhileBeg:             ; <--- CS:[12]
9     LOAD    A, OFFSET swapped ; 13 01 00 00 00 00
10    CMP     A, 1               ; 60 01 01 00 00 00
11    JNE     @LBubbleWhileEnd   ; 73 62 00 00 00
12 @LBubbleForInit:             ; <--- CS:[29]
13    MOV     C, 19              ; 10 03 13 00 00 00
14    MOV     A, 0               ; 10 01 00 00 00 00
15    STOR    A, OFFSET swapped ; 15 01 00 00 00 00
16    MOV     S, OFFSET array    ; 10 05 04 00 00 00
17    MOV     D, OFFSET array    ; 10 04 04 00 00 00
18    ADD     D, 4               ; 30 04 04 00 00 00
19 @LBubbleForBeg:              ; <--- CS:[65]
20    LOAD    A, S               ; 14 01 05
21    LOAD    B, D               ; 14 02 04
22    CMP     A, B               ; 61 01 02
23    JNG     @L01               ; 76 12 00 00 00
24    STOR    B, S               ; 16 02 05
25    STOR    A, D               ; 16 01 04
26    MOV     A, 1               ; 10 01 01 00 00 00
27    STOR    A, OFFSET swapped ; 15 01 00 00 00 00
28 @L01:                        ; <--- CS:[97]
29    DEC     C                  ; 39 03
30    ADD     S, 4               ; 30 05 04 00 00 00
31    ADD     D, 4               ; 30 04 04 00 00 00
32    CMP     C, 0               ; 60 03 00 00 00 00
33    JG      @LBubbleForBeg     ; 74 C7 FF FF FF
34 @LBubbleForEnd:              ; <--- CS:[122]
35    JMP     @LBubbleWhileBeg   ; 70 0C 00 00 00
36 @LBubbleWhileEnd:           ; <--- CS:[127]
37
38 @LOutputForInit:            ; <--- CS:[127]
39    MOV     C, 20              ; 10 03 14 00 00 00
40    MOV     S, OFFSET array    ; 10 05 04 00 00 00
41 @LOutputForBeg:             ; <--- CS:[139]
42    LOAD    A, S               ; 14 01 05
43    OUTD    A                  ; F0 01
44    ADD     S, 4               ; 30 05 04 00 00 00
45    DEC     C                  ; 39 03
46    CMP     C, 0               ; 60 03 00 00 00 00
47    JG      @LOutputForBeg     ; 74 E8 FF FF FF
48 @LOutputForEnd:            ; <--- CS:[157]
49    HALT                      ; 00
```