VŠB - Technická univerzita Ostrava
Katedra informatiky, FEI

Assembler x86

Studijní text pro předmět:

APPS - Architektury počítačů a paralelních systémů

Ing. Petr Olivka, Ph.D.
2020

e-mail: petr.olivka@vsb.cz
http://poli.cs.vsb.cz

Obsah

1 Úvod **2**

2 Procesory AMD a Intel - 64bitový režim **3**
2.1 Registry 3
2.2 Adresování 6
2.3 Datové typy 6

3 Spojování programů v JSI a v jazyce C **7**
3.1 Spojování C - C 7
3.2 Spojování JSI a C, syntaxe JSI 9
3.3 Používání proměnných v JSI 10
3.3.1 Přesun dat pomocí instrukce MOV 10
3.3.2 Přesun dat s rozšířením instrukcemi MOVZX a MOVSX 10
3.3.3 Příklady použítí proměnných v JSI 11

4 Instrukční soubor **17**
4.1 Přesunové instrukce 17
4.2 Logické a bitové instrukce 18
4.3 Aritmetické instrukce 20
4.4 Skokové instrukce 21
4.5 Pomocné a řídící instrukce 22
4.6 Instrukce násobení a dělení 24
4.7 Příklady použití instrukcí 25
4.7.1 Příklady použití instrukcí bitových a aritmetických . . 25
4.7.2 Příklady použití podmíněných skoků 27

5 64bitové rozhraní C - JSI **31**
5.1 Návratové hodnoty funkcí 31
5.2 Používání registrů 31
5.3 Volání funkcí s parametry 32
5.4 Typové příklady předávání parametrů do funkcí 33

6 Literatura **41**

1 Úvod

Nedílnou součástí předmětu Architektury počítačů a paralelních systémů je i využití strojových instrukcí při programování. Cílem je lépe pochopit základní principy činnosti počítače, adresování, správné používání znaménkových a bezznamenkových datových typů a návaznost na programování v jazyce C.

Při programování s využítím strojových instrukcí se v dnešní době nepíše program přímo v binárním strojovém kódu. Programy se píší s využítím jazyka symbolických instrukcí (JSI), pro který je často i v českém jazyce užíván anglický název Assembler. Každá strojová instrukce má v JSI svou zkratku a strojový kód je z JSI vytvořen až překladačem.

V předmětu APPS bude pro seznámení se strojovým kódem využit procesor AMD a Intel rodiny x86 v 64bitovém režimu a s omezením jen na celočíselnou ALU. Procesory Intel a AMD patří mezi nejčastěji zastoupené procesory v běžně používané počítačové technice.

Procesory jsou dobře dokumentovány výrobci, ale dokumentace je velmi rozsáhlá. Proto se tento text zaměřuje jen na výběr nejnutnějších informací.

V následujících kapitolách budou představeny základní informace o procesoru, informace o spojování více zdrojových kódů v C i JSI a minimalistický výběr instrukcí. Tyto informace jsou dostačují pro psaní vlastních jednoduchych programů.

2 Procesory AMD a Intel - 64bitový režim

První 64bitový procesor x86 byl navržen a realizován firmou AMD. Firma provedla rozšíření sady osmi 32bitových registrů procesorů předchozích generací na sadu registrů 64bitových. Při přechodu na 64bitový procesor však nebyly pouze rozšířeny registry, ale došlo také k navýšení počtu pracovních registrů na 16, tedy na dvojnásobek.

Instrukční sada procesoru zůstala zachována.

Firma AMD zvolila prakticky stejný způsob rozšíření, jaký při přechodu z 16bitového procesoru na 32bitový zvolila již dříve firma Intel. Tehdy bylo provedeno pouze rozšíření registrů.

Intel následně rozšířil své 32bitové procesory na 64bitové stejným způsobem, jako to realizovala firma AMD.

2.1 Registry

Celkový přehled 64bitových registrů je na obrázku 1.

64-bit	MSB			LSB	16-bit	32-bit
RAX			AH	AL	AX	EAX
RBX			BH	BL	BX	EBX
RCX			CH	CL	CX	ECX
RDX			DH	DL	DX	EDX
RDI				DIL	DI	EDI
RSI				SIL	SI	ESI
RSP				SPL	SP	ESP
RBP				BPL	BP	EBP
R8				R8L	R8W	R8D
R9				R9L	R9W	R9D
R10				R10L	R10W	R10D
R11				R11L	R11W	R11D
R12				R12L	R12W	R12D
R13				R13L	R13W	R13D
R14				R14L	R14W	R14D
R15				R15L	R15W	R15D

Obrázek 1: Přehled registrů 64bitového procesoru x86

Registry 64bitové pro všeobecné použití:

`RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8 až R15`.

Registry 32bitové:

výše uvedené registry dovolují přístup ke své dolní 32bitové části přes: `EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, R8D až R15D`.

Pozor! Při zápisu do 32bitové části registru se horní část registru Rxx automaticky vynuluje!

Registry 16bitové:

výše uvedené registry dovolují přístup ke své dolní 16bitové části přes: `AX, BX, CX, DX, SI, DI, BP, SP, R8W až R15W`.

Registry 8bitové:

první čtyři 16bitové registry jsou rozděleny na horní a dolní 8bitové části: `AH` (high), `AL` (low), `BH, BL, CH, CL, DH, DL`.

Další 8bitové registry jsou:

`DIL, SIL, SPL, BPL, R8L až R15L`.

Registry, které nejsou v obrázku 1 zakresleny:

Čítač instrukcí RIP (IP):

ukazuje na aktuální vykonávanou instrukci. Jeho změny se provádí skokovými instrukcemi, nikdy ne přímo.

Stavový registr FLAGS:

obsahuje stavové bity, které programátor využívá prostřednictvím podmíněných instrukcí. Mezi programátorem nejpoužívanější bity patří: ZF (zero flag), CF (carry), OF (overflow), SF (signum), DF (direction). Kromě těchto nejčastěji používaných bitů je implementován i PF (parity) a AF (auxiliary) bit.

CF – Carry flag – Nastaven při aritmetické operaci, kdy vznikne přenos či výpůjčka u nejvyššího bitu výsledku. V ostatních případech je vynulován. Tento příznak indikuje přetečení aritmetiky **bezznaménkových** celých čísel. Používá se také v aritmetice s rozšířenou přesností.

OF – Overflow flag – Nastavuje se pokud je celočíselný výsledek příliš velké kladné či záporné číslo (bez znaménkového bitu), které není možno uložit do cílového operandu. Jinak je vynulován. Tento příznak indikuje přetečení aritmetiky **znaménkových** celých čísel (ve dvojkovém doplňkovém kódu).

SF – Sign flag – Nastavuje se na hodnotu nejvyššího bitu výsledku, který obsahuje znaménko celých znaménkových čísel. 0 indikuje číslo kladné a 1 číslo záporné.

ZF – Zero flag – Nastaven v případě kdy výsledek je nula, jinak zůstává nulový.

Registry jsou pouze v některých případech vázány účelově. Pro úplnost některé příklady:

- RAX, EAX, AX, AL je akumulátor. Používá se při násobení, děleni a v řetězcových instrukcích.
- RCX, ECX je používán jako čítač.
- CL je použit jako čítač při bitových rotacích a posunech.
- RDX, EDX, DX je horní část argumentu při násobení a dělení.
- RSI, RDI je využíván jako indexový registr v řetězcových instrukcích.

2.2 Adresování

Adresování rozdělujeme na adresování přímé a nepřímé. Při přímém adresování uvádíme vždy konkrétní pevnou adresu. Programátor ovšem v praxi potřebuje přímou adresu jen výjimečně, nejčastěji jsou za přímou adresu považovány adresy globálních proměnných. Pokud nelze adresu určit přímo, lze použít adresování nepřímé přes registry.

V 64bitovém režimu je adresování možné pomocí konstanty a dvou registrů:

`[ Bázový + Indexový * Měřítko + Konstanta ].`

Na místě bázového i indexového registru je možno použít kterýkoliv ze šestnácti 64bitových registrů. Měřítko je jedno ze čtyř čísel: 1, 2, 4, a 8. Usnadňuje manipulaci s polem, jehož položky jsou jiné velikosti než 1 bajt.

2.3 Datové typy

Datové typy specifikují jen velikost vyhrazené paměti, nikde se neříká nic o obsahu, zda jde o znak, celé číslo se znaménkem či bez, nebo číslo reálné. Za obsah a typovou kontrolu proměnné si odpovídá programátor.

- DB - data BYTE (8 bitů)
- DW - data WORD (16 bitů)
- DD - data DWORD (32 bitů)
- DQ - data QWORD (64 bitů)

Zkrácený dvouznakový název se používá při deklaraci proměnných. Plný název se využívá v instrukcích pro určení velikosti operandu.

3 Spojování programů v JSI a v jazyce C

V dnešní době není zvykem psát v JSI celé programy, ale pouze některé jeho části. Během výuky bude využíván pro psaní hlavních částí programů jazyk C. Tento jazyk se bude využívat pro přípravu dat a následně pro výpisy výsledků, tedy pro základní vstupní a výstupní operace.

3.1 Spojování C - C

Než bude ukázáno, jak spojovat JSI s jazykem C, bude na úvod dobré uvést některé zásady a principy platné pro vytváření programu složeného z více zdrojových souborů jazyka C. Následující příklad bude složen ze dvou zdrojových souborů: `c-main.c` a `c-module.c`. Nejprve `c-module.c`:

```
// c-module.c
int g_module_sum;
static int g_module_counter = 0;

static int inc_counter()
{ m_counter++; }

int inc_sum()
{ g_module_sum++; inc_counter(); }
```

Následuje hlavní část programu s funkcí main.

```
// c-main.c
// external function prototypes
int inc_sum();
int inc_counter();
// external variables
extern int g_modul_sum;
extern int g_modul_counter;

int main() {
    g_modul_sum = 0;
    inc_sum();
    // g_modul_counter = 0; // impossible
    // inc_counter();       // impossible
    printf( "sum %d\n", g_modul_sum );
    //printf( "counter %d\n", g_modul_counter );
}
```

Nejprve je potřeba si uvědomit, jaký je rozdíl mezi funkcemi a proměnnými, u kterých je při jejich deklaraci uvedeno klíčové slovo `static`. V souboru `module.c` je takto deklarována proměnná `g_module_counter` a funkce `inc_counter`. Takto deklarované funkce a proměnné mají platnost pouze v tom zdrojovém souboru, kde jsou deklarovány. Nejsou tedy zveřejněny a nelze je použít z jiného modulu.

Všechny ostatní identifikátory funkcí a proměnných jsou veřejné.

Pokud je potřeba v jiném modulu, v našem případě v `main.c`, použít funkce nebo proměnné z jiného zdrojového souboru, je nutno uvést jejich prototyp. Správně se mají tyto prototypy uvádět v hlavičkových souborech. Zde v jednoduchém příkladu jsou tyto prototypy na začátku zdrojového kódu.

V programu `main.c` jsou uvedeny prototypy funkcí `inc_counter()` a `inc_sum()`. Tyto prototypy říkají, že se jedná o externí funkce.

Podobně je možno uvést prototypy externích proměnných `g_module_counter` a `g_module_sum`. U prototypů proměnných už je ale nutno uvést klíčové slovo `extern` (u funkcí překladač snadno pozná prototyp funkce od její deklarace podle středníku za závorkou, u proměnných nikoliv).

Ve funkci `main` je pak možno přistupovat ke všem proměnným a volat všechny funkce, které jsou známé. Překladač takový kód přeloží. Problém vznikne až v následujícím kroku, kdy se budou jednotlivé moduly linkovat do výsledného programu. Ty identifikátory, které nejsou nikde deklarovány, nebo nejsou veřejné, budou označeny jako neznámé a výsledný program nelze sestavit. Proto jsou ve funkci `main` některé řádky zakomentovány, protože i když by bylo možno program přeložit, uvedené symboly budou při linkování neznámé.

Je proto potřeba si uvědomit 3 hlavní zásady pro spojování více zdrojových souborů do jednoho programu. V každém programovacím jazyce jsou definovaná 3 základní pravidla a klíčová slova, která určují pro všechny proměnné a funkce:

- veřejné symboly,
- lokální (neveřejné) symboly,
- externí symboly.

Pravidla pro jazyk C byla uvedena v předchozím textu.

3.2 Spojování JSI a C, syntaxe JSI

V následující ukázce `module.asm` bude uveden kód, který rovnocenným způsobem nahrazuje `module.c` z kapitoly 3.1

```
    ; module.asm
    bits 64                            ; 64 bit code
    section  .data                     ; data section
    global   g_module_counter          ; public symbol
g_module_counter    dd 0               ; variable g_module_couter
g_module_sum        dd 0               ; variable g_module_sum

    section  .text                     ; code section
    global  inc_sum                    ; public symbol

inc_counter :                          ; function  inc_counter
    inc  dword  [  g_module_counter  ];  g_module_counter ++
    ret

inc_sum :                              ; function  inc_sum
    inc  dword  [  g_module_sum  ]     ;  g_module_sum ++
    call  inc_counter                  ;  inc_counter ()
    ret
```

Uvedený kód v JSI musí být vždy rozdělen na část datovou a kód. V datové části je ukázáno, jak s použitím datových typů z předchozí kapitoly deklarovat proměnné `g_module_counter` a `g_module_sum`. V JSI platí pravidlo, že všechny symboly jsou lokální. Proto pro zveřejnění symbolu `g_module_counter` je nutno použít klíčové slovo `global`.

Podobně je tomu tak i v části s kódem. Funkce jsou vytvořeny tím, že se v kódu uvede návěští odpovídajícího jména. Opět platí, že symboly jsou pouze lokální a pro jejich zveřejnění je potřeba použít `global`.

Z uvedené krátké ukázky kódu je také patrné:

- komentáře se ve zdrojovém kódu JSI označují středníkem,
- instrukce se píší odsazené od levého okraje,
- na levém okraji se píší pouze názvy proměnných a návěští.

Další podrobnosti o psaní zdrojového kódu v JSI jsou uvedeny v dokumentaci překladače NASM, který bude využíván ve výuce. Je potřeba se seznámit zejména se způsoby zápisů číselných a znakových konstant a také se způsobem zápisu řetězců.

Další příklady pro spojování modulů v jazyce C a JSI jsou přiloženy v příkladech `c-c-example` a `c-asm-example*`. Pro sestavení programů je potřeba použít příkaz `make`.

3.3 Používání proměnných v JSI

Pro přístup k proměnným je potřeba se nejprve podívat podrobněji na instrukce pro přesun dat. Z kapitoly 4 budou vybrány tři. Nejjednodušší instrukcí je instrukce MOV. Kromě této instrukce bude potřeba i dvojice instrukcí MOVZX z MOVSX.

3.3.1 Přesun dat pomocí instrukce MOV

Instrukce MOV je možno použít několika způsoby:

```
MOV cíl, zdroj            ; cíl = zdroj
```

Jako operandy instrukce lze použít registry (R), proměnné v paměti (M) a konstanty (K). Tyto 3 typy operandů je možné použít v pěti různých kombinacích a to i u většiny dalších instrukcí:

```
MOV R, R                  ; přesun registru do registru
MOV R, M                  ; přesun paměti do registru
MOV M, R                  ; přesun registru do paměti
MOV R, K                  ; přesun konstanty do registru
MOV M, K                  ; přesun konstanty do paměti
```

Ve všech těchto pěti případech platí několik zásad, které platí i pro naprostou většinu dalších instrukcí:

- velikost obou operandů musí být stejná,
- nikdy nelze použít dva paměťové operandy,
- v kombinaci R,M a M,R a R,K určuje velikost operandu vždy použitý registr,
- pokud není ani jeden operand registr, musí velikost operandu určit programátor pomocí typu (viz datové typy v kapitole 2.3: byte, word, dword, atd).

3.3.2 Přesun dat s rozšířením instrukcemi MOVZX a MOVSX

```
MOVZX cíl, zdroj          ; cíl = zdroj
MOVSX cíl, zdroj          ; cíl = zdroj
```

Jako parametry instrukce lze použít registry (R) a proměnné v paměti (M) a to jen ve dvou kombinacích:

```
MOV R, R                    ; přesun registru do registru
MOV R, M                    ; přesun paměti do registru
```

Pro oba parametry musí platit, že velikost operandu `cíl` musí být větší, než velikost operandu `zdroj`.

Insrukce `MOVZX` provede přesun operandu s rozšířením o nuly do vyšších bitů. Instrukce `MOVSX` rozšíří operand znaménkově.

Příklady použití budou vedeny dále.

3.3.3 Příklady použítí proměnných v JSI

Následující příklady používání proměnných v JSI budou demonstrovat přístup ke globálním proměnným a to deklarovaným v jazyce C. Některé příklady byly uvedeny již v příkladu `c-asm-example-*`. Další ukázky jsou uvedeny v kódu `variables-*`. Zde budou z příkladu vybrány a popsány jen některé funkce.

Nejprve hlavní program v jazyce C.

```
// c-main.c

#define N       10
#define LEN     128

// global variables
char g_character = '\0';
int32_t g_integer = 0;          // equivalent to int
int64_t g_longint = 0;          // equivalent to long

int32_t g_index = 0;
int32_t g_value = 0;

// global arrays
char g_string[ LEN ] = "Hello World!";
int32_t g_int_array[ N ] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
int64_t g_long_array[ N ] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

void print_int_array ()
{ ... }
void print_long_array ()
{ ...  }

int main()
{
    // assign a constant into global variables
    set_variables();
    printf( "g_character:'%c', g_integer:%d, g_longint:%lx\n\n",
            g_character, g_integer, g_longint );
```

```
    // select code you want to test

    // move value from g_value into g_index
    g_value = 333;
    g_index = 0;
    move_int_values();
    printf( "g_value:␣%d,␣g_index:␣%d\n\n", g_value, g_index );

    // move value from g_integer into g_longing
    g_integer = 1200300400 ;
    g_longint = 0;
    move_int_to_long();
    printf( "g_integer:␣%d,␣g_longint:␣%ld\n\n", g_integer, g_longint );

    // rewrite content of global string
    set_string();
    printf( "g_string:'%s'\n\n", g_string );

    // rewrite one element of g_int_array at g_index position
    g_index = 5;
    set_int_array_index();
    g_index = 6;
    set_int_array_index();
    print_int_array();

    // rewrite one element of g_long_array at g_index position with g_value
    g_value = -2020;
    g_index = 0;
    set_long_array_index_value();
    g_value = -2021;
    g_index = 1;
    set_long_array_index_value();
    print_long_array();
}
```

Nyní následují ukázky používání proměnných v JSI. Přístup k proměnným deklarovaným v C i v JSI je zcela shodný. V příkladech ale budou preferovány zejména proměnné deklarované v jazyce C.

```
    ; asm-module.c
    bits 64

    section .data

    ; extern variables
    extern   g_character
    extern   g_integer
    extern   g_longint

    extern  g_index
    extern  g_value

    ; extern arrays
    extern  g_string
    extern  g_int_array
    extern  g_long_array

    section .text

    ; assign a constant into global variables
    global  set_variables

set_variables:
    enter  0,0

    mov byte [ g_character ], 'X'        ; g_character = 'X';
    mov dword [ g_integer ], 12345       ; g_integer = 12345;
    mov qword [ g_longint ], 0x12345678  ; g_longint = 0x12345678;

    leave
    ret
```

V uvedeném kódu jsou nejprve všechny proměnné deklarované dříve v jazyce C označeny jako externí.

Ve funkci `set_variables()` se do třech proměnných: `g_character`, `g_integer` a `g_longint` nastavuje libovolná konstanta. Odpovídající zápis v jazyce C je uveden v komentáři.

Pro adresování se vždy používá zápis se závorkami `[]`, jak bylo uvedeno již v kapitole 2.2.

Proto je při přístupu k proměnným vždy použit identifikátor proměnné (vnitřně překladačem označující její adresu) uzavřený do hranaté závorky.

V uvedeném příkladu není ani u jedné instrukce `MOV` použit registr jako

zdrojový či cílový operand. Proto je potřeba uvést odpovídající velikost operandu, viz kapitola 2.3.

V dalším příkladu je ukázka přesunu obsahu proměnné `g_value` do proměnné `g_index`.

```
    ; move value from g_value into g_index

    global move_int_values

move_int_values :
    enter 0,0

    mov ecx, [ g_value ]      ; ecx = g_value
    mov [ g_index ], ecx      ; g_index = ecx

    leave
    ret
```

Při přesunu obsahu z jedné proměnné do druhé je potřeba mít na paměti popis instrukce `MOV` z kapitoly 3.3.1, kde bylo uvedeno, že instrukce nemůže použít dva paměťové operandy. Proto je potřeba přesun hodnoty mezi proměnnými provést přes libovolný registr. Za výběr registru správné velikosti odpovídá programátor.

V předchozím příkladu byl přesun hodnoty proveden mezi proměnnými stejné velikosti.

V následujícím příkladu bude ukázáno, jak přenést obsah mezi proměnnými různých velikostí.

```
    ; move value from g_integer into g_longint
    global move_int_to_long

move_int_to_long :
    enter 0,0

    movsx rax, dword [ g_integer ]  ; rax = g_integer
    mov [ g_longint ], rax          ; g_longing = rax

    leave
    ret
```

Obsah proměnné `g_integer` je přenesen to proměnné `g_longint`. Protože ale obě proměnné mají různou velikost, bylo potřeba nejprve provést přesun proměnné `g_integer` do registru `RAX` pomocí instrukce `MOVSX`. Musí být použita právě tato instrukce, protože se přenáší obsah mezi znaménkovými proměnnými.

Další ukázka funkce `set_string()` provede přepis části obsahu řetězce `g_string`.

```
    ; rewrite content of global string
    global set_string

set_string :
    enter 0,0

    mov byte [ g_string + 5 ], 'w'      ; g_string [ 5 ] = 'w'
    mov byte [ g_string + 6 ], 'e'      ; g_string [ 6 ] = 'e'
    mov byte [ g_string + 7 ], 'e'      ; g_string [ 7 ] = 'e'
    mov byte [ g_string + 8 ], 'n'      ; g_string [ 8 ] = 'n'
    mov byte [ g_string + 9 ], ':'      ; g_string [ 9 ] = ':'
    mov byte [ g_string + 10 ], '-'     ; g_string [ 10 ] = '-'
    mov byte [ g_string + 11 ], ')'     ; g_string [ 11 ] = ')'

    leave
    ret
```

V kódu je vidět adresování prvků pole. Proměnná `g_string` je pole znaků a pro přístup k jednotlivým znakům je použit identifikátor proměnné a posun v poli. Z kódu a komentáře ke každému řádku je patrné, že rozdíl v zápise v JSI a C je velmi podobný.

Další ukázka je zaměřena na indexování pole pomocí proměnné.

```
    ; rewrite one element of g_int_array at g_index position
    global set_int_array_index

set_int_array_index :
    enter 0,0

    movsx rax, dword [ g_index ]     ; 64-bit address
    mov dword [ g_int_array + rax * 4 ], 646464
                                     ; g_int_array [ g_index ] = 646464

    leave
    ret
```

Při adresování nelze přímo použít proměnnou jako index pole. V závorce `[]` smí být pouze konstanty a maximálně dva registry. Proto je kódu nejprve potřeba převést hodnotu proměnné `g_index` do registru. Převedení musí být s rozšířením na 64 bitů, protože v 64bitovém režimu se musí adresovat 64bitovou hodnotou.

Poté je registr použit jako index. Hodnota registru je navíc násobena tzv. měřítkem, viz 2.2, což v praxi není nic jiného, než velikost jednoho prvku pole v bytech.

Posledním vybraným příkladem je přenos obsahu proměnné `g_value` do pole `g_long_array` na pozici `g_index`.

```
    ; rewrite one element of g_long_array at g_index position with g_value
    global  set_long_array_index_value

set_long_array_index_value :
    enter 0,0

    movsx rcx, dword [ g_value ]      ; rcx = g_value with sign extension
    movsx rax, dword [ g_index ]      ; 64-bit address
    mov [ g_long_array + rax * 8 ], rcx
                                      ;  g_long_array [ g_index ] = rcx

    leave
    ret
```

V kódu je vidět složení předchozích příkladů. Přenos proměnné `g_value` se musí do prvku pole `g_long_array[ g_index ]` provést pomocí registru a se znaménkovým rozšířením. Indexování je také nutno provést s příslušným rozšířením a index správně vynásobit velikostí prvku pole, což je v tomto případě 8.

Za správné provedení převodů si zodpovídá programátor.

4 Instrukční soubor

Z celého instrukčního souboru procesoru i486 a jeho následníků se využívá v běžné praxi ani ne polovina všech instrukcí celočíselné jednotky ALU. Ty jsou v literatuře řazeny abecedně. Výhodnější je v začátku ovšem rozdělení instrukcí tématicky do několika skupin:

- přesunové,
- logické a bitové,
- aritmetické,
- skokové,
- pomocné a řídící.

Popis instrukcí je možno nalézt přímo v dokumentaci výrobce. V příloze tohoto textu jsou dokumenty `intel-AZ.pdf`. Tato dokumentace má více než dva tisíce stránek a pro běžnou programátorskou práci je nevhodná. Natož pro začínající programátory.

Za dokumentaci svým rozsahem vhodnou pro běžné použití lze považovat například v dokumentu `nasm-0.98.pdf` přílohu B, která v dostatečné míře popisuje všechny potřebné instrukce.

Následující popis řadí pro lepší přehlednost instrukce tématicky a popis jednotlivých instrukcí je velmi stručný. Funkcionalitu jednotlivých instrukcí si může každý vyzkoušet samostatně.

4.1 Přesunové instrukce

`MOV cíl, zdroj`

Instrukce provede přesun obsahu operandu `zdroj` do operandu `cíl`. Velikost obou operandů musí být stejná. Přesouvat lze obsah paměti, registru a konstantu.

`CMOVcc cíl, zdroj`

Instrukce provede přesun obsahu operandu `zdroj` do operandu `cíl`, pokud bude splněna podmínka `cc`. Význam této zkratky je vysvětlen dále u podmíněných skoků. I zde musí být velikost obou operandů stejná. Přesouvat lze obsah paměti nebo registru do registru.

`MOVZX cíl, zdroj`

Instrukce se používá v případě, že velikost operandu `zdroj` je menší, než velikost operandu `cíl` a provádí se rozšíření nulami.

`MOVSX cíl, zdroj`

Stejně jako MOVZX, ale provede se znaménkové rozšíření.

`XCHG cíl, zdroj`

Vymění se obsah obou operandů.

`BSWAP cíl`

Provede změnu pořadí bytů. Jedná se konverzi little a big endian formátu čísla.

`PUSH zdroj`

Uloží na vrchol zásobníku obsah operandu `zdroj` a posune vrchol zásobníku.

`POP cíl`

Z vrcholu zásobníku se přesune hodnota do operandu `cíl` a sníží vrchol zásobníku.

4.2 Logické a bitové instrukce

Instrukce ovlivňují obsah příznakového registru procesoru. Logické instrukce mění SF a ZF, bitové posuny i CF.

`AND cíl, zdroj`

Instrukce provede bitově `cíl = cíl and zdroj`.

`TEST cíl, zdroj`

Instrukce provede bitově `cíl and zdroj`. Jde o operaci AND bez uložení výsledku.

`OR cíl, zdroj`

Instrukce provede bitově `cíl = cíl or zdroj`.

`XOR cíl, zdroj`

Instrukce provede bitově `cíl = cíl xor zdroj`.

`NOT cíl`

Instrukce provede negaci všech bitů operandu.

`SHL/SAL cíl, kolik`

Bitový i aritmetický posun doleva operandu `cíl` o požadovaný počet bitů. Operand `kolik` je konstanta nebo registr CL.

`SHR cíl, kolik`

Bitový posun doprava operandu `cíl` o požadovaný počet bitů. Operand `kolik` je konstanta nebo registr CL.

`SAR cíl, kolik`

Aritmetický posun doprava operandu `cíl` o požadovaný počet bitů. Operand `kolik` je konstanta nebo registr CL.

`ROL cíl, kolik`

Bitová rotace doleva operandu `cíl` o požadovaný počet bitů. Operand `kolik` je konstanta nebo registr CL.

`ROR cíl, kolik`

Bitová rotace doprava operandu `cíl` o požadovaný počet bitů. Operand `kolik` je konstanta nebo registr CL.

`RCL cíl, kolik`

Bitová rotace doleva přes CF operandu `cíl` o požadovaný počet bitů. Operand `kolik` je konstanta nebo registr CL.

`RCR cíl, kolik`

Bitová rotace doprava přes CF operandu `cíl` o požadovaný počet bitů. Operand `kolik` je konstanta nebo registr CL.

`BT cíl, číslo`

Zkopíruje do CF hodnotu bitu daného operandem `číslo` z operandu `cíl`.

`BTR cíl, číslo`

Zkopíruje do CF hodnotu bitu daného operandem `číslo` z operandu `cíl` a nastaví jej na nulu.

`BTS cíl, číslo`

Zkopíruje do CF hodnotu bitu daného operandem `číslo` z operandu `cíl` a nastaví jej na jedničku.

`BTC cíl, číslo`

Zkopíruje do CF hodnotu bitu daného operandem `číslo` z operandu `cíl` a provede jeho negaci.

`SETcc cíl`

Nastaví `cíl` na hodnotu 0/1 podle toho, zda je splněna požadovaná podmínka (podmínky viz podmíněné skoky).

`SHRD/SHLD cíl, zdroj, kolik`

Provede nasunutí `kolik` bitů ze zdroje do cíle. Zdroj se nemění.

4.3 Aritmetické instrukce

Všechny aritmetické instrukce nastavují nejen cílový operand, ale nastavují i všechny příznakové bity v registru FLAGS.

`ADD cíl, zdroj`

Instrukce provede aritmetické sčítání `cíl = cíl + zdroj`

`ADC cíl, zdroj`

Instrukce provede aritmetické sčítání včetně CF `cíl = cíl + zdroj + CF`

`SUB cíl, zdroj`

Instrukce provede aritmetické odčítání `cíl = cíl - zdroj`

`CMP cíl, zdroj`

Instrukce provede aritmetické odčítání `cíl - zdroj`, ale neuloží výsledek.

`SBB cíl, zdroj`

Instrukce provede aritmetické odčítání s výpůjčkou `cíl = cíl - zdroj - CF`

`INC cíl`

Instrukce provede zvýšení operandu o jedničku. Zvláštností je, že nemění CF.

`DEC cíl`

Instrukce provede snížení operandu o jedničku. Nemění CF.

`NEG cíl`

Instrukce provede změnu znaménka operandu.

`MUL zdroj`

Instrukce pro násobení dvou bezznaménkových čísel. Operandem `zdroj` se podle jeho velikosti (8, 16, 32) bitů násobí akumulátor (AL, AX, EAX) a výsledek se uloží do (AX, AX-DX, EAX-EDX). Dále viz 4.6

`IMUL zdroj`

Instrukce pro násobení dvou znaménkových čísel. Operandem `zdroj` se podle jeho velikosti (8, 16, 32) bitů násobí akumulátor (AL, AX, EAX) a výsledek se uloží do (AX, AX-DX, EAX-EDX). Dále viz 4.6

`DIV zdroj`

Instrukce pro dělení dvou bezznaménkových čísel. Operandem `zdroj` se podle jeho velikosti (8, 16, 32) bitů dělí připravená hodnota v (AX, AX-DX, EAX-EDX) a výsledek se uloží do (AL, AX, EAX) a zbytek po dělení bude v (AH, DX, EDX). Dále viz 4.6

`IDIV zdroj`

Instrukce pro dělení dvou znaménkových čísel. Operandem `zdroj` se podle jeho velikosti (8, 16, 32) bitů dělí připravená hodnota v (AX, AX-DX, EAX-EDX) a výsledek se uloží do (AL, AX, EAX) a zbytek po dělení bude v (AH, DX, EDX). Dále viz 4.6

`CBW`

Instrukce pro znaménkové rozšíření registru AL do AX. Používá se před znaménkovým dělením.

`CWD`

Instrukce pro znaménkové rozšíření registru AX do AX-DX. Používá se před znaménkovým dělením.

`CDQ`

Instrukce pro znaménkové rozšíření registru EAX do EAX-EDX. Používá se před znaménkovým dělením.

`CQO` (64bitový režim)

Instrukce pro znaménkové rozšíření registru RAX do RAX-RDX. Používá se před znaménkovým dělením.

4.4 Skokové instrukce

`JMP cíl`

Provádění programu se přenese na adresu danou operandem `cíl`. Většinou jde o jméno návěští, kam se řízení přesouvá, ale může jít i o `cíl` daný adresou v registru nebo v paměti.

`CALL cíl`

Volání podprogramu. Stejně jako JMP, ale na vrchol zásobníku se uloží adresa instrukce následující za CALL.

`RET N`

Z vrcholu zásobníku se odebere adresa na kterou se následně předá řízení. Jde tedy o návrat z podprogramu. Volitelný operand `N` odstraní z vrcholu zásobníku dalších N bytů.

`LOOP cíl`

Vyjádřeno jazykem C se provede: `if ( --ECX ) goto cíl;`. Jde o řízení cyklu, kde počet opakování je dán registrem ECX. Registr ECX se dekrementuje před vyhodnocením podmínky!

`LOOPE/Z cíl`

Vyjádřeno jazykem C se provede: `if ( --ECX && ZF ) goto cíl;`.

`LOOPNE/NZ cíl`

Vyjádřeno jazykem C se provede: `if ( --ECX && !ZF ) goto cíl;`.

`JCXZ cíl`

Provede skok na požadované místo jen pokud je registr ECX (CX) nulový.

`Jcc cíl`

Skupina podmíněných skoků.

První podskupina řeší elementární podmíněné skoky:

Instrukce `JZ/E, JNZ/NE, JS, JNS, JC, JNC, JO, JNO` testují přímo jednotlivé bity ve stavovém registru procesoru.

Druhá podskupina řeší porovnávání čísel:

`JB/JNAE/JC` - menší než, není větší nebo rovno,

`JNB/JAE/JNC` - není menší, větší nebo rovno,

`JBE/JNA` - menší nebo rovno, není větší,

`JNBE/JA` - není menší nebo rovno, je větší,

`JL/JNGE` - menší než, není větší nebo rovno,

`JNL/JGE` - není menší, větší nebo rovno,

`JLE/JNG` - menší nebo rovno, není větší,

`JNLE/JG` - není menší nebo rovno, je větší.

Ve výše uvedených instrukcích mají písmena **A-B-L-G-N-E** svůj pevně daný význam.

Pro operace s bezznaménkovými operandy se používá **A** (above) a **B** (below). U operandů znaménkových se používá **L** (less) a **G** (greater).

Pro negaci je **N** (not) a pro rovnost **E** (equal).

4.5 Pomocné a řídící instrukce

`CLD`

DF nastaví na nulu.

`STD`

DF nastaví na jedničku.

`CLC`

CF nastaví na nulu.

`STC`

CF nastaví na jedničku.

`CMC`

Provede negaci (complement) CF.

`NOP`

Prázdná instrukce.

4.6 Instrukce násobení a dělení

V kapitole 4.3 byly stručně popsány instrukce pro násobení a dělení. Pro lepší názornost je činnost těchto aritmetických instrukcí přehledně vidět na obrázku 2.

	MUL/IMUL r/m		
	1st Factor	2nd Factor	Product
AH	AL	r/m 8 bits	AX
DX	AX	r/m 16 bits	AX-DX
EDX	EAX	r/m 32 bits	EAX-EDX
RDX	RAX	r/m 64 bits	RAX-RDX
Remainder	Quotient	Divisor	Divident
DIV/IDIV r/m			

Obrázek 2: Chování instrukcí násobení MUL/IMUL a dělení DIV/IDIV

V případě bezznaménkového násobení `MUL` a znaménkového `IMUL` je potřeba se na obrázek dívat shora a zleva doprava. Obě instrukce mají jediný operand a tím je registr nebo paměť. Tento operand je současně druhým činitelem (2nd Factor) součinu. První činitel (1st Factor) součinu je určen právě velikostí operandu. Výsledek (Product) je vždy dvojnásobné velikosti než operandy a je pak uložen do odpovídajících registrů.

Akumulátor `AL/AX/EAX/RAX` obsahuje vždy LSB část a datový registr `DX/EDX/RDX` obsahuje MSB část výsledku.

Pro bezznaménkové dělení `DIV` a znaménkové `IDIV` je nutné se na obrázek dívat zdola a zprava doleva. I u těchto instrukcí je opět rozhodující velikost jediného operandu, kterým je tentokrát dělitel (Divisor). Podle jeho velikosti je nutné, aby byl dělenec připraven do odpovídající registrů a má dvojnásobnou velikost, než operand. Výsledek (Quotient) a zbytek po dělení (Remainder) je pak uložen do odpovídajících registrů.

Před dělením musí být dělenec rozšířen na potřebnou velikost vždy odpovídajícím způsobem. Pro bezznaménkové dělení `DIV` musí proběhnout rozšíření o levostranné nuly. Před znaménkovým dělením `IDIV` se rozšíření provádí pomocí instrukcí `CBW`, `CWD`, `CDQ` a `CQO`, popsaných již v kapitole 4.3.

4.7 Příklady použití instrukcí

V této podkapitole budou uvedeny příklady použití instrukcí. Ve všech příkladech budou využívány globální proměnné.

Příklady budou dále rozděleny na dvě části. V první části budou uvedeny příklady na použíti bitových a aritmetických instrukcí. Ve druhé části pak příklady použíti podmíněných skokových instrukcí.

Všechny příklady jsou v přiloženém příkladu `instructions-64`. V textu nebude uveden zdrojový kód souboru `c-main.c`, který obsahuje funkci `main`. Kód neobsahuje žádné důležité informace potřebné pro dále popisované funkce. Ve zdrojovém kódu jsou jen deklarovány globální proměnné, volány dále popsané funkce a vypsány výsledky.

4.7.1 Příklady použití instrukcí bitových a aritmetických

V následujících příkladech jsou záměrně vynechány instrukce pro násobení a dělení.

Prvním příkladem je funkce `move_low_nibbles`. (Nibbles jsou označovány tzv. slabiky, tedy půlky bajtů. A tyto půlky mohou být horní a dolní.) Funkce přenese dolní půlky bajtů z proměnné `g_int_val1` do proměnné `g_int_val2`.

```
    ; move low nibbles from g_int_val1 to g_int_val2
    global  move_low_nibbles

move_low_nibbles :
    enter 0,0

    mov edx, 0 x0F0F0F0F      ; mask = low nibbles
    mov eax, [ g_int_val1 ] ; eax = g_int_val1
    and eax, edx            ; only low nibles in eax
    not edx                 ; ~mask (high nibbles)
    and [ g_int_val2 ], edx ; g_int_val &= mask (only high nibbles)
    or [ g_int_val2 ], eax  ; g_int_va2 |= eax

    leave
    ret
```

Pomocí masky v registru `EDX` se v původní hodnotě proměnné `g_int_val1` provede vynulování nepotřebných horních půlek bajtů pomocí `AND`. Pomocí masky invertované použitím `NOT` se provede v proměnné `g_int_val2` vynulování dolních půlek bajtů. A v posledním kroku se oba mezivýsledky spojí logickým součtem `OR`.

Další příklad ukazuje použití aritmetické operace pro sčítání `ADD` a bitového posunu vpravo `SHR`.

```
    ; compute mean value of g_long_array
    global  mean_long_array

mean_long_array :
    enter 0,0

    mov rax, [ g_long_array + 0 * 8 ] ; l_sum = g_long_array [ 0 ]
    add rax, [ g_long_array + 1 * 8 ] ; l_sum += g_long_array [ 1 ]
    add rax, [ g_long_array + 2 * 8 ] ; l_sum += g_long_array [ 2 ]
    add rax, [ g_long_array + 3 * 8 ] ; l_sum += g_long_array [ 3 ]

    shr rax, 2                        ; l_sum /= 4
    mov [ g_long_mean ], rax          ; g_long_mean = l_sum

    leave
    ret
```

Funkce vypočítá aritmetický průměr z pole `g_long_array`. Pole má jen 4 prvky a součet všech prvků je realizován opakovaným sčítáním. Následné dělení 4 je nahrazeno bitovým posunem vpravo o 2 bity.

V další ukázce je násobení proměnné `g_int_number` číslem 10. Bez použití násobení lze tento úkol realizovat snadno rozepsání do více kroků: $x * 10 = x * (2 + 8) = 2 * x + 8 * x$. Násobení mocninami čísla je 2 realizováno bitovým posunem vlevo `SHL`.

```
    ; multiply g_int_number by 10 using shl and add
    global  mult_int_number_10

mult_int_number_10 :
    enter 0,0

    mov eax, [ g_int_number ]   ; eax = g_int_number
    shl eax, 1                  ; eax *= 2
    mov ecx, eax                ; ecx = eax
    shl ecx, 2                  ; ecx *= 4 ( g_int_number * 8)
    add eax, ecx                ; eax += ecx
    mov [ g_int_number ], eax   ; g_int_number *= 10;

    leave
    ret
```

4.7.2 Příklady použití podmíněných skoků

V této podkapitole bude uvedeno několik typových příkladů na využívání podmíněných skoků. V prvé řadě bude ukázáno, jak realizovat cyklus. Poté příklad na podmínku v cyklu. Dále cyklus pro řetězec neznámé délky a jako poslední bude realizace složené podmínky.

Prvním příkladem je realizace cyklu `for(...)` pro výpočet průměrné hodnoty z prvků pole `g_int_array`. Pro výpočet průměru je použit postup známý již z předchozích příkladů.

```
    ; mean value of g_int_array
    global mean_int_array

mean_int_array :
    enter 0,0

    mov eax, 0                      ; l_sum = 0
    ; incorrect for ( rcx = 0; rcx < 8; rcx++ )
    mov rcx, 0
.back:
    add eax, [ g_int_array + rcx * 4 ]; l_sum += g_int_array [ rcx ]
    inc rcx                         ; rcx++
    cmp rcx, 8                      ; rcx < 8 ?
    jl .back                        ; yes? jump .back
    ; end for
    shr eax, 3                      ; sum /= 8
    mov [ g_int_mean ], eax         ; g_int_mean = sum

    leave
    ret
```

Cyklus je v příkladu realizován pomocí registru `RCX`. 64bitový registr je použit záměrně, aby pomocí tohoto registru bylo možno i adresovat prvky pole. Cyklus má předem známý počet opakování 8. Podmínka kontrolující chování cyklu je realizována pomocí instrukce `CMP` (což je `SUB` bez uložení výsledku) a podmíněným skokem `JL`, který kontroluje, zda `RCX` je menší než 8. Tato implementace cyklu `for()` ale není korektní, protože cyklus má podmínku až na konci. Implementace odpovídá cyklu `do {} while (...)`, kdy cyklus proběhne vždy minimálně jednou.

V tomto příkladu je tato realizace akceptovatelná, protože je předem znám počet opakování a ten není nulový. Vhodnější implementace je v následujícím příkladu.

Další příklad ukazuje vhodnější implementaci cyklu `for(...)` s podmínkou na začátku cyklu realizovanou pomocí `CMP` a `JNL`. Zde je dobré si povšimnout, že podmíněný skok řeší ukončení cyklu, samotné opakování je pomocí nepodmíněného skoku `JMP`.

V cyklu se pak provádí počítání lichých čísel. Test, zda číslo je liché, je možno realizovat snadno. Stačí ověřit hodnotu nejnižšího bitu. To lze provést např. instrukcí `AND`, když se hodnota z pole `g_int_array` nejprve přesune do registru, aby nedošlo k nechtěnému přepisu hodnot v poli. V komentáři v kódu je uvedena alternativa k instrukci `AND` a to instrukce `TEST`. Obě instrukce provádí totéž, ale `TEST` neukládá výsledek, pouze nastavuje příznakové bity v registru `FLAGS`. Pokud výsledek operace je nulový, číslo je sudé a nepočítá se do celkového součtu. Podmíněný skok je v tomto případě `JZ`.

Počítání lichých čísel se v cyklu provádí v registru. `ECX` a na závěr se ukládá do proměnné `g_odd_numbers`.

```
    ; count odd numbers in g_int_array
    global  odd_numbers_int_array

 odd_numbers_int_array :
    enter 0,0

    mov ecx, 0                          ; ecx - counter
    ; for ( rdx = 0; rdx < 8; rdx++ )
    mov rdx, 0
.back:
    cmp rdx, 8                          ; rdx < 8 ?
    jnl .end_for                        ; no? jump .end_for
    ;test dword [ g_int_array + rdx * 4 ], 1
    mov eax, [ g_int_array + rdx * 4 ]
    and eax, 1                          ; if ( g_int_array [ rdx ] & 1 )
    jz .no_odd
    inc ecx                             ; { ecx++ }
.no_odd:
    inc rdx                             ; rdx++
    jmp .back
.end_for:
    mov [ g_odd_numbers ], ecx          ; g_odd_numbers = ecx

    leave
    ret
```

Následující příklad ukazuje práci s řetězcem, kde není předem známa jeho délka. U řetězců v jazyce C je pouze definován ukončovací znak `'\0'`, což je binární nula a toho je potřeba využít při realizaci cyklu. V příkladu se počítá délka řetězce. Znaky řetězce se porovnávají s nulou a je-li nalezen ukončovací znak, podmíněný skok `JE` cyklus ukončí.

Délka řetězce se počítá v registru `RCX`, který současně slouží pro indexování znaků řetězce. Do výsledku `g_char_array_len` se uloží jen spodních 32 bitů tohoto registru.

```
    ; count lenght of g_char_array
    global  char_array_length

 char_array_length :
    enter 0,0

    mov rcx, 0                          ; rcx = lenght
.back:
    ; while ( g_char_array [ rcx ] != '\0' )
    cmp [ g_char_array + rcx ], byte 0
    je .found_0
    inc rcx                             ; { rcx++ }
    jmp .back
.found_0:
    mov [ g_char_array_len ], ecx       ; g_char_len = ecx

    leave
    ret
```

Poslední typový příklad je opět práce s řetězcem, kdy není známa jeho délka. A v cyklu se budou nahrazovat všechny číslice znakem z proměnné `g_char_replace`. Aby se mohly číslice nahradit, je potřeba realizovat složenou podmínku, protože číslice se v ASCII tabulce nachází v souvislé řadě od `'0'` do `'9'`. Cokoli je mimo tento rozsah nahrazeno nebude. Instrukční sada ovšem složené podmínky přímo nepodporuje. Je potřeba je rozložit na více podmíněných skoků.

```
    ; replace digits in g_char_array with g_char_replace
    global  char_array_replace

char_array_replace :
    enter 0,0

    mov rdx, 0
    mov al, [ g_char_replace ]
.back:
    cmp [ g_char_array + rdx ], byte 0
    je .found_0
    cmp [ g_char_array + rdx ], byte '0'
    jb .no_digit                    ; if ( g_char_array [ rdx ] >= '0' &&
    cmp [ g_char_array + rdx ], byte '9'
    ja .no_digit                    ;      g_char_array [ rdx ] <= '9' )
    mov [ g_char_array + rdx ], al  ; { g_char_array [ rdx ] = al }
.no_digit:
    inc rdx                         ; rdx++
    jmp .back
.found_0:

    leave
    ret
```

Protože bychom měli znaky považovat za čísla bezznaménková, jsou použity instrukce `JB` a `JA`. Zbytek kódu již odpovídá známým postupům z předchozích příkladů.

5 64bitové rozhraní C - JSI

Standard popisující 64bitové rozhraní je možno nalézt v přiloženém dokumentu `abi-64.pdf`

5.1 Návratové hodnoty funkcí

Způsoby předávání návratových hodnot jsou shodné v 64bitovém režimu, jako v režimu 32bitovém.

Došlo jen k malému rozšíření a změnám při předávání hodnot float a double. V 64bitovém režimu se již nevyužívá pro výpočty jednotka FPU, ale výhradně SSE.

- 8 bitů - registr AL,
- 16 bitů - registr AX,
- 32 bitů - registr EAX,
- 64 bitů - registr RAX,
- 128 bitů - registry RAX-RDX
- float/double - registr XMM0.

5.2 Používání registrů

Ve funkcích je možno obsah některých registrů měnit, u některých musí být jejich obsah zachován. Pravidla lze shrnout do následujících bodů.

- Registry `RDI, RSI, RDX, RCX, R8 a R9` používané pro předávání parametrů, se mohou ve funkci libovolně měnit.
- Registry `RAX, R10 a R11` je možné v kódu změnit.
- Registry `RSP a RBP` slouží pro práci se zásobníkem a musí být na konci funkce vráceny na původní hodnoty
- Registry `RBX, R12, R13, R14 a R15` musí být vždy obnoveny na původní hodnoty.
- Příznakový bit `DF` musí být na konci funkce vždy nastaven na hodnotu 0.
- Registry FPU - `ST0 až ST7` a registry SSE - `XMM0 až XMM15` je ve funkci možno použít a není potřeba obnovovat jejich obsah.

5.3 Volání funkcí s parametry

Pro předávání parametrů do funkcí se v 64bitovém režimu používá kombinace předávání parametrů přes registry i přes zásobník.

Pro předávání celočíselných parametrů a ukazatelů se využívá pro prvních šest parametrů **zleva** šestice registrů:

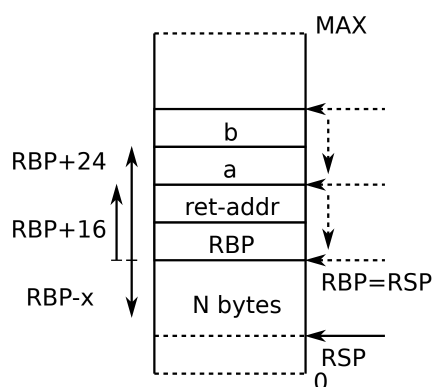
`RDI, RSI, RDX, RCX, R8 a R9.`

Prvních 8 parametrů **zleva** typu float/double se předává přes registry SSE:

`XMM0 až XMM7.`

V definici standardu je uvedena ještě jedna důležitá informace, která se týká funkcí využívajících proměnný počet parametrů (, ...). Tyto funkce vyžadují, aby byl v registru `AL` uveden počet parametrů předávaných přes registry XMMx.

Všechny další parametry se předávají přes zásobní. Všechny ukládané hodnoty jsou 64bitové.



Obrázek 3: Zásobník v 64bitovém režimu po instrukci `enter N,0`

Na obrázku 3 je ukázáno, jak by vypadal zásobník v případě, kdyby parametry `a` a `b` byly sedmým a osmým celočíselným parametrem funkce. Přístup k lokálním proměnným zůstává nezměněn.

5.4 Typové příklady předávání parametrů do funkcí

Všechny funkce uvedené dále v této kapitole jsou obsaženy v příkladu `functions-64`.

Jako první ukázka bude funkce `sum` pro sečtení dvou celých čísel. Ukázku lze rozvést na dvě varianty, a to pro parametry `int` a `long`.

```
int  sum_int( int t_a, int t_b );
long  sum_long( long t_a, long t_b );
```

Jak bude vypadat kód těchto funkcí v JSI, je ukázáno v následujícím kódu.

```
    ; function sum
sum_int:
    enter 0,0
    xor rax, rax
    mov eax, edi                ; parameter t_a
    add eax, esi                ; t_a += t_b
                                ; return value is in eax
    leave
    ret

sum_long:
    enter 0,0
    mov rax, rdi                ; parameter t_a
    add rax, rsi                ; t_a += t_b
                                ; return value is in rax
    leave
    ret
```

V uvedené ukázce kódu jsou předávány dva parametry přes registry ve standardtním pořadí: `RDI` a `RSI`. Funkce nepotřebují žádné lokální proměnné a všechny argumenty byly do funkce předány přes registy.

Pokud ve funkcích není potřeba manipulovat se zásobníkem, přebírat parametry, čí používat lokální proměnné, není nutno manipulovat s registry `RSP` *a* `RBP` *pomocí instrukcí* `ENTER` *a* `LEAVE`*.*

Předávané parametry ve funkci `soucet_int` jsou 32bitové, proto se výsledek počítá pouze z 32bitových registrů. Ve funkci `soucet_long` jsou parametry 64bitové a pro výpočet výsledku je použita plná velikost registrů.

Následujíc příklad je funkce `char_in_range` s parametry typu `char`. Úkolem funkce je ověřit, zda zadaný znak je v požadovaném intervalu. Výsledkem je návratová hodnota 0 nebo 1, podle toho zda je znak v intervalu.

Prototyp funkce je následující:

```
int char_in_range( char t_c, char t_low, char t_high );
```

Implementace v JSI následuje:

```
    ;int char_in_range( char t_c, char t_low, char t_high );
char_in_range:
    enter 0,0
    mov eax, 0       ; ret 0
    cmp dil, sil     ; cmp t_c, t_low
    jb .ret          ; out of range
    cmp dil, dl      ; cmp t_c, t_high
    ja .ret          ; out of range
    mov eax, 1       ; ret 1
.ret:
    leave
    ret
```

Z kódu je zřejmé, že pro řešení je potřeba použít výhradně dolní 8bitové části registrů, protože všechny 3 parametry jsou typu `char`. Dvěma podmíněnými skoky se řeší kontrola požadovaného intervalu mezi `SIL` (`low`) a `DL` (`high`). Návratová hodnota je v registru `EAX`.

Další funkce je první ukázkou práce s polem. Funkce `sum_int_array` má za úkol sečíst všechny prvky pole a součet bude návratová hodnota. Prototyp funkce je následující:

```
int sum_int_array ( int *t_array , int t_N );
```

První parametr funkce je ukazatel na předávané pole a druhý parametr udává jeho délku. Z prototypu funkce je patrné, že se ve funkci nebude řešit problém s přetečením součtu, protože návratová hodnota funkce je `int`. Kód v JSI může vypadat následovně:

```
    ; function sum_int_array ( int *t_array , int t_N )
sum_int_array :
    enter 0,0

    movsx rsi , esi               ; t_N
    mov rax , 0                   ; l_sum = 0
    mov rcx , 0                   ; i = 0
.for :
    cmp rcx , rsi                 ; i < N
    jge .endfor
    add eax , [ rdi + rcx * 4 ]   ; l_sum += t_array [ rcx ]
    inc rcx                       ; i++
    jmp .for
.endfor :
                                  ; result is in eax
    leave
    ret
```

Opakování je řešeno cyklem, kde počet opakování se řídí pomocí registru `RCX`, který současně slouží i jako index pro pole. Samotné sčítání prvků je pomocí instrukce `ADD`. Návratová hodnota funkce je pak součet v registru `EAX`.

Mezi prvními ukázkami kódu nemůže chybět ukázka práce s řetězcem. Jendou z nejjednodušších funkcí pro práci s řetězcem je nepochybně zjištění délky řetězce. Prototyp funkce je následující:

```
long str_length( char *t_str );
```

Funkce bude mít jediný parametr a tím je ukazatel na řetězec. Návratová hodnota bude `long`. Implementace kódu v JSI je následující:

```
    ;long str_length( char *t_str );
str_length:
    enter 0,0
    mov rax, 0                  ; l_len = 0
.back:
    cmp byte [ rdi + rax ], 0   ; while ( t_str[ l_len ] != 0 )
    je .done
    inc rax                     ; l_len++
    jmp .back
.done                           ; return rax
    leave
    ret
```

Počítání délky řetězce musí probíhat v cyklu, u kterého není předem znám počet opakování. Ukončovací podmínkou je nalezení znaku `'\0'`. Registr `RAX` v cyklu plní dvě role: je použít jako index v řetězci a současně počítá délku řetězce.

Dalším příkladem je opět práce s polem čísel typu `int`. Funkce bude počítat aritmetický průměr prvků pole. Kód funkce bude podobný kódu pro součet prvků pole, uvedený již dříve. Pro mezivýpočet součtu je ale možno použít 64bitový registr, aby během výpočtu nedošlo k přetečení.

```
int average_int_array ( int *t_array , int t_N );
```

První parametr funkce je ukazatel na předávané pole a druhý parametr udává jeho délku. Kód v JSI může vypadat následovně:

```
    ; function average_int_array
average_int_array :
    enter 0,0

    movsx rsi , esi                ; length of t_array
    mov rax , 0                    ; l_sum
    mov rcx , 0                    ; i = 0
.back :
    cmp rcx , rsi                  ; i < t_N
    jge .endfor
    movsx rdx , dword [ rdi + rcx * 4 ]
    add rax , rdx                  ; l_sum += t_array [ ecx ]
    inc rcx                        ; i++
    jmp .back
.endfor :
    cqo                            ; extension of rax to rdx
    movsx rcx , esi                ; t_N
    idiv rcx                       ; l_sum /= t_N
                                   ; result is in rax
    leave
    ret
```

V uvedené ukázce je 32bitová délka pole přenesena do registru `RCX` se znaménkovým rozšířením. Dále je pak každý prvek pole znaménkově rozšířen do registru `RDX` a přidán do celkového součtu. Před dělením je provedeno rozšíření registru `RAX` do `RDX` a dělí se délkou pole.

Dalším typovým příkladem může být funkce pro dělení dvou celých čísel, která bude vracet i zbytek po dělení přes ukazatel na proměnnou. Funkci lze implementovat opět pro `int` a pro `long`.

```
int  division_int ( int  t_a, int  t_b, int  *t_remainder  );
long  division_long ( long  t_a, long  t_b, long  *t_remainder  );
```

Kód funkcí v JSI je možno implementovat následovně:

```
    ; function  division
division_int :
    enter  0,0
    mov  rcx, rdx                  ; save  t_remainder
    mov  eax, edi                  ; parameter  t_a  to  eax
    cdq                            ; sign  extension  of  eax  do  edx
    idiv  esi                      ; eax /= t_b
                                   ; result  is  in  eax
                                   ; remainder  is  in  edx
    mov  [ rcx ], edx              ; *t_remainder  = edx
    ret

division_long :
    mov  rcx, rdx                  ; save  t_remainder
    mov  rax, rdi                  ; parameter  t_a  to  eax
    cqo                            ; extension  of  rax  to  rdx
    idiv  rsi                      ; rax /= t_b
                                   ; result  is  in  rax
                                   ; remainder  v  rdx
    mov  [ rcx ], rdx              ; *t_remainder  = rdx
    leave
    ret
```

Uvedené ukázký kódu jsou v obou případech téměř shodné, liší se jen velikost použitých registrů pro výpočet.

Pro ukládání zbytku je potřeba zachovat hodnotu třetího argumentu v registru `RDX`, který bude dělením přepsán.

Předchozí příklady ukázaly možnost předávání celočíselných parametrů do funkcí, předávání a použití ukazatele na pole celých čísel a předání parametru ukazatelem. Následující příklady budou zaměřeny na práci z řetězci.

Následujicí ukázka kódu provede otočení pořadí znaků v řetězci. Na začátku funkce bude použita standardní funkce `strlen` pro zjištění délky řetězce. Prototyp funkce v jazyce C má následující tvar:

```
char *strmirror( char *t_str );
```

Potřebný kód v JSI může být implementován např. následovně:

```
    ; function strmirror
strmirror:
    enter 0,0
    push rdi                    ; save rdi
    call strlen                 ; call strlen
    pop rdi                     ; restore rdi
                                ; in rax is length of string
    mov rcx, rdi                ; ptr. to first character
    mov rdx, rcx
    add rdx, rax
    dec rdx                     ; ptr. to last character
.back:
    cmp rcx, rdx                ; while ( ecx < edx )
    jae .end
    mov al, [ rcx ]             ; sel. of first and last char
    mov ah, [ rdx ]
    mov [ rcx ], ah             ; store back sel. chars
    mov [ rdx ], al
    inc rcx                     ; move to the right
    dec rdx                     ; move to the left
    jmp .back
.end:
    mov rax, rdi                ; return t_str
    leave
    ret
```

Z implementace je patrné, že před voláním funkce `strlen` je potřeba pomocí instrukcí `PUSH/POP` uložit hodnoty registru `RDI`, který je současně prvním parametrem funkce `strmirror` i `strlen`.

Další ukázkou je klasický úkol ze základů algoritmizace, převod celého čísla na řetězec. Prototyp funkce v jazyce C může být v následujícím tvaru:

```
char *int2str( unsigned long t_number, char *t_str );
```

Potřebný kód v JSI může být implementován např. následovně:

```
    ; function int2str
int2str:
    enter 0,0
    mov rax, rdi                ; t_number
    mov rcx, 10                 ; l_base of number system
    mov rdi, rsi                ; part of t_str. for mirror
    push rsi                    ; save t_str
    cmp rax, 0                  ; branches for < > = 0
    jg .positive
    jl .negative
    mov [ rsi ], word '0'       ; add to end of str "0\0"
    jmp .ret                    ; all is done
.negative:
    mov [ rsi ], byte '-'       ; sign at beginning of t_str
    inc rdi                     ; skip sign
    neg rax                     ; turn sign
.back:
    inc rsi                     ; t_str++
.positive:
    test rax, rax               ; while ( rax )
    je .end
    mov rdx, 0
    div rcx                     ; rax /= l_base
    add dl, '0'                 ; remainder += '0'
    mov [ rsi ], dl             ; *t_str = dl
    jmp .back
.end:
    mov [ rsi ], byte 0         ; *t_str = 0
                                ; rdi is t_str for mirror
    call strmirror
.ret:
    pop rax                     ; return value
    leave
    ret
```

6 Literatura

1. System V Application Binary Interface, Intel386 Architecture Processor Supplement, Fourth Edition, 1997

   abi-32.pdf

2. Intel 64 and IA-32 Architectures, Software Developer's Manual, Volume 2A: Instruction Set Reference, A-Z, Intel 2018

   intel-AZ.pdf

3. Michael Matz, jan Hubička, Andreas Jaeger, Mark Michell, System V Application Binary Interface, AMD64 Architecture Processor Supplement, 2013

   abi-64.pdf

4. The NASM Development Team, NASM - The Netwide Assembler 0.98, 2003

   nasm-0.98.pdf

5. The NASM Development Team, NASM - The Netwide Assembler 2.11, 2012

   nasm-2.12.pdf

6. Raymond Filiatreault, Simply FPU, 2003

   FPU-Tutorial.zip