

VŠB - Technická univerzita Ostrava  
Katedra informatiky, FEI

# Assembler x86

Studijní text pro předmět:

**Strojově orientované jazyky**

Ing. Petr Olivka, Ph.D.

2019

e-mail: [petr.olivka@vsb.cz](mailto:petr.olivka@vsb.cz)

<http://poli.cs.vsb.cz>

## Obsah

<b>1</b>	<b>Processor i486 a vyšší - 32bitový režim</b>	<b>4</b>
1.1	Registry . . . . .	4
1.2	Adresování . . . . .	7
1.3	Strojový kód, JSI, Assembler . . . . .	7
1.4	Datové typy . . . . .	7
<b>2</b>	<b>Spojování programů v JSI a v jazyce C</b>	<b>8</b>
2.1	Spojování C - C . . . . .	8
2.2	Spojování JSI a C . . . . .	10
2.3	Používání proměnných v JSI . . . . .	11
<b>3</b>	<b>Instrukční soubor</b>	<b>14</b>
3.1	Přesunové instrukce . . . . .	14
3.2	Logické a bitové instrukce . . . . .	16
3.3	Aritmetické instrukce . . . . .	18
3.4	Skokové instrukce . . . . .	19
3.5	Řetězcové instrukce . . . . .	21
3.6	Pomocné a řídicí instrukce . . . . .	22
<b>4</b>	<b>32bitové rozhraní C - JSI</b>	<b>23</b>
4.1	Návratové hodnoty z funkcí . . . . .	23
4.2	Používání registrů . . . . .	23
4.3	Volání funkcí s parametry . . . . .	24
4.3.1	Pořadí předávání parametrů . . . . .	24
4.3.2	Volání funkce, nastavení EBP . . . . .	25
4.3.3	Přístup k parametrům a lokálním proměnným . . . . .	26
4.3.4	Návrat z funkce, úklid parametrů . . . . .	26
4.3.5	Příklad funkce . . . . .	27
4.4	Typové příklady předávání parametrů do funkcí . . . . .	28
4.5	Použití řetězcových instrukcí . . . . .	32
<b>5</b>	<b>Procesory AMD a Intel - 64bitový režim</b>	<b>34</b>
5.1	Registry . . . . .	34
5.2	Adresování v 64bitovém režimu . . . . .	35
5.3	Instrukční soubor pro 64bitový režim . . . . .	35
<b>6</b>	<b>64bitové rozhraní C - JSI</b>	<b>36</b>
6.1	Návratové hodnoty funkcí . . . . .	36
6.2	Používání registrů . . . . .	36
6.3	Volání funkcí s parametry . . . . .	36
6.4	Typové příklady předávání parametrů do funkcí . . . . .	38
6.5	Použití řetězcových instrukcí . . . . .	43

<b>7</b>	<b>Čísla s desetinnou tečkou</b>	<b>44</b>
7.1	Fixed Point	44
7.1.1	Specifikace formátu Fixed Point	45
7.1.2	Převod čísla s desetinnou tečkou na Fixed Point a zpět	46
7.1.3	Sčítání a odčítání čísel Fixed Point	46
7.1.4	Násobení čísel Fixed Point	48
7.1.5	Dělení čísel Fixed Point	49
7.2	Float Point	50
7.2.1	Výpočty s čísly Float Point	52
7.2.2	Násobení Float Point čísel	52
7.2.3	Dělení Float Point čísel	54
7.2.4	Sčítání a odčítání Float Point čísel	55
<b>8</b>	<b>FPU</b>	<b>56</b>
8.1	Typové příklady	56
<b>9</b>	<b>SSE</b>	<b>61</b>
9.1	Registry SSE	61
9.2	Obsah registrů	62
9.3	Instrukce SSE	62
9.3.1	Instrukce přesunové	63
9.3.2	Instrukce přesunové se změnou pořadí čísel	64
9.3.3	Převod formátů čísel	65
9.3.4	Aritmetické operace	66
9.3.5	Bitové operace	68
9.3.6	Porovnávání čísel	69
9.4	Typové příklady použití SSE	70
<b>10</b>	<b>Počítání s velkými čísly</b>	<b>73</b>
10.1	Výpočty s čísly int32 a int64	73
10.1.1	Sčítání a odčítání čísla int64 a int32	73
10.1.2	Sčítání a odčítání čísel int64	75
10.1.3	Násobení čísla int64 číslem int32	76
10.1.4	Násobení čísel int64	77
10.1.5	Dělení čísla int64 číslem int32	78
10.2	Výpočty s čísly intN	80
10.2.1	Formát čísel intN	80
10.2.2	Délka čísla binárního a dekadického	80
10.2.3	Převod čísla intN na řetězec a zpět	81
10.2.4	Sčítání čísla intN a int32	82
10.2.5	Násobení čísla intN a int32	83
10.2.6	Dělení a zbytek po dělení čísla intN a int32	85
10.2.7	Implementace převodu čísla intN na řetězec a zpět	86
10.2.8	Sčítání a odčítání čísel intN	86

10.2.9 Bitový posun čísla $\text{intN}$ vlevo a vpravo o 1 bit . . . . .	88
10.2.10 Bitový posun vlevo a vpravo o více bitů . . . . .	89
10.2.11 Násobení čísel $\text{intN}$ . . . . .	90
10.2.12 Dělení čísel $\text{intN}$ . . . . .	91

**11 Literatura** **93**

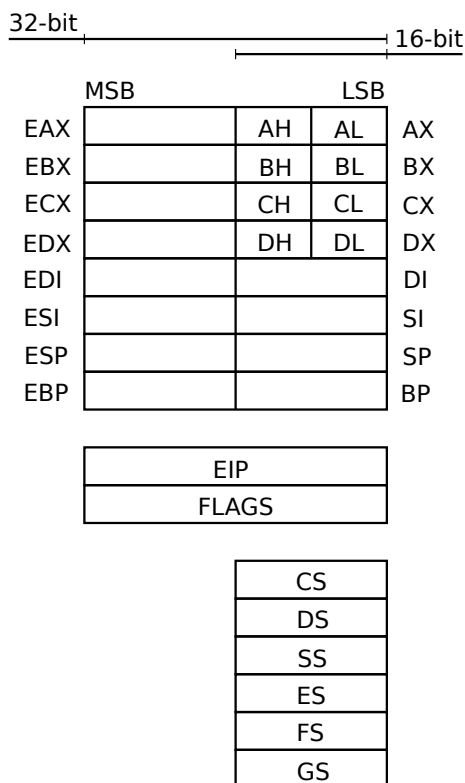
# 1 Procesor i486 a vyšší - 32bitový režim

Procesory i486 a vyšší jsou v technické literatuře dobře dokumentovány, ale dokumentace je rozsáhlá a obsahuje pro začínajícího i zkušeného programátora mnoho nadbytečných informací. Cílem tohoto textu je vymezit základní pojmy, principy a instrukce pro běžnou práci.

Z historického hlediska nemá smysl oddělovat postupný vývoj generací procesorů před verzí i486. Tato verze je zde považována za výchozí.

## 1.1 Registry

Celkový přehled registrů je na obrázku 1.



Obrázek 1: Přehled registrů procesoru i486

Procesory i486 obsahují 8 základních registrů velikosti 32 bitů pro všeobecné použití. Dále 6 registrů segmentových, stavový registr a čítač instrukcí.

Registry 32bitové pro všeobecné použití:

EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP.

Registry 16bitové:

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

Registry 8bitové:

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

Segmentové registry 16bitové:

DS (data), ES (extra), CS (code), SS (stack), FS, GS.

Čítač instrukcí EIP (IP):

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

Stavový registr FLAGS:

obsahuje stavové bity. Mezi programátorem nejpoužívanější 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:

- EAX, AX, AL je akumulátor. Používá se při násobení, dělení a v řetězových instrukcích.
- ECX je používán jako čítač.
- CL je použit jako čítač při bitových rotacích a posunech.
- EDX, DX je horní část argumentu při násobení a dělení.
- ESI, EDI je využíván jako indexový registr v řetězových instrukcích.

## 1.2 Adresování

Adresování rozdělujeme na 16 a 32bitový režim. Dále rozdělujeme adresování na přímé a nepřímé. Při přímém adresování uvádíme vždy konkrétní pevnou adresu. Programátor ovšem v praxi potřebuje přímou adresu je výjimečně, nejčastěji jsou za přímou adresu považovány adresy proměnných. Pokud nelze adresu určit přímo, může použít adresování nepřímé přes registry.

V 16bitovém režimu je formát:

[ Bázový + Indexový + Konstanta ],

kde indexové registry jsou dva: DI a SI, bázové registry také dva: BX a BP. Lze tedy vždy kombinovat jen jeden bázový s jedním indexovým, ale lze je používat i každý samostatně.

V 32bitovém režimu je adresování univerzálnější:

[ Bázový + Indexový \* Měřítko + Konstanta ].

Na místě bázového i indexového registru je možno použít kterýkoliv z osmi 32bitových registrů. Měřítko je jedno ze čtyř čísel: 1, 2, 4, a 8. Usnadňuje manipulaci s polem, jehož položky jsou jiné velikosti než 1 bajt.

## 1.3 Strojový kód, JSI, Assembler

Strojový kód je binární reprezentace strojových instrukcí, které procesor vykonává. V dnešní době ovšem žádný programátor nepíše programy přímo ve strojovém kódu, ale píše v „jazyce symbolických instrukcí“, tedy používá k vytvoření programu symbolické textové názvy jednotlivých instrukcí a registrů. Do strojového kódu jsou pak symbolické instrukce převáděny překladačem. V praxi bývá také JSI nazýván assemblerem.

## 1.4 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)
- DT - data TBYTE (80 bitů)



## 2 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.

### 2.1 Spojování C - C

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

---

```
// modul.c
int m_counter;
static int m_sum = 0;

static int inc_sum ()
{ m_sum++; }

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

---

Následuje hlavní část programu s funkcí `main`.

---

```
// main.c
// external function prototypes
int inc_sum ();
int inc_counter ();
// external variables
extern int m_sum;
extern int m_counter;

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

---

Nejprve je potřeba si uvědomit, jaký je rozdíl mezi funkcemi a proměnnými, u který je při jejich deklaraci uvedeno klíčové slovo `static`. V souboru `modul.c` je takto deklarována proměnná `m_sum` a funkce `inc_sum`. 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 uvedeme 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 můžeme uvést prototypy externích proměnných `m_counter` a `m_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` pak můžeme 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.

## 2.2 Spojování JSI a C

V následující ukázce `modul.asm` bude uveden kód, který rovnocenným způsobem nahrazuje `modul.c`.

---

```
; modul.asm
    bits 32                ; 32 bit code
    section .data         ; data section
    global m_counter     ; public symbol
m_counter    dd 0        ; variable m_couter
m_sum        dd 0        ; variable m_sum

    section .text        ; code section
    global inc_counter   ; public symbol

inc_sum :                ; function inc_sum
    inc dword [ m_sum ]
    ret

inc_counter :           ; function inc_counter
    inc dword [ m_counter ]
    call inc_sum        ; function calling
    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 `m_counter` a `m_sum`. V JSI platí pravidlo, že všechny symboly jsou lokální. Proto pro zveřejnění symbolu `m_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, že komentáře se ve zdrojovém kódu JSI označují středníkem. Instrukce se píše odsazené od levého okraje. Na levém okraji se píše 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 `soj-c-c.tgz` a `soj-c-asm.tgz`. Pro sestavení programů je potřeba použít příkaz `make`.

## 2.3 Používání proměnných v JSI

Pro přístup k proměnným je potřeba se nejprve podívat podrobněji alespoň na jednu instrukci. Nejjednodušší instrukcí je zajisté instrukce MOV:

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

Jako parametry instrukce lze použít registry (R), proměnné v paměti (M) a konstanty (K). Tyto 3 typy parametrů je možné použít v pěti různých kombinacích a to 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 další instrukce:

- 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 1.4: byte, word, dword, atd).

Následující příklad používání proměnných v JSI bude demonstrovat přístup ke globálním proměnným a to deklarovaným v jazyce C i JSI. Nejprve hlavní program v jazyce C.

---

```
// main.c
// public global variables
int c_number;
char c_char;
int c_iarray [ 10 ];
char c_text [] = "String declared in C\n";
// external variables
extern int a_counter;
extern char a_byte;
extern int a_numbers [];
extern char a_str [];
```

```

// external function
void changes ();

int main()
{
    changes ();
    // printf selected variables ...
}

```

---

Nyní následují ukázky používání proměnných v JSI. Přístup k proměnným deklarovaným v C i v JSI je zcela shodný.

---

```

; Example of the variables use in Assembly language
bits 32
section .data
; external variables
extern c_number, c_char, c_iarray, c_text
; list of public symbols
global a_counter, a_byte, a_numbers, a_str
a_counter dd 0 ; int
a_byte db 0 ; char
a_numbers dd 0,0,0,0,0 ; int[ 5 ]
; following string must be terminated by '\0'
a_str db 'Text defined in ASM', 10, 0
section .text
global changes
changes :
; integer numbers
mov eax, [ c_number ] ; eax = c_number
mov [ a_counter ], eax ; a_counter = eax
mov dword [ c_number ], 0 ; c_number = 0
; characters
mov byte [ a_byte ], 13 ; a_byte = 13
mov dl, [ c_char ] ; dl = c_char
; array elements access
mov ecx, [ c_iarray + 2 * 4 ]; ecx = c_iarray [ 2 ]
mov edx, 3
mov [ a_numbers + edx * 4 ], ecx
; a_numbers [ ebx ] = ecx
; access to characters in string
mov dh, [ c_text ] ; dh = c_text [ 0 ]
mov eax, 5
mov [ a_str + eax ], dh ; a_str [ 5 ] = dh
ret

```

---

V uvedeném příkladu je možno vidět používání adresování popsané v kapitole 1.2. Pro přístup k proměnným se vždy používá zápis se závorkami []. Tímto způsobem se jednoznačně označuje přístup do paměti. Obsah

závorky je pak adresa místa v paměti, kde je požadovaná hodnota. Proto při přístupu k proměnným se do závorky vkládá pouze název požadované proměnné. Každý symbol má překladačem přidělenou konstantní adresu.

Dále je možno dle informací v kapitole 1.2 uvést do závorky [] ještě až dva registry, kterými je možno výslednou hodnotu adresy změnit. Nejčastěji tak potřebujeme indexovat položky v polích. U jednoho registru, nazývaného formálně indexový, je navíc možno uvést i hodnotu násobku, odpovídající velikosti jedné položky v poli. V uvedeném příkladu je to vidět při přístupu k položkám pole `a_numbers` a `c_iarray`.

Výsledky provedené funkcí je možné v uvedeném příkladu následně vypsat ve funkci `main` pomocí `printf`. Další příklad použití proměnných v JSI je uveden v příkladu `soj-var.tgz`.

### 3 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,
- řetězcové,
- pomocné a řídicí.

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.

#### 3.1 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.

XLATB

Instrukce provede:  $AL = [EBX + AL]$ . Do registru AL se přesune AL-tý znak z tabulky, na kterou ukazuje EBX (BX).

LEA cíl, zdroj

Do operandu cíl se přesune adresa operandu zdroj.

LDS cíl, zdroj

Obsah paměti na kterou ukazuje cíl se přesune do DS a operandu cíl.

LES, LSS, LFS, LGS

Stejně jako LDS, ale použije se ES, SS, FS, GS.

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.

PUSHA

Uloží na vrchol zásobníku všech 8 registrů.

POPA

Obnoví z vrcholu zásobníku všech 8 registrů.

PUSHF

Stavový registr procesoru se uloží na vrchol zásobníku.

POPF

Hodnota na vrcholu zásobníku se přenesse do stavového registru.



LAHF

Přesune spodních 8 bitů stavového registru do AH.

SAHF

Přesune obsah AH do spodních 8 bitů stavového registru.

IN akumulátor, adresa

Přesun z portu daného operátorem *adresa* do akumulátoru AL, AX, EAX. Pokud je adresa velikosti 8 bitů, lze použít konstantu, jinak musí být adresa v registru DX.

OUT adresa, akumulátor

Přesun z akumulátoru do portu na dané adrese. Podobně jako instrukce IN.

### 3.2 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  $\text{cíl} = \text{cíl} \text{ and } \text{zdroj}$ .

TEST cíl, zdroj

Instrukce provede bitově  $\text{cíl} \text{ and } \text{zdroj}$ . Jde o operaci AND bez uložení výsledku.

OR cíl, zdroj

Instrukce provede bitově  $\text{cíl} = \text{cíl} \text{ or } \text{zdroj}$ .

XOR cíl, zdroj

Instrukce provede bitově  $\text{cíl} = \text{cíl} \text{ xor } \text{zdroj}$ .

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.

### 3.3 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  $\text{cíl} = \text{cíl} + \text{zdroj}$

**ADC** cíl, zdroj

Instrukce provede aritmetické sčítání včetně CF  $\text{cíl} = \text{cíl} + \text{zdroj} + \text{CF}$

**SUB** cíl, zdroj

Instrukce provede aritmetické odčítání  $\text{cíl} = \text{cíl} - \text{zdroj}$

**CMP** cíl, zdroj

Instrukce provede aritmetické odčítání  $\text{cíl} - \text{zdroj}$ , ale neuloží výsledek.

**SBB** cíl, zdroj

Instrukce provede aritmetické odčítání s výpůjčkou  $\text{cíl} = \text{cíl} - \text{zdroj} - \text{CF}$

**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).

**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).

**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) a výsledek se uloží do (AL, AX, EAX) a zbytek po dělení bude v (AH, DX, EDX).

#### 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).

#### 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.

### 3.4 Skokové instrukce

#### JMP cíl

Provádění programu se přenesse 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).

INT číslo

Vyvolá se programové přerušení číslo.

IRET

Návrat z přerušení.

### 3.5 Řetězcové instrukce

Řetězcové instrukce jsou vázány na povinnou konvenci používaných indexových registrů a velikost operandu:

ES:EDI - cílový operand.

DS:ESI - zdrojový operand.

B/W/D - velikost operandu 1, 2 a 4 byty. O tuto hodnotu se posouvají indexové registry.

DF - směr posunu (direction flag), nula nahoru, jednička dolů.

Před každou řetězcovou instrukcí je možno ještě použít prefix:

```
REP: while (ECX) { ECX--; ... }
```

```
REPE/Z: while (ECX && ZF) { ECX--; ... }
```

```
REPNE/NZ: while (ECX && !ZF) { ECX--; ... }
```

Řetězcové instrukce jsou jen přesunové a porovnávací:

MOVSB/W/D

Přesune jeden prvek ze zdroje do cíle. S prefixem může provést přesun bloku paměti.

LODSB/W/D

Přesune jeden prvek ze zdroje do akumulátoru (AL, AX, EAX).

STOSB/W/D

Přesune obsah akumulátoru (AL, AX, EAX) do cíle. S prefixem může vyplnit blok paměti požadovanou hodnotou.

SCASB/W/D

Porovnává obsah akumulátoru (AL, AX, EAX) s cílem: `null=akumulátor-ES:[EDI]`. S prefixem lze použít k vyhledání požadované hodnoty, nebo nalezení prvního rozdílu.

CMPSB/W/D

Porovnává obsah zdroje a cíle: `null=ES:[ESI]-DS:[EDI]`. S prefixem lze hledat první shodu nebo první rozdíl.

INSB/W/D

Přečte z portu na adresa DX jednu položku do cíle.

OUTSB/W/D

Jednu položku ze zdroje přesune na port na adrese DX.

### 3.6 Pomocné a řídicí 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.

SET`cc` `cíl`

Instrukce uloží do parametru `cíl` hodnotu 0 nebo 1 podle toho, zda je splněna podmínka `cc`. Podmínky byly vysvětleny u podmíněných skoků.

## 4 32bitové rozhraní C - JSI

Používání globálních proměnných, jak bylo uvedeno v kapitole 2.3, není při psaní programů ani obvyklé, ani pohodlné. Ve vyšších programovacích jazycích se proto při volání funkcí předávají parametry a návratové hodnoty přes zásobník a registry. Způsob předávání hodnot je standardizován v dokumentu "Application Binary Interface", zkráceně ABI. Čtvrté vydání toho dokumentu pro 32bitové rozhraní je přiloženo jako dokument `abi-32.pdf`

Dále bude na základě tohoto dokumentu podrobně vysvětleno, jak se předávají návratové hodnoty z funkcí a jak se provádí předávání parametrů do funkcí.

### 4.1 Návratové hodnoty z funkcí

Funkce mohou mít dle typu návratové hodnoty velikost 8, 16, 32 i 64 bitů. Této velikosti návratové hodnoty odpovídá i část registru, nebo jejich kombinace, které se pro návratovou hodnotu používají. Podle velikosti a typu návratové hodnoty jsou registry použity následovně:

- 8 bitů - registr AL,
- 16 bitů - registr AX,
- 32 bitů - registr EAX,
- 64 bitů - registry EAX-EDX,
- float/double - registr FPU ST0.

### 4.2 Používání registrů

Ve funkcích je potřeba dodržovat několik základních pravidel pro práci s registry.

- Registry EAX, ECX a EDX lze používat libovolně a není potřeba jejich hodnotu na konci funkce obnovovat.
- Registry EBX, ESI a EDI lze používat libovolně, na konci funkce je potřeba jejich hodnotu obnovit. Tyto registry mohou být využívány pro lokální proměnné.
- Registry ESP a EBP jsou používány pro práci se zásobníkem dle popsaného postupu.
- V registru FLAGS je potřeba vždy na konci funkce zajistit, aby bit DF byl nastavena vždy na hodnotu 0.



- Pokud byly použity registry FPU, je potřeba je na konci funkce všechny uvolnit. Pouze v případě, že se pomocí ST0 vrací návratová hodnota, bude nastavena právě v tomto registru.

### 4.3 Volání funkcí s parametry

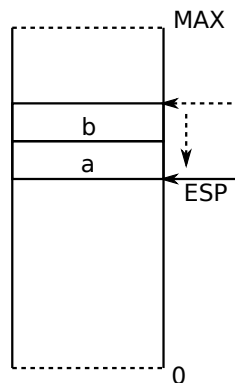
Předávání parametrů do funkcí je o něco komplikovanější, než předávání návratových hodnot. Bude proto potřeba vysvětlit celý princip krok po kroku.

#### 4.3.1 Pořadí předávání parametrů

Mějme následující hlavičku funkce pro sčítání dvou čísel:

```
int sum( int a, int b );
```

Parametry je možno předávat zleva i zprava. Předávání parametrů zleva je konvence používaná v jazyce Pascal. V jazyce C se používá konvence, kdy se parametry předávají zprava. Předávání parametrů se provádí přes zásobník pomocí instrukce PUSH. Celou situaci je možno znázornit na obrázku 2.



Obrázek 2: Uložení parametrů na zásobník

Pro správné pochopení principu předávání parametrů, je potřeba si uvědomit, že instrukce PUSH vkládá data na zásobník, jehož vrchol daný registrem ESP se posouvá směrem k 0. Pozice vrcholu zásobníku před vložením parametrů a a b je znázorněna na obrázku 2 vodorovnou čárkovanou čarou. Po vložení parametrů se aktuální pozice ESP posunula do pozice označené plnou čarou.

V tomto okamžiku je možno zavolat funkci sum pomocí instrukce CALL.

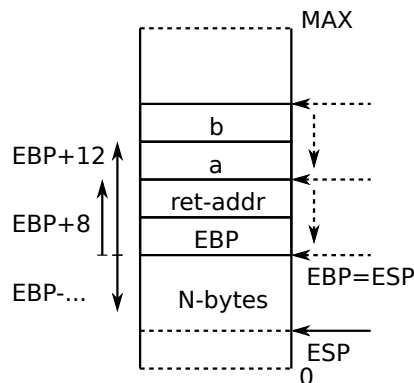
### 4.3.2 Volání funkce, nastavení EBP

Vykonávání kódu se do funkce přenesou instrukcí `CALL`. Tato instrukce před skokem do požadované funkce uloží na vrchol zásobníku adresu instrukce za instrukcí `CALL`. Je tak možno se pomocí instrukce `RET` z volané funkce vrátit do místa, odkud byla funkce volána.

Po provedení skoku do funkce je potřeba vyhradit prostor pro lokální proměnné a nastavit registr `EBP`, který bude následně sloužit pro přístup k lokálním proměnným a parametrům. Pro toto vstupní nastavení se používá instrukce `ENTER N,0`. Instrukci `ENTER` je možno pro pochopení její činnosti rozepsat na 3 instrukce:

```
; rozepsání instrukce ENTER N,0
push ebp          ; uložení EBP volající funkce
mov ebp, esp     ; do EBP se uloží aktuální hodnota ESP
sub esp, N       ; posunem ESP se alokuje prostor pro
                 ; lokální proměnné
```

Stav zásobníku po vstupu do funkce a provedení instrukce `ENTER` je vidět na obrázku 3.



Obrázek 3: Vstup do funkce a inicializace EBP

Na obrázku 3 je vidět, že za uložené parametry `a` a `b` uložila instrukce `CALL` návratovou adresu. Následně instrukce `ENTER` na zásobník přidala hodnotu `EBP` předchozí funkce a posunula `ESP` o velikost lokálních proměnných.

Na celé situaci je velmi důležité pro další činnost nastavení registru `EBP`. Pozice tohoto registru bude vždy a v každé funkci stejná, a to bez ohledu na počet předávaných parametrů a velikost lokálních proměnných!

### 4.3.3 Přístup k parametrům a lokálním proměnným

Jak bylo uvedeno v předchozí podkapitole, po vstupu do funkce se vždy nastaví registr EBP na pevně stanovené místo, tj. 4 bajty pod návratovou adresu. Tato pozice umožňuje přistupovat vždy stejným způsobem k parametrům uloženým na zásobníku, i k lokálním proměnným. Jak je znázorněno na obrázku 3, k prvnímu parametru je možno přistupovat vždy pomocí [EBP+8], ke druhému [EBP+12] a stejně tak lze pokračovat na další parametry. Přístup k lokálním proměnným se realizuje opačným směrem, například pro první 32bitový parametr je možno přistupovat pomocí [EBP-4].

### 4.3.4 Návrat z funkce, úklid parametrů

Posledním krokem funkce je návrat zpět do volající funkce. Samotnému návratu ještě předchází instrukce LEAVE. Tuto instrukci je možno rozepsat podobně jako byla rozepsána instrukce ENTER.

```
; rozepsání instrukce LEAVE
mov esp, ebp      ; návrat ESP na definované místo
pop ebp          ; obnovení EBP předchozí funkce
```

Instrukcí LEAVE se vrací stav zásobníku do situace na obrázku 2. Po instrukci LEAVE již může následovat instrukce RET, kterou se provádění kódu přenesou zpět do volající funkce.

Volající funkce je následně zodpovědná za úklid parametrů ze zásobníku. Při malém počtu parametrů se provádí úklid pomocí instrukce POP Registr (nejčastěji ECX) a v případě 3 a více parametrů se provádí přímo potřebný posun registru ESP o požadovaný počet bajtů pomocí instrukce ADD.

### 4.3.5 Příklad funkce

Na začátku této podkapitoly byl představena hlavička funkce pro součet dvou čísel:

```
int sum( int a, int b );
```

Jak bude vypadat kód této funkce v JSI a jakým způsobem se tato funkce volá, bude ukázáno v následujícím kódu.

Listing 1: Implementace jednoduché funkce a ukázka jejího volání

---

```
; function sum
sum:
    enter 0,0

    mov eax, [ ebp + 8 ]      ; parameter a
    add eax, [ ebp + 12 ]    ; a += b
                                ; return value is in eax

    leave
    ret

.....
; function call: sum( ecx, 10 )
push dword 10                ; parameter b
push ecx                     ; parameter a
call sum                     ; function call
add esp, 8                   ; remove a and b from the stack
; result is in eax
.....
```

---

V uvedené ukázce kódu jsou předávány dva parametry zprava. Po vstupu do funkce se instrukcí ENTER nealokují žádné lokální proměnné. Do registru EAX se ukládá hodnota prvního parametru a následně se k této hodnotě přičítá hodnota druhého parametru. Tím je připravena návratová hodnota v registru EAX.

Následuje uvedení zásobníku do původního stavu pomocí instrukce LEAVE a návrat do kódu volající funkce instrukcí RET.

V kódu volající funkce se musí provést úklid parametrů ze zásobníku a pak již lze využít získaný výsledek funkce z registru EAX.

## 4.4 Typové příklady předávání parametrů do funkcí

Všechny funkce uvedené dále v této kapitole jsou obsaženy v `soj-fun32.tgz`.

První příklad na předávání dvou celočíselných parametrů byl uveden v předchozí podkapitole.

Jako další příklad je možno uvést práci s polem celých čísel. Tímto příkladem může být funkce pro výpočet aritmetického průměru čísel pole. Prototyp funkce v jazyce C bude následující:

```
int average ( int *array , int N );
```

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

---

```
; function average
average :
    enter 0,0
    mov ecx, [ ebp + 12 ]      ; length of array
    mov edx, [ ebp + 8 ]      ; *array
    mov eax, 0                 ; sum
.back :
    add eax, [ edx + ecx * 4 - 4 ] ; sum+=array[ecx-1]
    loop .back
    cdq                        ; extension of eax to edx
    idiv dword [ ebp + 12 ]    ; sum /= N
                                ; result in eax
    leave
    ret
```

---

V uvedené ukázce je ukazatel na předávané pole uložen do registru `EDX`. Pro průchod celým polem je použita instrukce `LOOP`, která je řízena registrem `ECX`, kde se na začátku funkce uloží délka pole. Registr `ECX` je současně použit pro adresování prvků pole.

Sčítání se provádí v registru `EAX`. Po ukončení smyčky se tato hodnota znaménkově rozšíří a provede se dělení pomocí funkce `IDIV` délkou pole. Výsledek je uložen v registru `EAX`.

Ve funkci nedošlo k přepsání žádného registru, který by bylo potřeba na konci funkce obnovovat.

## Lokální návěští

Ve funkci `average` je možno si všimnout jedné vlastnosti návěští, která dosud nebyla zmíněna. U návěští `.back` je na začátku názvu uvedena tečka. Tímto způsobem se definují tzv. lokální návěští. Jejich platnost je vždy pouze mezi dvěma globálními návěštími, která tečkou nezačínají. Vzhledem k tomu, že globální návěští je potřeba pouze u vstup do funkcí, pak lze pro návěští v kódu funkcí používat výhradně lokálních návěští.

Dalším typovým příkladem může být funkce pro dělení dvou celých čísel, která bude vracet i zbytek po dělení přes ukazatel na proměnnou.

```
int division( int a, int b, int *remainder );
```

Kód funkce v JSI je možno implementovat následovně:

---

```
; function division
division :
    enter 0,0
    mov eax, [ ebp + 8 ]      ; parameter a to eax
    cdq                     ; extension of eax to edx
    idiv dword [ ebp + 12 ]  ; eax /= b
                             ; result is in eax
                             ; remainder in edx
    mov ecx, [ ebp + 16 ]    ; remainder
    mov [ ecx ], edx        ; *remainder = edx
    leave
    ret
```

---

Uvedená ukázka se do značné míry shoduje s dříve uvedenou funkcí `sum`. Navíc je uvedeno předávání výsledku nejen přes `EAX`, ale také přes ukazatel, který je třetím parametrem funkce. Použití tohoto ukazatele je prakticky shodné s použitím pole v příkladu `average`. Na zásobníku je uložena adresa paměti, kam se bude ukládat výsledek. V tomto příkladu je to právě místo pro uložení požadovaného zbytku po dělení.

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.

Následující 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 *str );
```

Potřebný kód v JSI může být implementován např. následovně:

---

```

; function strmirror
strmirror :
    enter 0,0
    push dword [ ebp + 8 ]      ; passing *str to strlen
    call strlen                ; call strlen
    pop ecx                    ; clean stack
                                ; length of string in eax
    mov ecx, [ ebp + 8 ]       ; first character of string
    mov edx, ecx
    add edx, eax
    dec edx                    ; last character of string
.back :
    cmp ecx, edx               ; while ( ecx < edx )
    jae .end
    mov al, [ ecx ]            ; sel. of first and last char
    mov ah, [ edx ]
    mov [ ecx ], ah           ; store sel. chars back
    mov [ edx ], al
    inc ecx                    ; move to the right
    dec edx                    ; move to the left
    jmp .back
.end :
    mov eax, [ ebp + 8 ]
    leave
    ret

```

---

V kódu funkce je patrné, že je potřeba dbát na správnou velikost operandů instrukcí. Znaky v řetězci jsou velikosti 8 bitů a proto je potřeba správně používat 8bitové registry AL a AH.

Posun dvou posouvajících se indexů proti sobě zleva a zprava je v kódu zřejmý.

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( int number, char *str );
```

Potřebný kód v JSI může být implementován např. následovně:

---

```
; function int2str
int2str:
    enter 8,0
    mov eax, [ ebp + 8 ]      ; number
    mov ecx, [ ebp + 12 ]    ; *str
    mov [ ebp - 4 ], ecx     ; part of str. for mirror
    mov [ ebp - 8 ], dword 10 ; base of number system
    cmp eax, 0              ; branches for < > = 0
    jg .positive
    jl .negative
    mov [ ecx ], word '0'    ; add to end of str "0\0"
    jmp .ret                ; all is done
.negative:
    mov [ ecx ], byte '--'   ; sign at beginning of str
    inc dword [ ebp - 4 ]    ; skip sign
    neg eax                  ; turn sign
.back:
    inc ecx                  ; str++
.positive:
    test eax, eax           ; while ( eax )
    je .end
    mov edx, 0
    div dword [ ebp - 8 ]   ; eax /= base
    add dl, '0'             ; remainder += '0'
    mov [ ecx ], dl        ; *str = dl
    jmp .back
.end:
    mov [ ecx ], byte 0     ; *str = 0
    push dword [ ebp - 4 ]  ; begin of str. for mirror
    call strmirror
    pop ecx
.ret:
    mov eax, [ ebp + 12 ]   ; return value is str
    leave
    ret
```

---



## 4.5 Použití řetězcových instrukcí

Základní popis řetězcových instrukcí byl popsán v kapitole 3.5. Před každým použitím řetězcové instrukce je potřeba správně nastavit potřebné indexové registry a segmentový registr ES. Příznakový bit DF je automaticky nastaven na 0, což znamená, že se indexové registry budou automaticky inkrementovat.

Následující kód ukazuje použití řetězcové instrukce pro zjištění délky řetězce. V tomto případě je výhodné využít instrukci SCAS a hledaným znakem bude ukončovací znak řetězce - '\0'. Prototyp funkce v jazyce C bude následující:

```
int strlen( char *str );
```

Kód funkce v JSI:

---

```
; function strlen
strlen :
    enter 0,0
    mov edi, [ ebp + 8 ]
    push ds
    pop es                ; es = ds
    mov ecx, -1          ; ecx = MAX
    mov al, 0            ; searched character '\0'
    ; cld                ; not necessary, DF is 0
    repne scasb          ; searching
    inc ecx              ; length without '\0'
    not ecx              ; turn sign
    mov eax, ecx         ; string length
    leave
    ret
```

---

Jako druhý příklad použití řetězcových instrukcí bude kód funkce pro odstranění všech mezer z řetězce. Pro tento případ je vhodná dvojice instrukcí LODS a STOS. Funkce bude mít v jazyce C následující prototyp:

```
char * strnospaces ( char *str );
```

Kód funkce lze v JSI implementovat následovně:

---

```
; function strnospaces
strnospaces :
    enter 0,0
    push edi                ; save registers
    push esi
    mov edi, [ ebp + 8 ]
    mov esi, edi            ; esi = edi
    push ds
    pop es                  ; es = ds
    ; cld                   ; not necessary, DF is 0
.back :
    lodsb                   ; al = [ esi++ ]
    test al, al
    jz .end                 ; end of string
    cmp al, ' '
    je .back                ; skip space
    stosb                   ; [ edi++ ] = al
    jmp .back
.end :
    stosb                   ; [ edi ] = '\0'
    mov eax, [ ebp + 8 ]    ; return value
    pop esi                 ; restore registers
    pop edi
    leave
    ret
```

---

## 5 Procesory AMD a Intel - 64bitový režim

### 5.1 Registry

Pro 64bitový procesor došlo firmou AMD k rozšíření 32bitové sady registrů na sadu registrů 64bitových. 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.

Při přechodu na 64bitový procesor však nebyly pouze rozšířeny registry, ale došlo také k rozšíření počtu pracovních registrů na dvojnásobek. Celkový přehled registrů je na obrázku 4.

	64-bit				32-bit	
	MSB		LSB		16-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 4: Přehled registrů 64bitového procesoru

Celá původní sada pracovních registrů byla rozšířena na 64 bitů a pro přístup k celému 64bitovému registru je v názvech registrů změněno Exx na Rxx. Nová skupina registrů je pojmenována jako R8-R15 a spodních 8/16/32 bitů je pojmenováno jako RxL/RxW/RxD. Je také možno přistupovat i ke spodním 8 bitům registrů SI, DI, SP, BP, což dříve možné nebylo.

Další registry zůstaly beze změny. Pouze registr EIP byl samozřejmě rozšířen na 64bitovou verzi RIP.

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

## 5.2 Adresování v 64bitovém režimu

Adresování zůstalo prakticky shodné v 64bitovém režimu, jako bylo v režimu 32bitovém. Přestože je možno používat i 32bitové registry, je potřeba používat důsledně registry 64bitové. Používání menších registrů není v principu správné.

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

## 5.3 Instrukční soubor pro 64bitový režim

Instrukční sada zůstala zachována a všechny dosud uvedené instrukce mohou využívat 64bitové registry i proměnné v paměti. V 64bitovém režimu však není možno používat instrukce `POPA` a `PUSHA`.

Jediné rozšíření, které již bylo zmíněno, je instrukce `CQO`, která rozšiřuje sadu instrukcí `CBW`, `CWD`, `CDQ` pro 64bitový režim. Instrukce pro znaménkové rozšíření registru `RAX` do `RAX-RDX`. Používá se před znaménkovým dělením.

## 6 64bitové rozhraní C - JSI

### 6.1 Návrátové hodnoty funkcí

Způsoby předávání návratových hodnot zůstaly prakticky shodné i 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.

### 6.2 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.

### 6.3 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:

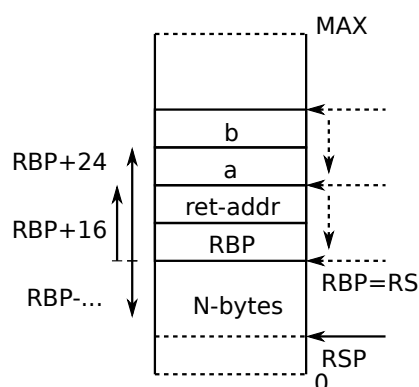
RDI, RSI, RDX, RCX, R8 a R9.

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 zásobník stejným způsobem, jak tomu bylo v 32bitovém režimu. Popis práce se zásobníkem, přístup k parametrům a k lokálním proměnným byl popsán v předchozím textu. Velikost všech uložených údajů na zásobníku je však dvojnásobné velikosti.



Obrázek 5: Zásobník v 64bitovém režimu

Na obrázku 5 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.

## 6.4 Typové příklady předávání parametrů do funkcí

V následujícím textu bude proveden přepis funkcí, jejichž kód pro 32bitový režim byl uveden v kapitole 4. Bude tak možno přímo porovnat rozdíly mezi 64bitovým a 32bitovým režimem. Všechny funkce uvedené dále v této kapitole jsou obsaženy v `soj-fun64.tgz`.

Jako první ukázka bude opět funkce `sum`. Ukázkou lze rozvést na dvě varianty, a to pro parametry `int` a `long`.

```
int sum_int( int a, int b );
long sum_long( long a, long b );
```

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

---

```
; function sum
sum_int :
    xor rax, rax
    mov eax, edi                ; parameter a
    add eax, esi                ; a += b
                                ; return value is in eax
    ret

sum_long :
    mov rax, rdi                ; parameter a
    add rax, rsi                ; a += b
                                ; return value is in rax
    ret
```

---

V uvedené ukázce kódu jsou předávány dva parametry přes registry ve standardním pořadí: RDI a RSI. Funkce nepotřebují žádné lokální proměnné a všechny argumenty byly do funkce předány přes registry, není proto potřeba manipulovat s registry ESP a EBP 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.

Dalším příkladem je práce s polem čísel typu `int`. Funkce bude počítat aritmetický průměr prvků pole. Pro mezivýpočet součtu je možno použít 64bitový registr, aby během výpočtu nedošlo k přetečení.

```
int average ( int *array, int N );
```

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

---

```
; function average
average :
    movsx rcx, esi                ; length of array
    mov rax, 0                    ; sum
.back :
    movsx rdx, dword [ rdi + rcx * 4 - 4 ]
    add rax, rdx                  ; sum += array[ ecx - 1 ]
    loop .back

    cqo                          ; extension of rax to rdx
    movsx rcx, esi                ; N
    idiv rcx                      ; sum /= N
                                   ; result is in rax

    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 deleni_int ( int a, int b, int *remainder );
long deleni_long ( long a, long b, long *remainder );
```

Kód funkcí v JSI je možno implementovat následovně:

---

```
; function division
division_int :
    mov rcx, rdx                ; save *remainder
    mov eax, edi                ; parameter a to eax
    cdq                         ; sign extension of eax do edx
    idiv esi                     ; eax /= b
                                ; result is in eax
                                ; remainder is in edx

    mov [ rcx ], edx            ; *remainder = edx
    ret

division_long :
    mov rcx, rdx                ; save *remainder
    mov rax, rdi                ; parameter a to eax
    cqo                         ; extension of rax to rdx
    idiv rsi                     ; rax /= b
                                ; result is in rax
                                ; remainder v rdx

    mov [ rcx ], rdx            ; *remainder = rdx
    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.

Následující 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 *str );
```

Potřebný kód v JSI může být implementován např. následovně:

---

```
; function strmirror
strmirror :
    push rdi                ; save rdi
    call strlen             ; call strlen
    pop rdi                 ; restore rdi
                            ; in rax is length of string
    mov rcx, rdi            ; first character of string
    mov rdx, rcx
    add rdx, rax
    dec rdx                  ; last character of string
.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 value
    ret
```

---

Z implementace je patrné, že 64bitová verze se od předešlé 32bitové liší použitou velikostí ukazatelů a rozdíl je také vidět při volání funkce `strlen`. V tomto kódu neslouží instrukce PUSH/POP k předání parametru do funkce, ale k uložení 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( long number, char *str );
```

Potřebný kód v JSI může být implementován např. následovně:

---

```
; function int2str
int2str:
    mov rax, rdi                ; number
    mov rcx, 10                 ; base of number system
    mov rdi, rsi                ; part of str. for mirror
    push rsi                     ; save 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 str
    inc rdi                       ; skip sign
    neg rax                       ; turn sign
.back:
    inc rsi                       ; str++
.positive:
    test rax, rax                ; while ( rax )
    je .end
    mov rdx, 0
    div rcx                       ; rax /= base
    add dl, '0'                   ; remainder += '0'
    mov [ rsi ], dl              ; *str = dl
    jmp .back
.end:
    mov [ rsi ], byte 0           ; *str = 0
                                   ; rdi is str for mirror
    call strmirror
.ret:
    pop rax                       ; return value
    ret
```

---

## 6.5 Použití řetězcových instrukcí

Následující kód ukazuje použití řetězcové instrukce pro zjištění délky řetězce. Prototyp funkce v jazyce C bude následující:

```
long strlen ( char *str );
```

Kód funkce v JSI:

---

```
; function strlen
strlen :
    mov ax, ds
    mov es, ax                ; es = ds
    mov rcx, -1              ; rcx = MAX
    mov al, 0                 ; searched character '\0'
    repne scasb              ; searching
    inc rcx                   ; length without '\0'
    not rcx                   ; turn sign
    mov rax, rcx             ; string length
    ret
```

---

Jako druhý příklad použití řetězcových instrukcí bude funkce pro odstranění všech mezer z řetězce. Funkce bude mít v jazyce C následující prototyp:

```
char *strnospaces ( char *str );
```

Kód funkce lze v JSI implementovat následovně:

---

```
; function strnospaces
strnospaces :
    mov rsi, rdi              ; rsi = rdi
    mov rdx, rdi              ; save rdi
    mov ax, ds
    mov es, ax                ; es = ds
    ; cld                     ; not necessary, DF is 0
.back :
    lodsb                     ; al = [ rsi++ ]
    test al, al
    jz .end                   ; end of string
    cmp al, ' '
    je .back                  ; skip space
    stosb                     ; [ rdi++ ] = al
    jmp .back
.end :
    stosb                     ; [ rdi ] = '\0'
    mov rax, rdx              ; return value
    ret
```

---

## 7 Číslo s desetinnou tečkou

V praxi je možno se v dnešní době setkat nejčastěji s čísly s desetinnou tečkou ve dvou formátech:

- Float Point - čísla s plovoucí desetinnou tečkou,
- Fixed Point - čísla s desetinnou tečkou na pevné pozici.

V následujícím textu bude věnován prostor popisu obou formátů. Jako první bude popsán formát Fixed Point, který je v praxi používaný nejdéle.

### 7.1 Fixed Point

Číslo Fixed Point byla prvním formátem čísel s desetinnou tečkou, který se v počítačích využíval. Hlavní nevýhodou těchto čísel, jak bude dále vysvětleno, je jejich omezený rozsah. Na druhé straně mají výhodu v tom, že pracují s konstantní přesností. To o číslech s plovoucí desetinnou tečkou neplatí. (Stačí se např. podívat na výraz (19), ze kterého je zřejmé, že pro velké hodnoty  $Q$  bude docházet k tak velkému posunu mantisy čísla, že se výsledná hodnota mantisy ve výsledku vůbec neprojeví.)

Další nevýhodou desetinných čísel je, že převod mezi desítkovou a binární soustavou není možno ve většině případů provádět bez nepřesností. Stačí se podívat na převod čísla 0.1 do dvojkové soustavy:

$$(0.1)_{10} = (0.00011)_2$$

Vyjádření čísla 0.1 je ve dvojkové soustavě vyjádřeno pomocí periodicky se opakující čtveřice čísel. Protože má každé číslo omezený počet platných číslic, je desetinné číslo v desítkové soustavě v počítači uloženo s odchylkou. Tato odchylka bude navíc narůstat se zvyšující se celou částí čísla, protože se tím současně zkracuje jeho desetinná část.

Pokud by bylo např. potřeba provádět nastavování určité veličiny s krokem 0.1, bude po několika opakovaných změnách hodnoty nemožné vyjádřit výslednou hodnotu s přesností právě na jednu desetinu.

Podobný postup je také zcela nepřijatelný pro práci s peněžními prostředky. Není v žádném případě možné, aby manipulací se stavem peněžního účtu docházelo k nepřesnostem.

V praxi lze najít i další příklady, kdy je potřeba pracovat s definovanou přesností. A právě pro tyto případy aplikací je vhodné použití čísel Fixed Point.

### 7.1.1 Specifikace formátu Fixed Point

Čísla s pevnou desetinnou tečkou jsou čísla s pevně daným počtem číslic před a za desetinnou tečkou. Formát čísla se označuje obvykle dvojicí čísel oddělených dvojtečkou:

$$C : D \quad (1)$$

**C** - znamená celkový počet platných číslic,

**D** - označuje počet míst za desetinnou tečkou z celkového počtu číslic.

K tomuto formátu musí být ještě obvykle slovně doplněna informace, zda se jedná o počty číslic v desítkové, nebo v binární číselné soustavě.

Vzhledem k pevně stanovenému počtu platných číslic, je číselný rozsah omezený na daný počet řádů v dané číselné soustavě. V tomto ohledu nabízí programátorům formát Float Point se svým exponentem daleko větší možnosti.

Jako první příklad může posloužit hodnota reprezentující teplotu v řídicím systému ve formátu: 4:1 v dekadické soustavě. Číslo s pevnou desetinnou tečkou v tomto formátu má celkem 4 platné číslice, z toho jednu desetinnou. Lze tak vyjádřit teploty v intervalu  $\langle 000.0, 999.9 \rangle$ . Pokud se bude využívat i znaménko, bude rozsah hodnot  $\langle -999.9, 999.9 \rangle$

Pro uložení tohoto formátu čísla do paměti počítače se obvykle použije jakýkoliv celočíselný typ, který je schopen pracovat s hodnotami v požadovaném rozsahu. Pro uvedený příklad by dostačovalo i znaménkové 16bitové číslo. Desetinná tečka se do čísla vkládá až po převodu do dekadické soustavy při prezentaci výsledku.

Jakékoliv přičtení a odečtení jednotky celého čísla znamená, že se hodnota změní o požadovaný počet desetin, a to bez jakékoliv ztráty přesnosti převodem mezi číselnými soustavami.

Dalším příkladem Fixed Point formátu může být formát 32:8 v binární soustavě. Tento formát dává k dispozici 24bitovou celočíselnou část, což je v případě znaménkového čísla rozsah  $\langle -16777216, 16777215 \rangle$ . Kromě toho umožňuje provádět výpočty s přesností  $2^{-8}$ . Tento formát bývá využíván v grafických aplikacích, např. pokud není k dispozici jednotka FPU, nebo její použití není nezbytné.

Zobrazení grafických informací se provádí v celých bodech a rozlišení grafických rozhraní je mnohem menší, než rozsah hodnot tohoto formátu. Při zobrazování je však např. pro potřeby antialiasingu potřeba provádět výpočty na několik desetinných míst. A pro tyto účely je tento formát naprosto dostačující.

Tento formát bývá také využíván při vykreslování vektorového písma.

### 7.1.2 Převod čísla s desetinnou tečkou na Fixed Point a zpět

Jak již bylo v textu dříve zmíněno, čísla Fixed Point jsou v počítači obvykle reprezentována jako čísla celá. Je proto potřeba si vyjádřit převodní vztah mezi desetinnými čísly a čísly celými, které představují číslo Fixed Point.

Nejprve je potřeba zavést značení čísel desetinných a jejich celočíselné reprezentace.

Číslo s desetinnou tečkou se bude označovat velkými písmeny abecedy, např.  $A, B, C, \dots$

Jeho celočíselná reprezentace (Fixed Point) bude označována odpovídajícím velkým písmenem, ale s čárkou, např.  $A', B', C', \dots$

Převod mezi těmito čísly bude dán jako posun desetinné tečky o potřebný počet řádů dle (1). Tento posun se bude vyjadřovat jako konstanta  $Z$  dle vztahu:

$$Z = z^D, \quad (2)$$

kde  $z$  je základ číselné soustavy, nejčastěji 2 nebo 10. Exponent  $D$  odpovídá počtu desetinných míst formátu čísla Fixed Point dle (1).

Převodní vztah mezi čísly desetinnými a Fixed Point budou dle následujících vztahů:

$$A' = A \cdot Z, \quad (3)$$

$$A = \frac{A'}{Z}. \quad (4)$$

Uvedené vztahy (3) a (4) představují ve dvojkové soustavě bitové posuny vlevo a vpravo, v soustavě dekadické operaci násobení a dělení. (V případě textového vyjádření čísla se však jedná jen o pozici desetinné tečky v zobrazeném čísle.)

### 7.1.3 Sčítání a odčítání čísel Fixed Point

Pokud bude provedeno sčítání dvou desetinných čísel  $A$  a  $B$ , bude výsledkem číslo  $C$  dle vztahu:

$$C = A + B \quad (5)$$

Jaký bude výsledek sčítání dvou Fixed Point čísel  $A'$  a  $B'$  je možno ověřit s použitím vztahů (3) a (4):

$$\begin{aligned} A' + B' &= A \cdot Z + B \cdot Z \\ &= Z \cdot (A + B) \\ &= Z \cdot C \\ &= C' \end{aligned} \quad (6)$$

Ze vztahu (6) je zřejmé, že součtem dvou čísel Fixed Point je opět číslo Fixed Point. Je proto možné sčítat dvě čísla Fixed Point stejného formátu, aniž by bylo potřeba výsledek sčítání upravovat.

Ze vztahu (6) je také zřejmé, že stejný postup lze aplikovat i na odčítání a výsledek bude opět číslo Fixed Point ve stejném formátu, jako čísla odčítaná.

Jak bylo v této kapitole vysvětleno, implementace sčítání a odčítání Fixed Point čísel je velmi snadná.



### 7.1.4 Násobení čísel Fixed Point

Budou-li násobena dvě desetinná čísla  $A$  a  $B$ , bude jejich výsledkem číslo  $C$  dle vztahu:

$$C = A * B \quad (7)$$

Pokud bude provedeno násobení dvou čísel Fixed Point, je možno s využitím vztahů (3) a (4) vyjádřit výsledek:

$$\begin{aligned} A' \cdot B' &= A \cdot Z \cdot B \cdot Z \\ &= A \cdot B \cdot Z \cdot Z \\ &= C \cdot Z \cdot Z \\ &= C' \cdot Z \end{aligned} \quad (8)$$

Ze vztahu (8) plyne, že výsledek násobení dvou Fixed Point čísel je potřeba upravit dle následující rovnice, aby výsledkem součinu bylo opět číslo ve formátu Fixed Point:

$$C' = \frac{(A' \cdot B')}{Z} \quad (9)$$

Závorka v čitateli výrazu (9) zdůrazňuje, že při výpočtu je potřeba nejprve vypočítat součin a pak teprve provést dělení. Jinak by došlo k výraznější ztrátě přesnosti, než k jaké dochází dle (9).

Jak by mohla vypadat implementace násobení dvou Fixed Point čísel ve formátu  $32:D$ , kde  $D$  je počet bitů za desetinnou tečku, je ukázáno v následujícím kódu. Nejprve prototyp funkce v jazyce C:

```
int mul_fixp( int a, int b, int dec );
```

Následuje ukázka implementace v JSI:

---

```
bits 32
section .data
section .text
global mul_fixp
mul_fixp:                                ; function mul_fixp
    enter 0,0
    mov eax, [ ebp + 8 ]                  ; a
    mov ecx, [ ebp + 16 ]                 ; dec
    imul dword [ ebp + 12 ]              ; (eax-edx) = a * b
    shrd eax, edx, cl                     ; (eax-edx) /= 2^dec
    leave
    ret
```

---

### 7.1.5 Dělení čísel Fixed Point

Bude-li provedeno dělení dvou desetinných čísel  $A$  a  $B$ , bude jejich výsledkem číslo  $C$  dle vztahu:

$$C = A/B \quad (10)$$

Pokud bude provedeno dělení dvou čísel Fixed Point, je možno s využitím vztahů (3) a (4) vyjádřit výsledek:

$$\begin{aligned} \frac{A'}{B'} &= \frac{A \cdot Z}{B \cdot Z} \\ &= \frac{A}{B} \cdot \frac{Z}{Z} \\ &= C \cdot Z \cdot \frac{1}{Z} \\ &= \frac{C'}{Z} \end{aligned} \quad (11)$$

Ze vztahu (11) plyne, že výsledek dělení dvou Fixed Point čísel je potřeba upravit dle následující rovnice, aby výsledkem dělení bylo opět číslo ve formátu Fixed Point:

$$C' = \frac{(A' \cdot Z)}{B'} \quad (12)$$

Závorka v čitateli výrazu (12) zdůrazňuje, že při výpočtu je potřeba nejprve vypočítat součin a pak teprve provést dělení. Jinak by došlo ke ztrátě přesnosti. Jak by mohla vypadat implementace dělení v JSI bude uvedeno v následující ukázce. Nejprve prototyp funkce v jazyce C:

```
int div_fixp( int a, int b, int dec );
```

Ukázka implementace v JSI:

---

```
bits 32
section .data
section .text
global div_fixp
div_fixp:                                ; function div_fixp
    enter 0,0
    mov eax, [ ebp + 8 ]                  ; a
    mov ecx, [ ebp + 16 ]                 ; dec
    cdq                                   ; sig. ext. to edx
    shld edx, eax, cl                      ; (eax-edx) *= 2^dec
    shl eax, cl
    idiv dword [ ebp + 12 ]               ; (eax-edx) /= b
    leave
    ret
```

---

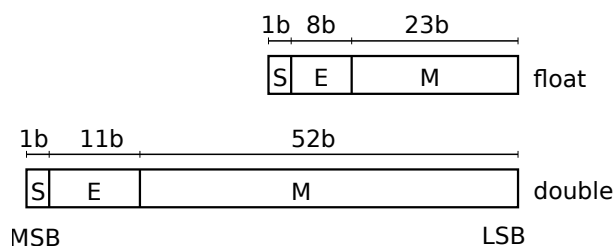
## 7.2 Float Point

Formáty čísel s plovoucí desetinnou tečkou jsou popsány v normě IEEE 754. Tato norma byla poprvé vydána v roce 1985 a aktualizována byla v roce 2008. Norma popisuje formáty čísel s plovoucí desetinnou tečkou, a to se základem 2 i se základem 10. Přehled formátů s dvojkovým základem je uveden v následující tabulce 1.

Přesnost	Délka	Mantisa	Exponent	Znaménko
poloviční	16b	10b	5b	1b
jednoduchá	32b	23b	8b	1b
dvojitá	64b	52b	11b	1b
čtyřnásobná	128b	112b	15b	1b

Tabulka 1: Formáty čísel s plovoucí desetinnou tečkou

V praxi se nejčastěji využívají formáty čísel jednoduché (**float**) a dvojitě (**double**) přesnosti a proto budou dále popsány podrobněji. Uspořádání bitů v číslech těchto formátů je znázorněno na obrázku 6.



Obrázek 6: Uspořádání bitů čísel float a double

Z informací uvedených v tabulce 1 a v obrázku 6 je patrné, že každé číslo s plovoucí desetinnou tečkou je vyjádřeno pomocí třech základních údajů: znaménkem, mantisou a exponentem.

Vyjádření hodnoty desetinného čísla daného uspořádanou trojicí hodnot  $\{S, M, E\}$  se provádí dle následujícího vzorce:

$$X = (-1)^S \cdot 1.M \cdot 2^{E-B}. \quad (13)$$

Z uvedeného vzorce je zřejmé, že hodnota znaménka 0 udává kladné číslo a hodnota 1 číslo záporné.

Zápis mantisy s číslem 1 na začátku znamená tzv. normalizovaný tvar. Mantisa je vždy upravena tak, aby nejvyšší platná jednička byla uvedena přímo před desetinnou tečkou. Tato jednička se však nezapisuje do mantisy

číslo, mantisu v čísle tvoří pouze její desetinná část. Tím je dosaženo přesnosti uloženého čísla o jeden bit větší, než je reálná velikost mantisy v čísle. Skutečný rozsah hodnot mantisy čísla je tak vždy v intervalu uzavřeném zleva:

$$1.M \in (1, 2) \quad (14)$$

Uvedený formát mantisy však nedovoluje definovat číslo 0. Pro tento a další speciální případy je formát uvedený v tabulce 2.

Exponent se ukládá ve formátu s posunutou nulou. Proto je ve vyjádření čísla uvedeno  $E - B$ . Hodnota  $B$  (bias) představuje posunutí hodnoty 0 na číselné ose. Pro čísla typu `float` a `double` je hodnota  $B$  následující:

$$B_{float} = 127$$

$$B_{double} = 1023$$

Některé kombinace samých 0 a 1 v mantise či exponentu definují speciální číselné hodnoty. Seznam těchto případů je v tabulce 2.

Znaménko	Mantisa	Exponent float	Exponent double	Význam
0	0	0	0	kladná nula
1	0	0	0	záporná nula
0	$\geq 0$	$0 < E < 255$	$0 < E < 2047$	kladné číslo
1	$\geq 0$	$0 < E < 255$	$0 < E < 2047$	záporné číslo
0	0	255	2047	kladné $\infty$
1	0	255	2047	záporné $\infty$
0	$> 0$	255	2047	NaN
1	$> 0$	255	2047	NaN

Tabulka 2: Formáty speciálních číselných hodnot

### 7.2.1 Výpočty s čísly Float Point

Aby bylo možno ukázat realizaci základních matematických operací s čísly Float Point, je potřeba nejprve čísla rozložit na znaménko, mantisu a exponent. Rozklad na jednotlivé části lze zajisté provést pomocí bitových operací. Snadnější cesta ale je v jazyce C/C++ použít datovou strukturu `union`.

```
union float_sep
{
    float f_num;
    struct
    {
        unsigned int m:23;
        unsigned int e:8;
        unsigned int s:1;
    };
};
```

Uvedená datová struktura má velikost 32 bitů a překrývají se v ní dvě položky: číslo jednoduché přesnosti `f_num` a bitové pole s položkami `m`, `e`, `s`. Přiřazením desetinného čísla do položky `f_num` se naplní obsah celé datové struktury a bitové pole umožní čtení jednotlivých částí čísla `f_num`.

```
s.f_num = 3.1415;
printf ( "%5.2f->s:0x%d m:%06x e:%d\n",
         s.f_num, s.s, s.m, s.e );
```

Pro čísla [ 1, 1.25, 1.5, 1.75, 2 ] bude rozklad vypadat následovně:

```
1.00 -> s:0 m:0x000000 e:127
1.25 -> s:0 m:0x200000 e:127
1.50 -> s:0 m:0x400000 e:127
1.75 -> s:0 m:0x600000 e:127
2.00 -> s:0 m:0x000000 e:128
```

S rozloženými čísly již lze realizovat základní matematické výpočty.

### 7.2.2 Násobení Float Point čísel

Pokud jsou dána dvě Float Point čísla  $X$  a  $Y$ , kdy je každé číslo zadáno svými třemi hodnotami  $\{S, M, E\}$ , je možno tato čísla vyjádřit dle (13):

$$\begin{aligned} X &= (-1)^{S_X} \cdot 1.M_X \cdot 2^{(E_X-B)} , \\ Y &= (-1)^{S_Y} \cdot 1.M_Y \cdot 2^{(E_Y-B)} . \end{aligned} \tag{15}$$

S čísly  $X$  a  $Y$  lze provést násobení a získaný výraz upravit:

$$\begin{aligned} Z &= X \cdot Y \\ &= (-1)^{S_X} \cdot 1.M_X \cdot 2^{(E_X-B)} \cdot (-1)^{S_Y} \cdot 1.M_Y \cdot 2^{(E_Y-B)} \\ &= (-1)^{(S_X+S_Y)} \cdot (1.M_X \cdot 1.M_Y) \cdot 2^{(E_X+E_Y-B-B)} \end{aligned} \quad (16)$$

$$= (-1)^{S_Z} \cdot 1.M_Z \cdot 2^{(E_Z-B)} \quad (17)$$

Výraz (16) lze rozložit na 3 samostatné výpočty, jak je v tomto výrazu naznačeno závorkami.

Výpočet výrazu  $(S_X + S_Y)$  je snadný. Pro sudý součet hodnot v exponentu bude výsledné číslo kladné, pro lichý výsledek exponentu bude výsledné číslo záporné. Hodnoty znaménka jsou jednobitová čísla a výsledné znaménko lze vyjádřit následujícím vztahem:

$$S_Z = (S_X + S_Y) \pmod{2} .$$

Exponent čísla 2 je jednoduchým součtem, který lze upravit dle pravidel pro počítání s čísly s posunutou nulou:

$$\begin{aligned} (E_X + E_Y - B - B) &= (E_Z - B) \\ E_Z &= E_X + E_Y - B \end{aligned}$$

Nyní je potřeba spočítat již jen poslední část výrazu:  $(1.M_X \cdot 1.M_Y)$ .

Z výrazu (14) je známo, že  $1.M_X, 1.M_Y \in \langle 1, 2 \rangle$ .

Výsledkem násobení čísel  $1.M_X \cdot 1.M_Y$  musí být číslo  $M \in \langle 1, 4 \rangle$ .

Pokud bude výsledkem číslo  $M \in \langle 1, 2 \rangle$ , je výsledek přímo v normalizovaném tvaru a bity za první jedničkou tvoří přímo mantisu  $M_Z$ .

Pro výsledné číslo  $M \in \langle 2, 4 \rangle$  je potřeba provést normalizaci. Ta se provede posunem o jeden bit doprava. Tento posun se musí promítnout do navýšení exponentu  $S_Z$ .

Samotná realizace výpočtu  $1.M_X \cdot 1.M_Y$  je snadná. Obě čísla  $M_X$  a  $M_Y$  jsou čísla celá velikosti 23 bitů. Před tato čísla stačí na začátek vložit jedničku, vynásobit je jako dvě celá čísla a upravit velikost. (Čísla  $1.M_X$  a  $1.M_Y$  jsou čísla s pevnou desetinnou tečkou a postup výpočtu byl podrobněji popsán v předchozí kapitole).

Pro realizaci kódu násobení v jazyce C bude ještě potřeba jedna pomocná datová struktura pro složení a rozložení mantisy:

```

union mantissa
{
    struct
    {
        unsigned int m:23;
        unsigned int m1:9;
    };
    int i_num;
};

```

Výsledný kód pro násobení dvou Float Point čísel lze implementovat dle výše popsaného postupu následovně:

---

```

#define B 127

int main()
{
    // float x, y, z
    float_sep x, y, z;
    // x and y initialization
    x.f_num = num1;
    y.f_num = num2;

    // z signum
    z.s = ( x.s + y.s ) % 2;
    // z exponent
    z.e = ( x.e + y.e - B );
    // mantissas extension with leading 1
    mantissa mx = { { .m = x.m, .m1 = 1 } };
    mantissa my = { { .m = y.m, .m1 = 1 } }, mz;
    // mantissa computing
    mz.i_num = ( ( long long ) mx.i_num ) * my.i_num >> 23;
    // adjust result in range <2,4)
    if ( mz.m1 >= 2 )
    {
        mz.i_num >>= 1; // normalize mantissa
        z.e++;        // adjust exponent
    }
    // z mantissa
    z.m = mz.m;
    // result
    printf( "%f*%f=%f\n", x.f_num, y.f_num, z.f_num );
}

```

---

### 7.2.3 Dělení Float Point čísel

Dělení dvou Float Point čísel lze realizovat stejným postupem, jako bylo v předchozí podkapitole realizováno násobení. Dělení dvou čísel lze roze-

psat do výrazu podobně jako násobení, provést úpravu výrazu a po částech vyhodnotit. Výsledný kód programu pro dělení dvou čísel pak bude velmi podobný kódu pro násobení. Lišit se bude provedení normalizace.

#### 7.2.4 Sčítání a odčítání Float Point čísel

Sčítání i odčítání dvou čísel z (15) je možno vyjádřit pomocí následujícího výrazu:

$$\begin{aligned} Z &= X \pm Y \\ &= (-1)^{S_X} \cdot 1.M_X \cdot 2^{(E_X-B)} \pm (-1)^{S_Y} \cdot 1.M_Y \cdot 2^{(E_Y-B)} \end{aligned} \quad (18)$$

Výraz (18) lze dále upravit pouze v případě, že budou oba exponenty čísla 2 stejné. Toho lze dosáhnout rozšířením jednoho ze dvou členů součtu výrazem:

$$2^Q/2^Q .$$

Hodnota  $Q$  musí být kladná a rozšíření se provede u toho členu, kde je exponent menší. Například kdyby platilo  $E_Y < E_X$ , určí se hodnota  $Q$  z následujícího výrazu:

$$E_X = E_Y + Q .$$

Pro známou hodnotu  $Q$  lze pokračovat v úpravě výrazu (18):

$$\begin{aligned} Z &= (-1)^{S_X} \cdot 1.M_X \cdot 2^{(E_X-B)} \pm (-1)^{S_Y} \cdot 1.M_Y \cdot 2^{(E_Y-B)} \cdot 2^Q/2^Q \\ &= (-1)^{S_X} \cdot 1.M_X \cdot 2^{(E_X-B)} \pm (-1)^{S_Y} \cdot 1.M_Y/2^Q \cdot 2^{(E_Y+Q-B)} \\ &= 2^{(E_X-B)} \cdot \{(-1)^{S_X} \cdot 1.M_X \pm (-1)^{S_Y} \cdot 1.M_Y/2^Q\} \end{aligned} \quad (19)$$

Exponent výsledku bude odpovídat exponentu čísla 2, který je ve výrazu (19) vytknutý před závorku:

$$E_Z = E_X .$$

Výraz  $1.M_Y/2^Q$  je bitovým posunem doprava:  $M' = 1.M_Y \gg Q$ .

Po provedení bitového posunu je potřeba vyhodnotit znaménka a následovat může sčítání nebo odčítání mantis:

$$M_Z = (\pm 1.M_X) \pm (\pm M') .$$

U výsledku se vyhodnotí a upraví znaménko  $S_Z$ . Výsledek se dle potřeby normalizuje a upraví se exponent výsledku  $E_Z$ .

Na základě příkladu z předchozí kapitoly lze uvedený postup snadno implementovat.



## 8 FPU

Jednotka FPU slouží pro výpočty s čísly s plovoucí desetinou tečkou. Tato jednotka je od verze procesoru i486 integrována na čipu CPU. Primárně se FPU jednotka používá v 32bitovém režimu, pro 64bitový režim je využívána hlavně jednotka SSE.

Téma FPU je velmi dobře zpracováno v samostatném dokumentu FPU-Tutorial.zip. Autorem textu i obrázků je Raymond Filiatreault - *rayfil@hotmail.com*.

### 8.1 Typové příklady

Všechny funkce uvedené dále v této kapitole jsou obsaženy v `soj-fpu.tgz`.

Jako první ukázka bude uvedena dvojice funkcí pro sčítání dvou argumentů. Jedna funkce bude sčítat dva argumenty typu `float` a druhá `double`. Prototypy funkcí v jazyce C vypadají následovně:

```
float add_float ( float a, float b );
double add_double ( double a, double b );
```

Kód funkcí lze v JSI implementovat následovně:

---

```
add_float :
    enter 0,0
    fld dword [ ebp + 8 ]      ; a
    fadd dword [ ebp + 12 ]   ; a += b
    leave
    ret

add_double :
    enter 0,0
    fld qword [ ebp + 8 ]     ; a
    fadd qword [ ebp + 16 ]   ; a += b
    leave
    ret
```

---

Jak je z ukázky kódu vidět, je kód obou funkcí prakticky totožný. Liší se pouze přístupem k parametrům funkce v zásobníku. Návrátová hodnota zůstává v registru `ST0`, všechny ostatní registry musí zůstat prázdné.

Následující příklad bude implementace výpočtu povrchu koule podle známého vzorce:

$$S = 4 \cdot \pi \cdot r^2.$$

Prototyp funkce v jazyce C bude následující:

```
double area_sphere ( double r );
```

Kód funkce lze v JSI implementovat následovně:

---

```
area_sphere :
    enter 0,0
    fld qword [ ebp + 8 ]      ; st0 = r
    fmul st0, st0             ; st0 = r*r
    fld1                      ; st0 = 1
    fadd st0, st0             ; st0 = 2
    fadd st0, st0             ; st0 = 4
    fmulp st1                 ; st1 = r*r*st0 and pop
    fldpi                     ; st0 = pi
    fmulp st1                 ; st1 = r*r*4*st0 and pop
    leave
    ret
```

---

Kód funkce se pomocí instrukcí implementuje vcelku snadno. FPU jednotka má k dispozici instrukce pro natažení předpřipravených konstant do registrů. Lze tak snadno z čísla 1 vytvořit potřebnou hodnotu 4 a získaný výsledek násobit  $\pi$ .

Dalším příkladem použití jednotku FPU bude přístup k číslům typu float v poli. Z těchto čísel se bude vybírat maximum.

Prototyp funkce v jazyce C:

```
float find_max( float *array, int N );
```

Kód funkce lze v JSI implementovat následovně:

---

```
find_max :
    enter 0,0
    mov edx, [ ebp + 8 ]      ; array
    mov ecx, [ ebp + 12 ]    ; N
    fld dword [ edx ]        ; first elem. as MAX
    dec ecx                   ; skip first element
.back :
    fld dword [ edx + ecx * 4 ] ; st0 = array[ ecx ]
    fcomi st1                 ; cmp st0, st1
    jb .skip
    fst st1                   ; exchange st1 = st0
.skip :
    fstp st0                  ; pop st0
    loop .back
                                ; MAX is in st0

    leave
    ret
```

---

Pro porovnávání čísel je výhodné používat instrukci FCOMI, která ukládá výsledek porovnání dvou čísel přímo do příznakových bitů registru FLAGS. Následně lze použít podmíněné skoky pro vyhodnocení bezznaménkových čísel.

V další ukázce bude ukázán přístup k prvkům pole typu `double`. Úkolem implementované funkce bude vypočítat průměrnou hodnotu všech prvků.

Prototyp funkce v jazyce C:

```
double array_average ( double *array, int N );
```

Kód funkce lze v JSI implementovat následovně:

---

```
array_average :
    enter 0,0
    mov edx, [ ebp + 8 ]      ; array
    mov ecx, [ ebp + 12 ]    ; N
    fldz                      ; st0 = 0
.back :
    fadd qword [ edx + ecx * 8 - 8 ] ; st0+=pole[ecx]
    loop .back
    fild dword [ ebp + 12 ]   ; N
    fdivp st1                 ; st0 /= N
    leave
    ret
```

---

Nejsložitějším typovým příkladem bude funkce pro výpočet obecné mocniny  $y = x^R$ . Tento výpočet se obvykle realizuje pomocí logaritmu, kdy se výraz mocniny přepisuje do následujícího tvaru:

$$y = x^R = 2^{R \cdot \log_2 x}$$

FPU jednotka umožňuje snadno vypočítat hodnotu exponentu čísla 2. K tomu má jedinou instrukci `FYL2X`. Pak ale nastává problém, protože v instrukční sadě FPU není obsažena instrukce pro výpočet  $2^R$ .

Jednotka FPU má k dispozici instrukce pro umocnění čísla 2 na exponent v intervalu  $\langle 0, 1 \rangle$  a instrukci pro umocnění čísla 2 na celočíselný exponent. Výpočet je proto nutno rozdělit na dva kroky:

$$2^R = 2^{\text{int}(R) + \text{des}(R)} = 2^{\text{int}(R)} \cdot 2^{\text{des}(R)},$$

kde funkce `int` znamená získání celočíselné části čísla  $R$  a funkce `des` získá pouze desetinnou část  $R$ .

Prototyp funkce v jazyce C:

```
double power_xy( double x, double exp );
```

Kód funkce lze v JSI implementovat následovně:

---

```
power_xy :
    enter 4,0
    fld qword [ ebp + 16 ]      ; st0 = exp
    fld qword [ ebp + 8 ]      ; st0 = x, st1 = exp
    fyl2x                       ; st1 *= log2(st0), pop
                                ; st0 is el2x

    fld st0
    fld st0                     ; 2 copies of el2x

    fstcw [ ebp - 4 ]          ; save CW
    fstcw [ ebp - 2 ]          ; save CW
    or word [ ebp - 4 ], 0xc00 ; rounding to 0
    fldcw [ ebp - 4 ]          ; restore CW
    frndint                    ; st0 = int( st0 )
    fldcw [ ebp - 2 ]          ; restore CW

    fsubp st1, st0             ; st1 -= st0, pop
                                ; st0 = frac. of el2x
    f2xm1                      ; st0 = 2^st0 - 1
    fld1
    faddp st1, st0             ; st1 += 1 and pop
                                ; st1 = el2x
    fscale                     ; st0 = 2^int(st1)*st0
    fstp st1                   ; st1 = st0 and pop

    leave
    ret
```

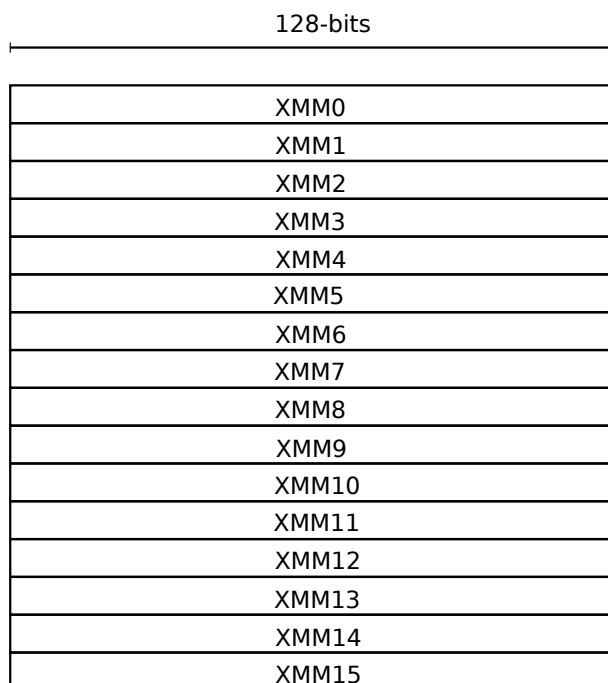
---

## 9 SSE

V 64bitovém režimu se pro výpočty s čísly s plovoucí desetinnou tečkou využívá převážně jednotka SSE. Lze stále používat i jednotku FPU, ale pro spojování JSI s dalšími jazyky je nutno využívat SSE.

### 9.1 Registry SSE

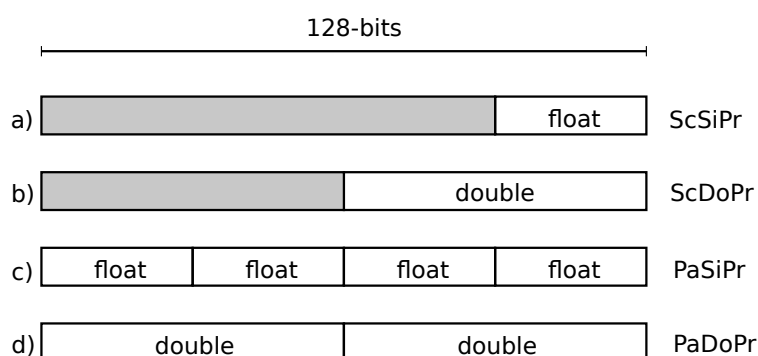
V první verzi SSE bylo k dispozici 8 registrů o velikosti 128 bitů. S rozšířením procesoru na 64 bitů byl navýšen počet registrů na 16. Registry jsou pojmenovány jako XMM0 až XMM15, jak je vidět na obrázku 7.



Obrázek 7: Registry SSE

## 9.2 Obsah registrů

Do registrů `XMMx` je možno ukládat desetinná čísla několika způsoby. Přehledně jsou možnosti zobrazeny na obrázku 8.



Obrázek 8: Uložení čísel v registrech

Čísla jsou v registrech uložena buď jednotlivě (scalar), nebo jako dvojice či čtveřice čísel (packed). Pro výpočty se používají čísla typu `float` nebo `double`. Celkem jsou tedy 4 možnosti, jak čísla do registrů ukládat:

- formát Scalar Single-Precision (`ScSiPr`) je na obrázku 8 a),
- formát Scalar Double-Precision (`ScDoPr`) je na obrázku 8 b),
- formát Packed Single-Precision (`PaSiPr`) je na obrázku 8 c),
- formát Packed Double-Precision (`PaDoPr`) je na obrázku 8 d).

Pro uvedené formáty byly pro účely tohoto textu zavedeny i zkratky, které budou dále použity v popisu instrukcí.

## 9.3 Instrukce SSE

Instrukční sada SSE se během vývoje procesorů rozrostla na několik stovek. Z tohoto množství instrukcí je potřeba pro počáteční seznámení se s používáním SSE jednotky jen ty, které dostačují pro realizaci základních výpočtů. Instrukce zde budou opět řazeny tématicky, nikoliv abecedně.

### 9.3.1 Instrukce přesunové

#### MOVAPD cíl, zdroj

Instrukce provede přesun dvojice čísel `double` ve formátu PaDoPr z operandu `zdroj` do operandu `cíl`.

Zdroj i cíl může být registr `XMMx` i paměť, nelze však použít dva paměťové operandy. Instrukce předpokládá zarovnání paměťového operandu na adresu, která je celistvým násobkem čísla 16.

#### MOVUPD cíl, zdroj

Instrukce je stejná jako MOVAPD, ale u paměťových operandů se nepředpokládá zarovnání adresy na násobek 16.

#### MOVAPS cíl, zdroj

Instrukce provede přesun čtveřice čísel `float` ve formátu PaSiPr z operandu `zdroj` do operandu `cíl`.

Zdroj i cíl může být registr `XMMx` i paměť, nelze však použít dva paměťové operandy. Instrukce předpokládá zarovnání paměťového operandu na adresu, která je celistvým násobkem čísla 16.

#### MOVUPS cíl, zdroj

Instrukce je stejná jako MOVAPS, ale u paměťových operandů se nepředpokládá zarovnání adresy na násobek 16.

#### MOVDQA cíl, zdroj

Instrukce provede přesun 16 bajtů z operandu `zdroj` do operandu `cíl`.

Instrukce předpokládá zarovnání paměťového operandu na adresu, která je celistvým násobkem čísla 16.

#### MOVDQU cíl, zdroj

Instrukce je stejná jako MOVDQA, ale u paměťových operandů se nepředpokládá zarovnání adresy na násobek 16.

#### MOVSD cíl, zdroj

Instrukce provede přesun jednoho čísla `double` ve formátu ScDoPr z operandu `zdroj` do operandu `cíl`.

Zdroj i cíl může být registr `XMMx` i paměť, nelze však použít dva paměťové operandy.

#### MOVSS cíl, zdroj

Instrukce provede přesun jednoho čísla `float` ve formátu ScSiPr z operandu `zdroj` do operandu `cíl`.

Zdroj i cíl může být registr `XMMx` i paměť, nelze však použít dva paměťové operandy.



### 9.3.2 Instrukce přesunové se změnou pořadí čísel

MOVDDUP cíl, zdroj

Instrukce provede převod jednoho čísla `double` na formát PaDoPr pomocí duplikace. Pro lepší představu je možno chování instrukce popsat následujícím kódem:

```
cíl[ 0 ] = cíl[ 1 ] = zdroj[ 0 ];
```

MOVHPD/MOVLPD cíl, zdroj

Instrukce provede přesun jednoho čísla `double` ze zdroje do cíle. Instrukce `MOVHPD` přesune horní číslo a instrukce `MOVLPD` přesune dolní číslo.

SHUFPD cíl, zdroj, k8

Instrukce provede přesun jednoho čísla `double` ze zdroje a jednoho čísla z cíle do cíle. Konstanta `k8` určuje svými dvěma dolními bity, které číslo bude přesunuto ze zdroje a které z cíle. Pro lepší představu je možno chování instrukce popsat následujícím kódem:

```
k8 = 0b000000i1i0;  
cíl[ 0 ] = cíl[ i0 ]  
cíl[ 1 ] = zdroj[ i1 ];
```

MOVSHDUP/MOVSLDUP cíl, zdroj

Instrukce provede přesun dvou `float` čísel ze zdroje do cíle s duplikací. Instrukce `MOVSHDUP` vybírá ze zdroje liché prvky:

```
cíl[ 0 ] = cíl[ 1 ] = zdroj[ 1 ];  
cíl[ 2 ] = cíl[ 3 ] = zdroj[ 3 ];
```

Instrukce `MOVSLDUP` vybírá ze zdroje sudé prvky:

```
cíl[ 0 ] = cíl[ 1 ] = zdroj[ 0 ];  
cíl[ 2 ] = cíl[ 3 ] = zdroj[ 2 ];
```

MOVHPS/MOVLPS cíl, zdroj

Instrukce provede přesun dvou `float` čísel ze zdroje do cíle. Instrukce `MOVHPS` přesune horní dvě čísla a instrukce `MOVLPS` přesune dolní dvě čísla.

MOVHLPS/MOVHLPS cíl, zdroj

Instrukce provede přesun dvou `float` čísel ze zdroje do cíle. Instrukce `MOVHLPS` přesune horní dvě čísla do dolní části cíle a instrukce `MOVLHPS` přesune dolní dvě čísla do horní části cíle.

SHUFPS cíl, zdroj, k8

Instrukce provede přesun dvou čísel `float` ze zdroje a dvou čísel z cíle do cíle. Konstanta `k8` určuje svými bity (čtyřmi dvojicemi), která čísla budou přesunuta ze zdroje a které z cíle. Pro lepší představu je možno chování instrukce popsat následujícím kódem:

```
k8 = 0bi76i54i32i10;  
cíl[ 0 ] = cíl[ i10 ];  
cíl[ 1 ] = cíl[ i32 ];  
cíl[ 2 ] = zdroj[ i54 ];  
cíl[ 3 ] = zdroj[ i76 ];
```

### 9.3.3 Převod formátů čísel

CVTPD2PS cíl, zdroj

Instrukce provede přesun a změnu formátu dvojice čísel `double` z formátu `PaDoPr` z operandu `zdroj` do operandu `cíl` tak, že cílový operand bude obsahovat dvojici čísel `float` ve formátu `PaSiPr`. Horní dvě čísla budou 0.

Zdroj i cíl může být registr `XMMx` i paměť, nelze však použít dva paměťové operandy.

CVTPS2PD cíl, zdroj

Instrukce provede přesun a změnu formátu dolní dvojice čísel `float` z formátu `PaSiPr` z operandu `zdroj` do operandu `cíl` tak, že cílový operand bude obsahovat dvojici čísel `double` ve formátu `PaDoPr`.

Zdroj i cíl může být registr `XMMx` i paměť, nelze však použít dva paměťové operandy.

CVTSD2SS cíl, zdroj

Instrukce provede přesun a změnu formátu čísla `double` z formátu `ScDoPr` z operandu `zdroj` do operandu `cíl` tak, že cílový operand bude obsahovat číslo `float` ve formátu `ScSiPr`.

Zdroj i cíl může být registr `XMMx` i paměť, nelze však použít dva paměťové operandy.

CVTSS2SD cíl, zdroj

Instrukce provede přesun a změnu formátu čísla `float` z formátu `ScSiPr` z operandu `zdroj` do operandu `cíl` tak, že cílový operand bude obsahovat číslo `double` ve formátu `ScDoPr`.

Zdroj i cíl může být registr `XMMx` i paměť, nelze však použít dva paměťové operandy.

CVTSD2SI/CVTSS2SI cíl, zdroj

Instrukce provede přesun a změnu formátu čísla `double` nebo `float` z formátu `ScDoPr` nebo `ScSiPr` z operandu `zdroj` do operandu `cíl` tak, že cílový operand bude obsahovat celé 32 nebo 64bitové číslo.

Zdroj může být registr `XMMx` i paměť, cílový operand musí být 32 nebo 64bitový registr `ALU`.

CVTSI2SD/CVTSI2SS cíl, zdroj

Instrukce provede přesun celého 32 nebo 64bitového čísla z operandu `zdroj` do operandu `cíl` tak, že cílový operand bude obsahovat číslo `double` ve formátu `ScDoPr` nebo číslo `float` ve formátu `ScSiPr`.

Zdrojový operand musí být 32 nebo 64bitový registr `ALU`, cílovým operandem musí být registr `XMMx`

### 9.3.4 Aritmetické operace

Instrukce pro základní aritmetické operace sčítání, odčítání, násobení a dělení, se používají stejným způsobem. Liší se jen použitým formátem operandů.

ADDPD/DIVPD/MULPD/SUBPD cíl, zdroj

Instrukce provede požadovanou matematickou operaci mezi operandem `cíl` a `zdroj` a výsledek uloží do cílového operandu. Oba operandy musí být ve formátu `PaDoPr`.

Cílový operand musí být registr `XMMx`, zdrojový operand může být registr nebo paměť zarovnaná na 16 bajtů.

ADDPS/DIVPS/MULPS/SUBPS cíl, zdroj

Instrukce provede požadovanou matematickou operaci mezi operandem `cíl` a `zdroj` a výsledek uloží do cílového operandu. Oba operandy musí být ve formátu `PaSiPr`.

Cílový operand musí být registr `XMMx`, zdrojový operand může být registr nebo paměť zarovnaná na 16 bajtů.

ADDSD/DIVSD/MULSD/SUBSD cíl, zdroj

Instrukce provede požadovanou matematickou operaci mezi operandem `cíl` a `zdroj` a výsledek uloží do cílového operandu. Oba operandy musí být ve formát `ScDoPr`.

Cílový operand musí být registr `XMMx`, zdrojový operand může být registr nebo paměť.

**ADDSS/DIVSS/MULSS/SUBSS** cíl, zdroj

Instrukce provede požadovanou matematickou operaci mezi operandem cíl a zdroj a výsledek uloží do cílového operandu. Oba operandy musí být ve formátu ScSiPr.

Cílový operand musí být registr XMMx, zdrojový operand může být registr nebo paměť.

**SQRTPS/SQRTSS/SQRTPD/SQRTSD** cíl, zdroj

Instrukce provede výpočet druhé odmocniny. Operand může být ve formátu PaSiPr, ScSiPr, PaDoPr i ScDoPr.

**RSQRTPS/RSQRTSS** cíl, zdroj

Instrukce provede výpočet převrácené hodnoty odmocniny čísel/čísla. Operand může být ve formátu PaSiPr i ScSiPr.

**RCPPS/RCPSS** cíl, zdroj

Instrukce provede výpočet převrácené hodnoty čísel/čísla. Operand může být ve formátu PaSiPr i ScSiPr.

### 9.3.5 Bitové operace

Mezi registry **XMMx** lze provádět i bitové operace. U těchto operací nezáleží na vnitřním formátu registru, protože operace probíhají po bitech. Proto je rozlišení formátů PaDoPr a PaSiPr jen formální.

U všech operací je nutno dbát na zarovnání paměťového operandu na adresu 16 bajtů.

#### ANDPD/ANDPS/ANDNPD/ANDNPS cíl, zdroj

Instrukce provede bitovou operaci AND mezi operandem cíl a zdroj. Výsledek se uloží do cílového operandu.

Instrukce ANDNxx provede před provedením operace AND negaci operandu zdroj.

Cílový operand musí být registr **XMMx**, zdrojový operand může být registr nebo paměť zarovnaná na 16 bajtů.

#### ORPD/ORPS cíl, zdroj

Instrukce provede bitovou operaci OR mezi operandem cíl a zdroj. Výsledek se uloží do cílového operandu.

Cílový operand musí být registr **XMMx**, zdrojový operand může být registr nebo paměť zarovnaná na 16 bajtů.

#### XORPD/XORPS cíl, zdroj

Instrukce provede bitovou operaci XOR mezi operandem cíl a zdroj. Výsledek se uloží do cílového operandu.

Cílový operand musí být registr **XMMx**, zdrojový operand může být registr nebo paměť zarovnaná na 16 bajtů.

### 9.3.6 Porovnávání čísel

CMPccPD/CMPccPS/CMPccSD/CMPccSS cíl, zdroj, podmínka

Instrukce provádí porovnání dvou parametrů stejného formátu. Formát obou parametrů může být PaDoPr, PaSiPr, ScDoPr a ScSiPr. Porovnání se provádí mezi operandem cíl a zdroj a třetí parametr podmínka udává, jaká podmínka se bude vyhodnocovat. Výsledek operace se uloží do cílového operandu, kde na příslušném místě dle typu operandu budou samé 0, když podmínka splněna není, nebo samé 1 v případě splnění podmínky.

Cílový operand musí být registr XMMx, zdrojový operand může být registr nebo paměť.

Jako výběr vyhodnocované podmínky v třetím operandu instrukce může být některé z následujících čísel:

- 0 EQ - Equal,
- 1 LT - Less-than,
- 2 LE - Less-than or equal,
- 3 UNORD - Unordered (alespoň jeden z operandů je NAN),
- 4 NE - Not-equal,
- 5 NLT - Not-less-than,
- 6 NLE - Not-less-than,
- 7 ORD - Ordered (ani jeden z operandů není NAN).

COMISD/COMISS cíl, zdroj

Instrukce COMISD a COMISS porovná dva skalární parametry float nebo double ve formátu ScDoPr nebo ScSiPr. Operand cíl musí být XMMx registr, zdroj může být registr nebo paměť. Výsledek porovnání se ukládá do FLAGS registru ALU a lze pro vyhodnocení podmínky použít instrukce podmíněných skoků. Instrukce nastavuje dle provedeného porovnání pouze příznakové bity ZF, PF a CF. Bity AF, OF a SF se nulují.

ZF,PF,CF = 111: Unordered.

ZF,PF,CF = 000: Greater-than.

ZF,PF,CF = 001: Less-than.

ZF,PF,CF = 100: Equal.

MINPS/MINSS/MINPD/MINSD cíl, zdroj

MAXPS/MAXSS/MAXPD/MAXSD cíl, zdroj

Instrukce porovná číslo/čísla ve zdroji a cíli a do cíle uloží nalezená minima nebo maxima.

## 9.4 Typové příklady použití SSE

Všechny funkce uvedené dále v této kapitole jsou obsaženy v `soj-sse.tgz`.

Jako první ukázka může sloužit příklad funkce pro sečtení dvou čísel `float` a `double`. Prototypy funkcí v jazyce C mohou vypadat následujícím způsobem:

```
float add_float ( float a, float b );
double add_double ( double a, double b );
```

Následující implementace v JSI je velmi jednoduchá:

---

```
add_float :
    addss xmm0, xmm1          ; a += b
    ret

add_double :
    addsd xmm0, xmm1          ; a += b
    ret
```

---

Jak je na ukázce kódu vidět, parametry funkce s desetinnou tečkou jsou předávány přímo přes registry `XMMx` a proto je možné předané hodnoty přímo použít. Návrátová hodnota funkce je vrácena také přes registr `XMM0` a proto je možné kód obou funkcí implementovat pomocí jediné instrukce. Je pouze potřeba správně zvolit odpovídající instrukci v závislosti na typu parametrů.

Další ukázkou může být funkce pro výpočet objemu koule. Výpočet je dán známým vzorcem:

$$V = 4/3 \cdot \pi \cdot r^3.$$

Implementace funkce pro výpočet objemu koule je pomocí instrukcí SSE snadná. Prototyp funkce v jazyce C:

```
double volume_sphere ( double R );
```

Implementace v JSI je následující:

---

```
volume_sphere :
    movsd xmm1, xmm0          ; r
    mulsd xmm0, xmm0          ; r*r
    mulsd xmm0, xmm1          ; r*r*r
    mulsd xmm0, [ pi ]        ; *pi
    mov eax, 4
    cvtsi2sd xmm1, eax
    mulsd xmm0, xmm1          ; *4
    dec eax
    cvtsi2sd xmm1, eax
    divsd xmm0, xmm1          ; /3
    ret
```

---

V další ukázce bude uveden příklad přístupu k položkám pole typu float. Z toho pole bude funkce vybírat maximální prvek.

Prototyp funkce v jazyce C může vypadat následovně:

```
float find_max( float *array, int N );
```

Implementace v JSI bude následující:

---

```
find_max :
    movss xmm0, [ rdi ]           ; sel. first element as MAX
    movsx rcx, esi                ; N
    dec rcx                       ; skip first element
.back :
    comiss xmm0, [ rdi + rcx * 4 ] ; compare
    jae .skip
    movss xmm0, [ rdi + rcx * 4 ] ; exchange MAX
.skip :
    loop .back
                                   ; result is in XMM0
    ret
```

---



V následující ukázce kódu bude uveden příklad použití formátu čísel PaDoPr. Úkolem následující funkce bude výpočet průměrné hodnoty prvků pole typu `double`. Pro sčítání prvků pole je možno využít formát PaDoPr a sčítat prvky pole po dvojicích. Cyklus pro sčítání se tak zkrátí na polovinu. Na konci cyklu pak jen stačí sečíst oba mezisoučty.

Prototyp funkce v jazyce C je následující:

```
double array_average ( double *array, int N );
```

Implementace v JSI bude následující:

---

```
array_average :
    xorpd xmm0, xmm0           ; sum = 0
    xor rdx, rdx               ; inx = 0
    movsx rcx, esi            ; N
    shr rcx, 1                 ; N /= 2
    jnc .nocf                  ; is N odd?
    movsd xmm0, [ rdi ]       ; store odd element
    inc rdx                     ; skip odd element
.nocf :
    xorpd xmm1, xmm1          ; sum2 = 0,0
.back :
    movupd xmm2, [ rdi + rdx * 8 ]
    addpd xmm1, xmm2           ; sum2 += pair of numbers
    add rdx, 2                  ; skip two numbers
    loop .back                 ; while ( --rcx )

    addsd xmm0, xmm1           ; sum += sum2
    shufpd xmm1, xmm1, 1       ; exchange numbers in sum2
    addsd xmm0, xmm1           ; sum += sum2
    cvtsi2sd xmm1, esi
    divsd xmm0, xmm1           ; sum /= N
    ret
```

---

## 10 Počítání s velkými čísly

V praxi se často vyskytuje potřeba počítat s čísly většími, než je velikost registrů procesoru. Pro tyto výpočty je instrukční sada procesorů obvykle přizpůsobena, je však potřeba znát základní principy výpočtů, aby byly instrukce použity správně a výsledný kód dával správné výsledky. V následujícím textu budou čísla rozdělena do 3 skupin.

1. Čísla velikosti registru.
2. Čísla velikosti dvou registrů.
3. Čísla obecné velikosti  $M$  bitů.

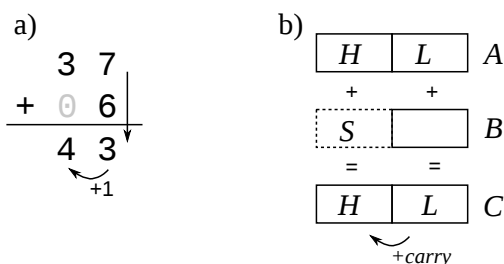
Aby bylo možno následující programy zkusit v 32 i 64bitovém režimu, bude jako základní velikost registru vybrána délka 32 bitů. Čísla se budou dělit na 3 skupiny uvedené výše a budou pojmenována následovně:

1. čísla `int32`,
2. čísla `int64`,
3. čísla `intN`.

### 10.1 Výpočty s čísly `int32` a `int64`

#### 10.1.1 Sčítání a odčítání čísla `int64` a `int32`

Pro vysvětlení sčítání dvou čísel různé velikosti je možno použít příklad sčítání dvouciferného čísla s číslem jednociferným. Příklad je na obrázku 9 a).



Obrázek 9: Sčítání čísel `int64` a `int32`

Uvedený princip je možno zobecnit. Pokud se jednotlivé číslice z příkladu 9 a) nahradí obecným blokem, vznikne situace na obrázku 9 b). V této zobecněné variantě nezáleží na velikosti bloku, zda je 8, 16 nebo 32 bitů velký, princip sčítání bude vždy stejný.

Důležitým krokem před samotným sčítáním je doplnění prázdného místa označeného *S*. V obrázku 9 a) je prázdné místo doplněno číslicí 0. Při sčítání čísel v počítači ve formátu dvojkového doplňku je však potřeba doplnit chybějící levostranné bity znaménkovým rozšířením čísla *B*.

Čísla *A*, *B* a *C* uvedená na obrázku 9 b) jsou rozdělena na horní část označenou *H* a dolní část označenou *L*. Sčítání začíná součtem spodních řádů čísel a případný přenos se přenáší do vyššího řádu.

Pro popsany postup je možno implementovat funkci v JSI. Nejprve prototyp funkce v jazyce C:

```
long long add_int64int32 ( long long a, int b );
```

Implementace v JSI bude následující:

---

```
add_int64int32 :
    enter 0,0
    mov eax, [ ebp + 16 ]      ; b
    cdq                      ; sign-ext. eax->edx
    mov ecx, edx              ; s = edx
    mov eax, [ ebp + 8 ]      ; eax = a_l
    mov edx, [ ebp + 12 ]     ; edx = a_h
    add eax, [ ebp + 16 ]     ; eax += b
    adc edx, ecx              ; edx += s + cf
    leave
    ret                       ; return eax-edx
```

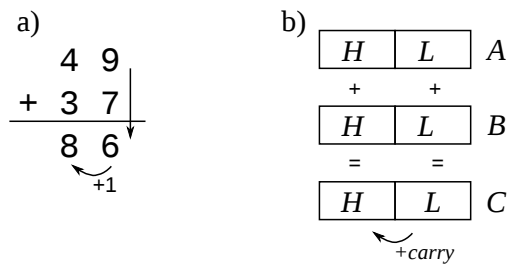
---

Princip odčítání jednociferného čísla od čísla dvouciferného je prakticky shodný s výše popsaným principem pro sčítání. Lze proto uvedený kód po drobné úpravě použít i pro realizaci odčítání čísla `int32` od čísla `int64`. Stačí nahradit instrukci `ADD` instrukcí `SUB` a instrukci `ADC` instrukcí `SBB`.

### 10.1.2 Sčítání a odčítání čísel int64

Sčítání a odčítání dvou čísel stejné velikosti je v zásadě jednodušší, než bylo v předchozím případě sčítání dvou čísel různých velikostí. V tomto případě není potřeba provádět znaménkové rozšíření a je možno provést přímo sčítání. Postup je opět zobrazen na obrázku 10 a).

Na obrázku 10 b) je vyobrazeno zobecnění postupu sčítání, bez ohledu na velikost sčítaných bloků.



Obrázek 10: Sčítání čísel int64 a int64

Pro znázorněný postup sčítání lze implementovat v JSI odpovídající funkci. Nejprve prototyp funkce v jazyce C:

```
long long add_int64int64 ( long long a, long long b );
```

Implementace v JSI bude následující:

---

```
add_int64int64 :
    enter 0,0
    mov eax, [ ebp + 8 ]           ; eax = a_l
    mov edx, [ ebp + 12 ]        ; edx = a_h
    add eax, [ ebp + 16 ]        ; eax += b_l
    adc edx, [ ebp + 20 ]        ; edx += b_h + cf
    leave
    ret                           ; return eax-edx
```

---

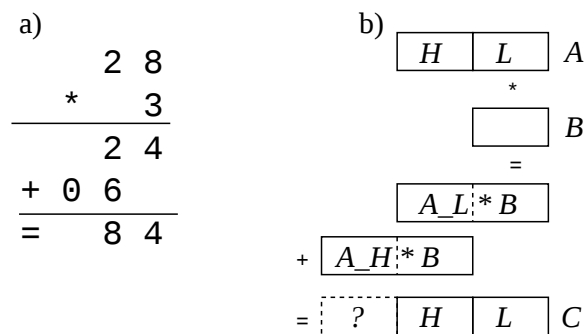
Uvedený kód je možno snadno upravit na funkci pro odčítání dvou čísel int64. Stačí nahradit instrukci ADD instrukcí SUB a instrukci ADC instrukcí SBB.

### 10.1.3 Násobení čísla `int64` číslem `int32`

Násobení čísla `int64` číslem `int32` si lze opět snadno připodobnit násobení dvouciferného čísla číslem jednociferným. Pokud bude číslo  $A$  číslem dvouciferným, které se dá rozložit na část  $H$  a  $L$ , a číslo  $B$  bude číslo jednociferné, je možné provést násobení pomocí roznásobení:

$$C = A \cdot B = (A\_H + A\_L) \cdot B = A\_H \cdot B + A\_L * B$$

Uvedený postup je znázorněn na obrázku 11 a).



Obrázek 11: Násobení čísel `int64` an `int32`

Popsaný postup násobení je zobecněn pomocí bloků obecné velikosti na obrázku 11 b). Z uvedeného postupu je zřejmé, že implementace kódu pro násobení bude provádět postupně dvě násobení a jednotlivé mezivýsledky bude potřeba sečíst. Pokud se po sčítání objeví v místě označeném "?" nenulová hodnota, došlo při násobení k přetečení.

Dle popsaného postupu lze implementovat kód funkce pro násobení v JSI následujícím způsobem. Nejprve prototyp funkce v jazyce C:

```
long long mul_int64int32 ( long long a, int b );
```

Implementace v JSI bude následující:

---

```
mul_int64int32 :
    enter 0,0
    mov eax, [ ebp + 8 ]           ; a_l
    mul dword [ ebp + 16 ]        ; a_l * b
    mov esi, eax                  ; c_l
    mov edi, edx                  ; c_h
    mov eax, [ ebp + 12 ]        ; a_h
    mul dword [ ebp + 16 ]        ; a_h * b
    add edi, eax                  ; c_h += eax
    ;adc edx, 0                   ; overflow?
    mov eax, esi                  ; c_l
    mov edx, edi                  ; c_h
    leave
    ret                           ; return eax-edx
```

---

Uvedený kód umožňuje násobit **pouze čísla kladná**. Před a po volání uvedené funkce je potřeba upravit znaménka parametrů a výsledku, případně rozšířit kód o úpravu znamének.

#### 10.1.4 Násobení čísel int64

Podobně jako v předchozí podkapitole, tak i v tomto případě je možno provést násobení dvou čísel  $A$  a  $B$ , která se dají rozložit na část  $H$  a  $L$  dle vzorce:

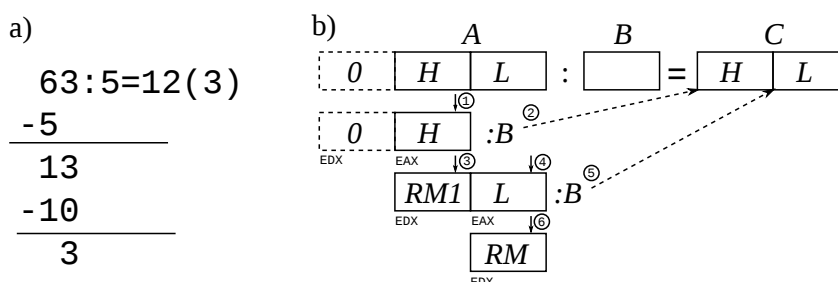
$$\begin{aligned} C &= A \cdot B = (A_H + A_L) \cdot (B_H + B_L) \\ &= A_H * B_H + A_H * B_L + A_L * B_H + A_L * B_L \end{aligned}$$

Pro výpočet výsledku bude zapotřebí provést postupně čtyři násobení a postupné sčítání mezivýsledků. Při násobení dvou čísel `int64` může být vypočítaný výsledek až 128bitový. Je proto potřeba kontrolovat vstupní parametry, aby na vstupu nebyly hodnoty čísel takové, ze kterých by bylo možné přetečení detekovat během výpočtu a vhodným způsobem předávat informaci o přetečení zpět do volající funkce. Je také možné vrátit jako výsledek 128bitové číslo dle vlastní specifikace.

### 10.1.5 Dělení čísla int64 číslem int32

Dělení čísla `int64` číslem `int32` není možné provést přímo jednou instrukcí (I)DIV, i když tato instrukce má na vstupu 64bitové číslo. Výsledek instrukce se totiž musí vejít do 32bitového registru. V případě příliš velkého dělence a malého dělitele však může být výsledek větší než 32 bitů (např.  $2^{50}/4 > 2^{32}$ ).

Pro realizaci dělení je možno se opět inspirovat v klasickém postupu dělení dvouciferného čísla číslem jednociferným na papíře. Postup je znázorněn na obrázku 12 a).



Obrázek 12: Dělení čísel `int64` an `int32`

Tento postup lze stejně jako v předchozích podkapitolách zobecnit na bloky obecné velikosti, jak je ukázáno na obrázku 12 b). Jednotlivé kroky jsou v obrázku pro lepší pochopení postupu číslovány.

Z obrázku je zřejmé, že dělení se musí rozdělit na dvě instrukce dělení. Instrukce DIV, jejíž implementace se mohla dosud stále zdát zbytečně komplikovaná, se zde projeví jako velmi výkonný pomocník. Zbytek po dělení se ukládá do registru EDX a tento registr pokračuje vždy do dalšího kroku v horním řádu dělence. V obrázku 12 b) je to patrné v kroku číslo 3. Do prvního dělení vstupuje registr EDX s hodnotou 0.

Uvedený postup lze implementovat velmi snadno v JSI. Prototyp funkce v jazyce C bude vypadat následovně:

```
long long div_int64int32 ( long long a, int b );
```

Implementace v JSI bude následující:

---

```
div_int64int32 :
    enter 0,0
    mov edx, 0                ; left 0
    mov eax, [ ebp + 12 ]    ; a_h
    div dword [ ebp + 16 ]   ; eax-edx /= b
    mov ecx, eax             ; save c_h
    mov eax, [ ebp + 8 ]    ; a_l
    div dword [ ebp + 16 ]   ; eax-edx /= b
    ;edx remainder          ; remainder?
    mov edx, ecx             ; restore c_h
    leave
    ret                       ; return eax-edx
```

---

Uvedený kód umožňuje dělit **pouze čísla kladná**. Před a po volání uvedené funkce je potřeba upravit znaménka parametrů a výsledku, případně rozšířit kód o úpravu znamének.





### 10.2.3 Převod čísla `intN` na řetězec a zpět

Při práci s jakýmikoliv čísly je potřeba mít k dispozici dvě základní funkce: zobrazení binárně uloženého čísla v textové formě a funkci opačnou pro převod textového vstupu do binární podoby pro výpočty.

V jazyce C je možno naznačit převodní funkce čísla `int` na řetězec a zpět následujícím způsobem:

---

```
#define B 10

char *int_to_str ( int X, char *str )
{
    while ( X )
    {
        *str++ = X % B + '0';
        X /= B;
    }
    *str = '\0';
    return str;
}

int str_to_int ( char *str )
{
    int X = 0;
    while ( *str )
    {
        X *= B;
        X += *str++ - '0';
    }
    return X;
}
```

---

Pokud by se v uvedených funkcích podařilo nahradit proměnnou `X` číslem typu `intN`, budou obě funkce fungovat i pro převody velkých čísel. Nestačí však pouze změnit typ proměnné `X`. Pro typ `intN` nelze použít matematické operace ve stejné podobě, jak jsou uvedeny v kódu. Bude proto potřeba nahradit tyto základní operace funkcemi. V převodních funkcích se vyskytuje pouze čtveřice operací.

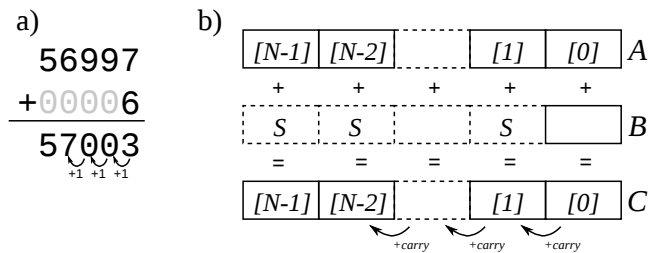
Ve funkci `int_to_str` je potřeba vypočítat zbytek po dělení základem číselné soustavy, který je v kódu definován jako `B`. Druhou operací je dělení čísla `X` základem soustavy. Ve strojovém kódu se řeší obě operace současně v jedné instrukci.

Ve funkci `str_to_int` je potřeba také dvojice operací. Číslo `X` je potřeba vynásobit základem soustavy a následně číslo `X` zvýšit o další číslici z řetězce.

Implementace 4 funkcí bude stačit k tomu, aby bylo možno převádět čísla `intN` na řetězec a zpět.

### 10.2.4 Sčítání čísla intN a int32

Princip sčítání čísla `int32` a čísla většího byl již vysvětlen v kapitole 10.1.1. Sčítání s číslem velikosti `intN` je v principu stejné, jen proběhne ve více řádech čísla. Princip je možno i zde připodobnit sčítání vícemístného čísla s číslem jednomístným, jako na obrázku 14 a).



Obrázek 14: Sčítání čísla `intN` a `int32`

Zobecnění postupu pomocí bloků obecné velikosti je na obrázku 14 b). Stejně jako v případě 9 b) je i zde potřeba provést správně znaménkové rozšíření označené v obrázku 14 b) jako *S*.

V souladu s vyobrazeným a popsaným principem lze funkci sčítání čísla `int32` a `intN` implementovat v JSI. Prototyp funkce v jazyce C bude mít následující podobu:

```
// a += b
void add_intNint32 ( int *a, int b, int N );
```

Implementace v JSI bude následující:

---

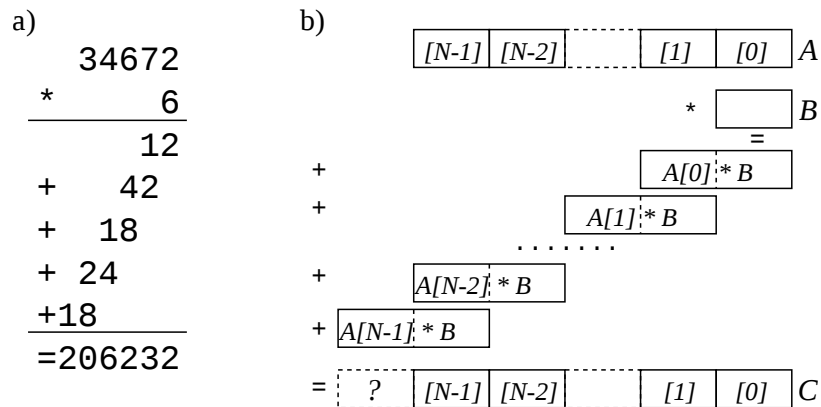
```
add_intNint32 :
    enter 0,0
    push ebx                ; save registers
    push esi
    mov ebx, [ ebp + 8 ]    ; a
    mov eax, [ ebp + 12 ]   ; b
    cdq                    ; sign. ext. eax->edx
    mov ecx, [ ebp + 16 ]   ; N
    dec ecx                ; skip a[ 0 ]
    mov esi, 0              ; inx = 0
    add [ ebx ], eax        ; a[ 0 ] += b
.back
    inc esi                ; inx++
    adc [ ebx + esi * 4 ], edx ; a[inx] += edx + cf
    loop .back
    pop esi                ; restore registers
    pop ebx
    leave
    ret
```

---

Navrženou funkci lze použít i pro odčítání čísla `int32` od čísla `intN`. Stačí před voláním funkce otočit znaménko parametru `b`.

### 10.2.5 Násobení čísla `intN` a `int32`

Násobení velkého čísla `intN` číslem `int32` lze realizovat postupným násobením jednotlivých jeho částí a mezivýsledky sčítat. V jednodušší podobě byl postup již použit v kapitole 10.1.3 a jeho rozšířená varianta je znázorněna na obrázku 15.



Obrázek 15: Sčítání čísla `intN` a `int32`

Obrázek znázorňuje postup násobení víceciferného čísla číslem jednociferným a vedle toho variantu zobecněnou na bloky obecné velikosti. Násobením čísla velikosti  $N$  může vzniknout výsledek velikosti  $N + 1$ , což je na obrázku 15 b) označeno znakem '?'. Jak s možným přetečením naložit záleží na programátorovi.

Po vysvětlení principu násobení je možno celý postup implementovat v JSI. Nejprve prototyp funkce v jazyce C:

```
// a *= b
void mul_intNint32 ( int *a, int b, int N );
```

Implementace násobení v JSI bude následující:

---

```
mul_intNint32 :
    enter 0,0
    push ebx                ; save registers
    push edi
    mov ebx, [ ebp + 8 ]    ; a
    mov ecx, [ ebp + 16 ]  ; N
    mov edi, 0              ; carry
.back
    mov eax, [ ebx ]       ; eax = *a
    mul dword ptr [ ebp + 12 ] ; eax *= b
    add eax, edi           ; eax += carry
    adc edx, 0             ; edx += cf
    mov [ ebx ], eax       ; *a = eax
    add ebx, 4             ; a++
    mov edi, edx           ; new carry
    loop .back
    ; edi                  ; overflow?
    pop edi                ; restore registers
    pop ebx
    leave
    ret
```

---

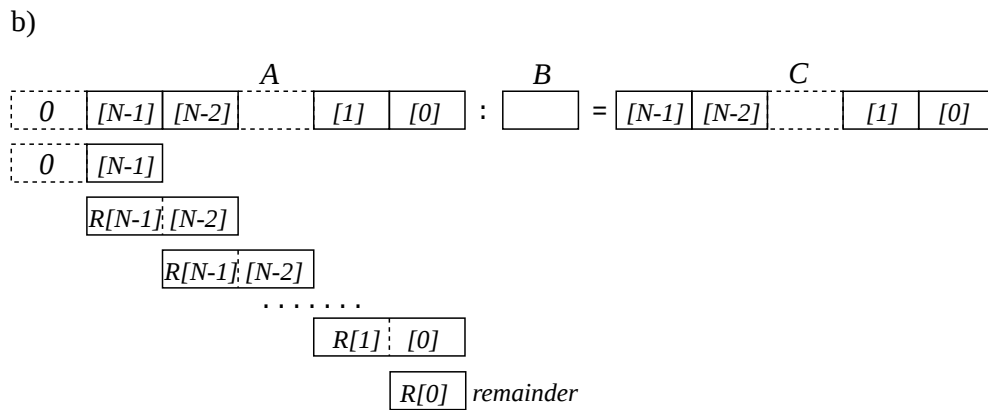
Uvedený kód umožňuje násobit **pouze čísla kladná**.

### 10.2.6 Dělení a zbytek po dělení čísla $\text{int}_N$ a $\text{int}_{32}$

Postup dělení čísla  $\text{int}_N$  číslem  $\text{int}_{32}$  je znázorněn na obrázku 16 a) pomocí postupu dělení víceciferného čísla jednociferným číslem.

a)

$$\begin{array}{r}
 73571 : 6 = 12254(1) \\
 \underline{13} \\
 15 \\
 \underline{37} \\
 21 \\
 \underline{1} \\
 1
 \end{array}$$



Obrázek 16: Dělení čísla  $\text{int}_N$  a  $\text{int}_{32}$

Stejně jako v předchozích kapitolách, lze i zde zobecnit uvedený postup pomocí bloků obecné velikosti. Toto zobecnění je uvedeno na obrázku 16 b). Jedná se o rozšíření principu popsaného v kapitole 10.1.5. Postupným dělením jsou získávány jednotlivé řády výsledku a vždy i odpovídající zbytky po dělení, které jsou v obrázku označeny jako  $R[n]$ . Každý zbytek po dělení pokračuje do dalšího dělení v horním řádu. Poslední zbytek po dělení je označen jako  $R[0]$ .

Implementace popsaného postupu dělení v JSI bude následovat. Nejprve prototyp funkce v jazyce C:

```
// a /= b; return remainder
int div_intNint32 ( int *a, int b, int N );
```

Implementace násobení v JSI bude následující:

---

```
div_intNint32 :
    enter 0,0
    push ebx                ; save register
    mov ebx, [ ebp + 8 ]    ; a
    mov ecx, [ ebp + 16 ]   ; N
    mov edx, 0              ; remainder = 0
.back
    mov eax, [ ebx + ecx * 4 - 4 ] ; eax = a[ ecx - 1 ]
    div dword [ ebp + 12 ]    ; eax-edx /= b
    mov [ ebx + ecx * 4 - 4 ], eax ; a[ ecx - 1 ] = eax
    loop .back
    mov eax, edx              ; eax = remainder
    pop ebx                  ; restore register
    leave
    ret
```

---

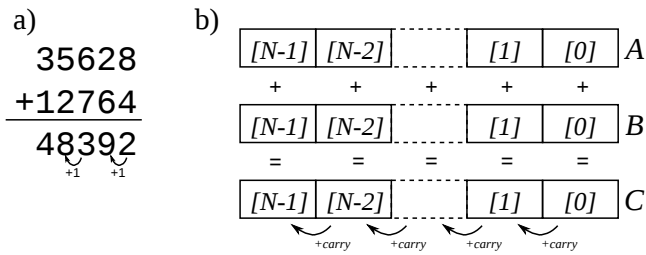
Uvedený kód umožňuje dělit **pouze čísla kladná**.

### 10.2.7 Implementace převodu čísla intN na řetězec a zpět

Dle postupu popsaného v kapitole 10.2.3, lze s využitím funkcí popsaných v předchozích třech podkapitolách implementovat převodní funkce pro čísla intN. Kostra těchto funkcí, včetně vyzkoušených funkcí z předchozích kapitol, je k dispozici v příloženém archívu `soj-bignum1.tgz`.

### 10.2.8 Sčítání a odčítání čísel intN

Princip sčítání dvou čísel velikosti intN je odvozen od sčítání popsaného v kapitole 10.2.4. Na obrázku 17 a) je postup sčítání znázorněn nejprve na příkladu sčítání dvou vícemístných čísel v dekadické soustavě.



Obrázek 17: Sčítání čísel `intN`

Při čítání dvou čísel plné velikosti `intN` obsahují oba sčítance *A* i *B* stejný počet platných řádů a sčítání lze provádět postupně od nejnižšího řádu až po řád nejvyšší. Při sčítání je potřeba předávat správně přenos mezi řády. Postup sčítání je v zobecněné formě znázorněn na obrázku 17 b) pomocí obecných bloků.

Dle popsaného principu lze snadno implementovat funkci pro sčítání čísel `intN` v JSI. Nejprve však prototyp funkce v jazyce C:

```

// a += b
void add_intNintN ( int *a, int *b, int N );

```

Implementace v JSI bude následující:

---

```

add_intNintN :
    enter 0,0
    push edi                    ; save registers
    push esi
    mov edi, [ ebp + 8 ]        ; a
    mov esi, [ ebp + 12 ]       ; b
    mov ecx, [ ebp + 16 ]       ; N
    mov edx, 0                  ; inx = 0
    cld                          ; cf = 0
.back :
    mov eax, [ esi + edx * 4 ]
    adc [ edi + edx * 4 ], eax   ; a[inx]+=b[inx]+cf
    inc edx
    loop .back
    pop esi                      ; restore registers
    pop edi
    leave
    ret

```

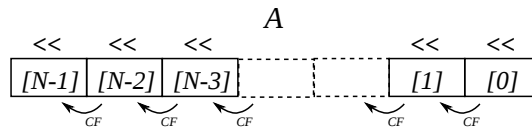
---

Navrženou funkci lze snadno upravit i pro odčítání čísel `intN`. Stačí nahradit instrukci `ADC` instrukcí `SBB`.



### 10.2.9 Bitový posun čísla $\text{int}N$ vlevo a vpravo o 1 bit

Operace bitového posunu bude asi nejjednodušší operací implementovanou pro čísla velikosti  $\text{int}N$ . Číslo  $\text{int}N$  je uloženo v poli a tvoří souvislý paměťový blok. Posun o jeden bit v tomto bitovém řetězci je velmi snadný. Stačí využít instrukce RCL nebo RCR a pro přenos bitu mezi sousedními bloky využít CF. Na obrázku 18. je znázorněn příklad posunu jednoho bitu vlevo v čísle  $A$ .



Obrázek 18: Bitový posun doleva

Následnému kódu funkce v JSI pro bitový posun vlevo bude předcházet prototyp funkce v jazyce C:

```
// a <<= 1
void shl_intN( int *a, int N );
```

Implementace v JSI bude následující:

---

```
shl_intN :
    enter 0,0
    mov edx, [ ebp + 8 ]           ; a
    mov ecx, [ ebp + 12 ]        ; N
    mov eax, 0                   ; inx = 0
    cld                          ; cf = 0
.back
    rcl dword [ edx + eax * 4 ], 1 ; a[inx]<<=1
    inc eax                      ; inx++
    loop .back
    leave
    ret
```

---

Implementovaný kód lze snadno použít i pro implementaci posunu o jeden bit vpravo. Je potřeba pouze zaměnit instrukci RCL za instrukci RCR a bitový posun provádět od nejvyššího řádu čísla  $A$ .

Pro posun o více bitů je potřeba funkci volat opakovaně. Nemá však smysl provádět posuny o větší počet bitů, než 7. Jakýkoliv posun o násobek 8 bitů lze nahradit přímo posunem paměti. Teprve po paměťovém posunu se provede potřebný počet volání funkce bitového posunu.

### 10.2.10 Bitový posun vlevo a vpravo o více bitů

V instrukčním souboru jsou obsaženy i instrukce pro bitový posun přes dva operandy. Jsou to instrukce SHRD a SHLD. Tyto instrukce lze s výhodou použít pro bitové posuny od 1 do 31 bitů (posun o větší počet bitů lze provést posunem obsahu paměti).

Princip použití obou instrukcí je snadný a není k němu potřeba grafické znázornění. Jako příklad bude následovat implementace bitového posunu vlevo. Prototyp funkce v jazyce C může vypadat následovně:

```
// a <<= bits
void shld_intN( int *a, int bits, int N );
```

Implementace v JSI bude následující:

---

```
shld_intN :
    enter 0,0
    push ebx                ; save register
    mov edx, [ ebp + 8 ]   ; a
    mov ecx, [ ebp + 12 ]  ; bits
    mov ebx, [ ebp + 16 ]  ; N
    dec ebx                ; skip first int
.back
    mov eax, [ edx + ebx * 4 - 4 ] ; eax = a[ N - 1 ]
    shld [ edx + ebx * 4 ], eax, cl ; edx-eax <<= bits
    dec ebx                ; N--
    jnz .back
    shl dword [ edx ], cl  ; a[0] <<= bits
    pop ebx                ; restore register
    leave
    ret
```

---

Funkci pro bitový posun vpravo o více bitů lze na základě uvedené kódu implementovat pouze s drobnými úpravami. Je potřeba nahradit instrukci SHLD instrukcí SHRD a posuny bitů realizovat od spodních řádů čísla *A*.

### 10.2.11 Násobení čísel `intN`

Násobení dvou čísel velikosti `intN` je možno si představit jako násobení dvou víceciferných čísel pod sebou. Názorně je to vidět obrázku 19 a).

a)	$\begin{array}{r} 372 \\ * 468 \\ \hline 2976 \\ + 2232 \\ +1488 \\ \hline =174096 \end{array}$	b)	$\begin{array}{r} 1001 \quad A \\ * 1011 \quad B \\ \hline 1001 \quad (A \ll 0) * B_0 \\ + 1001 \quad (A \ll 1) * B_1 \\ + 0 \quad (A \ll 2) * B_2 \\ +1001 \quad (A \ll 3) * B_3 \\ \hline =1100011 \quad C \end{array}$
----	---	----	---

Obrázek 19: Násobení čísel `intN`

Takový postup násobení se rozloží na několik násobení čísla víceciferného číslem jednociferným.

Ve dvojkové soustavě se celý postup ještě více zjednoduší. Příklad je na obrázku 19 b). Násobení čísla  $A$  číslem  $B$  ve dvojkové soustavě znamená postupné násobení čísla  $A$  jednotlivými bity čísla  $B$ . Násobení číslem 1 a 0 znamená pouze opis čísla  $A$  nebo 0. Opis čísla  $A$  samozřejmě s patřičným bitovým posunem vlevo.

Uvedený postup je možno ukázat na násobení dvou čísel `int32`.

---

```
int mul_int32(int a, int b)
{
    int c = 0;
    while ( b )
    {
        if ( b & 1 )
            c += a;
        a <<= 1;
        b >>= 1;
    }
    return c;
}
```

---

Uvedený postup je možno aplikovat i na násobení čísel `intN`. Stačí změnit typ parametrů na `intN` a pro sčítání i bitové posuny čísel `intN` použít funkce popsané a implementované v předchozích podkapitolách.

Prázdňá šablona kódu pro násobení čísel `intN` je připravena ve zdrojových kódech přiložených v archivu `soj-bignum2.tgz`, včetně potřebných funkcí pro implementaci.

### 10.2.12 Dělení čísel intN

Pro pochopení principu dělení dvou velkých čísel `intN` je možno použít analogii se vzájemným dělením víceciferných čísel, jak je uvedeno na obrázku 20 a).

a)	A	B	C	b)	A	B	C
	46839	: 324	= 00144		1100100	: 1011	= 0001001
	4				1		≥ B → 0
	46				11		≥ B → 0
	468				110		≥ B → 0
	<u>- 324</u>				1100		≥ B → 1
	= 1443				<u>- 1011 (=B)</u>		
	<u>- 1296 (=4*324)</u>				= 00011		≥ B → 0
	= 01479				110		≥ B → 0
	<u>- 1296 (=4*324)</u>				1100		≥ B → 1
	= 0183		remainder		<u>- 1011 (=B)</u>		
					= 0001		remainder

Obrázek 20: Dělení čísel `intN`

Postup dělení probíhá po jednotlivých krocích a začíná od nejvyšších řádů dělence *A*. Postupně se hledá tolik cifer, aby vytvořily číslo větší než dělitel *B*. Až se taková část čísla najde, provede se dělení a zbytek pokračuje dále, připojují se za něj další cifry dělence a proces se opakuje.

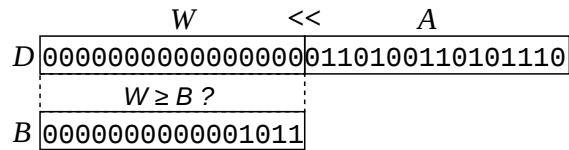
V dekadické soustavě je postup dělení na první pohled trochu nepřehledný. Pokud se ale stejný postup dělení uplatňuje ve dvojkové soustavě, celý proces se zjednoduší. Ukázka postupu je na obrázku 20 b).

Postupně se hledá v děliteli *A* tolik bitů, aby číslo jimi tvořené bylo větší než dělitel. Pokud je řetězec bitů menší než *B*, do výsledku se vkládá 0. V případě splnění podmínky se od bitového řetězce odečítá dělitel *B* (bez nutnosti násobení) a do výsledku se vkládá číslo 1. Ke zbytku se dále přidávají bity z dělence *A* a opět se hledá číslo větší než dělitel. Proces se opakuje až do vyčerpání bitů dělence.

Z popsaného postupu a dle obrázku 20 b) je zřejmé, že proces dělení je v podstatě pouze postupný výběr bitů z dělence a podmíněné odčítání. Splnění či nesplnění podmínky vkládá do výsledku *C* čísla 1 či 0.

Z popsaného postupu však vzniká jeden technický problém. Jak vybírat postupně bity dělence a porovnávat je s dělitelem.

Řešení tohoto problému se obchází pomocným bitovým polem dle obrázku 21.



Obrázek 21: Realizace kroků dělení

Před dělením je potřeba připravit v programu bitové pole dvojnásobné velikosti, než je velikost dělence  $A$ . Na obrázku 21 je toto pole označeno jako  $D$  (double). Do spodní poloviny  $D$  se vloží dělenec  $A$  a horní polovina se vyplní nulami. Horní část se označí jako  $W$  (working).

Bitovým posunem  $D$  o jeden bit vlevo se vždy v pracovní části  $W$  objeví postupně jeden bit dělence  $A$ . Obsah  $W$  lze již snadno porovnat s velikostí dělitele  $B$  a dle splnění podmínky provést odčítání.

Pro názornost je uveden následující kód v jazyce C, který popsany postup dělení implementuje pro číslo `int32`.

---

```
int div_int32int32 ( int a, int b )
{
    long long d;
    int *tmp = ( int * ) &d;
    int *w = tmp + 1;
    *tmp = a;
    *w = 0;
    int c = 0;
    for ( int i = 0; i < sizeof ( a ) * 8; i++ )
    {
        d <<= 1;
        c <<= 1;
        if ( *w >= b )
        {
            *w -= b;
            c |= 1;
        }
    }
    return c; // *w remainder
}
```

---

Stejně jako u násobení čísel `intN` v předchozí podkapitole, stačí i zde nahradit typ čísla typem `intN` a aritmetické operace nahradit voláním i funkcí z předchozích podkapitol. Šablona kódu pro dělení čísel `intN` je připravena ve zdrojových kódech příložených v archivu `soj-bignum2.tgz`.

## 11 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**