

Programová knihovna pro podporu celočíselné aritmetiky na mikropočítačích Freescale HC08

Programming library to support integer arithmetics on Freescale HC08 microcontrollers

Dušan Šild



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Dušan ŠILD**
Osobní číslo: **A07102**
Studijní program: **B 3902 Inženýrská informatika**
Studijní obor: **Informační a řídicí technologie**

Téma práce: **Programová knihovna pro podporu celočíselné aritmetiky na mikropočítačích Freescale HC08**

Zásady pro vypracování:

1. Prostudujte instrukční sadu mikropočítače rodiny HCS08 s centrální procesní jednotkou CPU08 se zaměřením na aritmetické operace.
2. Navrhněte vhodnou strukturu knihovny pro celočíselnou aritmetiku s podporou datových typů byte, word a long se znaménkem i bez znaménka včetně konverzních funkcí.
3. Vytvořte knihovnu v jazyce symbolických adres mikropočítačů rodiny HCS08.
4. Proveďte porovnání z hlediska rychlosti a obsazené paměti Vámi vytvořených knihovních funkcí s obdobnými funkcemi implementovanými v C jazyce.
5. Vypracujte ukázkový program používající vytvořenou knihovnu pro demonstraci její funkce.

Rozsah bakalářské práce:

Rozsah příloh:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

1. BURKHARD, Mann. C pro mikrokontroléry. Praha : BEN – technická literatura, 2003. 280 s. ISBN 80-7300-077-6.
2. Freescale Semiconductor. CPU08 Central Processor Unit Reference Manual., 2001. Dostupný z WWW:
3. Freescale Semiconductor. HCS08 Family Reference Manual, Rev.1., 2003. Dostupný z WWW:
4. Freescale Semiconductor. MC9S08GB/GT Data Sheet, Rev.2.3., 2004. Dostupný z WWW:
5. VÁŇA V.: Začínáme s mikrokontroléry Motorola HC08 Nitron. Praha: BEN ? technická literatura, 2003. 96 s. ISBN 80-7300-124-1

Vedoucí bakalářské práce:

Ing. Petr Dostálek, Ph.D.

Ústav automatizace a řídicí techniky

Datum zadání bakalářské práce:

25. února 2011

Termín odevzdání bakalářské práce:

7. června 2011

Ve Zlíně dne 25. února 2011



prof. Ing. Vladimír Vašek, CSc.
děkan



prof. Ing. Vladimír Vašek, CSc.
ředitel ústavu

ABSTRAKT

Práce se zabývá problematikou realizace aritmetických operací s celými čísly v programové knihovně jazyka Assembler. Probírá a vysvětluje jednotlivé algoritmy aritmetických operací, ukazuje na možné problémy a popisuje jejich příčinu i řešení.

Praktická část osvětluje implementaci těchto algoritmů v jazyce Assembler a rozebírá návrh celé programové knihovny. Dále srovnává rychlost vytvořených funkcí s funkcemi jazyka C a popisuje jednoduchý program využívající danou knihovnu pro výpočty.

Klíčová slova:

Mikroprocesor, Assembler, programová knihovna, aritmetické operace.

ABSTRACT

The work deals with the implementation of arithmetic operations with whole numbers in assembly language programming library. It discusses and explains the algorithms of arithmetic operations, points to possible problems and describes their causes and possible solutions.

The practical section highlights the implementation of these algorithms in assembly language, and examines the design throughout the program library. Furthermore, comparing the speed generated by functions with the functions of the C language and describes a simple program that uses the library for calculations.

Keywords:

Microcontroller, Assembler, programming library, arithmetic operations.

Chtěl bych poděkovat Všem, kteří mně s touto bakalářskou prací podporovali a pomáhali – své rodině a vedoucímu práce.

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

.....
podpis diplomanta

OBSAH

ÚVOD.....	9
I TEORETICKÁ ČÁST	10
1 MIKROPROCESOR FREESCALE HCS08	11
1.1 ZÁKLADNÍ INSTRUKCE PRO OPERACI S REGISTRY	13
1.1.1 Instrukce LDA, LDX, LDHX	13
1.1.2 Instrukce STA, STX, STHX	13
1.1.3 Instrukce PSHA, PSHX, PSHH	13
1.1.4 Instrukce PULA, PULX, PULH.....	14
1.2 INSTRUKCE ARITMETICKÝCH OPERACÍ	14
1.2.1 Instrukce ADD, ADC.....	14
1.2.2 Instrukce SUB, SBC.....	14
1.2.3 Instrukce MUL	14
1.2.4 Instrukce DIV	14
1.2.5 Instrukce INC, INCA, INCX	14
1.2.6 Instrukce DEC, DECA, DECX	14
1.3 INSTRUKCE PRO POROVNÁVÁNÍ.....	15
1.3.1 Instrukce CMP, CPX, CPHX	15
1.4 INSTRUKCE MOV	15
2 ARITMETICKÉ OPERACE S BINÁRNÍMI ČÍSLY	16
2.1 SČÍTÁNÍ BINÁRNÍCH ČÍSEL	16
2.1.1 Sčítání neznaménkových čísel	16
2.1.2 Sčítání kladných znaménkových čísel	17
2.1.3 Sčítání záporných znaménkových čísel	17
2.1.4 Sčítání různých čísel, z nichž jedno je záporné.....	18
2.2 ODČÍTÁNÍ BINÁRNÍCH ČÍSEL	19
2.3 NÁSOBENÍ BINÁRNÍCH ČÍSEL	20
2.4 DĚLENÍ BINÁRNÍCH ČÍSEL	21
II PRAKTICKÁ ČÁST	24
3 NÁVRH MATEMATICKÉ KNIHOVNY	25
3.1 FUNKCE PRO NAČÍTÁNÍ DAT DO AKUMULÁTORŮ	26
3.1.1 Funkce GETA a GETB	26

3.2	FUNKCE PRO UKLÁDÁNÍ DAT Z AKUMULÁTORŮ	27
3.2.1	Funkce PUTA a PUTB.....	27
3.3	FUNKCE PRO PROHOZENÍ AKUMULÁTORU A FUNKCE PRO POROVNÁVÁNÍ.....	28
3.3.1	Funkce SWAB	28
3.3.2	Funkce CMPAB	28
3.4	ARITMETICKÉ FUNKCE	29
3.4.1	Funkce ADDAB	29
3.4.1.1	Sčítání neznaménkových čísel ADDAB_UNSGN.....	29
3.4.1.2	Sčítání kladných znaménkových čísel ADDAB_0NEG.....	30
3.4.1.3	Sčítání záporných znaménkových čísel ADDAB_2NEG.....	31
3.4.1.4	Sčítání různých čísel, z nichž jedno je záporné ADDAB_ANEG a ADDAB_BNEG.....	31
3.4.2	Funkce SUBAB.....	32
3.4.3	Funkce MULAB.....	32
3.4.4	Funkce DIVAB	34
3.5	DOPLŇKOVÉ FUNKCE	37
3.5.1	Funkce STR2INT	37
3.5.2	Funkce INT2STR	37
3.5.3	Funkce GETSA a GETSB.....	37
4	SROVNÁNÍ S FUNKCEMI JAZYKA C	38
4.1	SČÍTÁNÍ.....	39
4.2	ODČÍTÁNÍ	42
4.3	NÁSOBENÍ	44
4.4	DĚLENÍ.....	47
5	PŘÍKLAD VYUŽITÍ KNIHOVNY	50
	ZÁVĚR	51
	ZÁVĚR V ANGLIČTINĚ.....	52
	SEZNAM POUŽITÉ LITERATURY.....	53
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....	54
	SEZNAM OBRÁZKŮ A GRAFŮ	55
	SEZNAM TABULEK.....	56

ÚVOD

Mikroprocesory HCS08 jsou používány pro různé účely a v mnoha případech je potřeba provádět složitější výpočty, nebo případně jednoduché s velkými čísly. K tomuto dochází například při použití mikroprocesoru jako časoměry (sčítání mezičasů), čítače průchodů, kalkulačky atd.

U složitějších výpočtů je hlavním problémem omezení na maximální velikosti registrů procesoru a typy operandů instrukcí aritmetických operací. Proto u těchto výpočtů musí autor programu navrhnout funkce, které budou aritmetické operace s většími čísly obstarávat. Tyto funkce je pak vhodné umístit do jednotné knihovny, aby byly snadno přenositelné do dalších programů. Pak by autor programu nemusel znovu s těmito funkcemi ztrácet čas při vývoji dalších programů. Toto je jedním z hlavních důvodů pro vytvoření programové knihovny uvedené v této práci pro podporu celočíselné aritmetiky.

Každá aritmetická funkce má určitý algoritmus, dle kterého probíhá výpočet výsledku. Tyto algoritmy je nutno upravit pro použití se zadanými typy čísel. V teoretické části této bakalářské práce jsou vysvětleny obecné algoritmy aritmetických operací navržené pro proměnný počet bitů, uvedení problémů, které jsou s nimi spojeny a jejich řešení. V praktické části bakalářské práce je probrán návrh knihovny s popisem jednotlivých funkcí. Funkce využívají částečně upravené algoritmy z teoretické části, protože ve skutečném výpočtu je počet bitů čísel pevně dán jejich typem. Pro vytvoření knihovny bylo použito vývojové prostředí CodeWarrior firmy Freescale. Program byl poskytnut vedoucím práce.

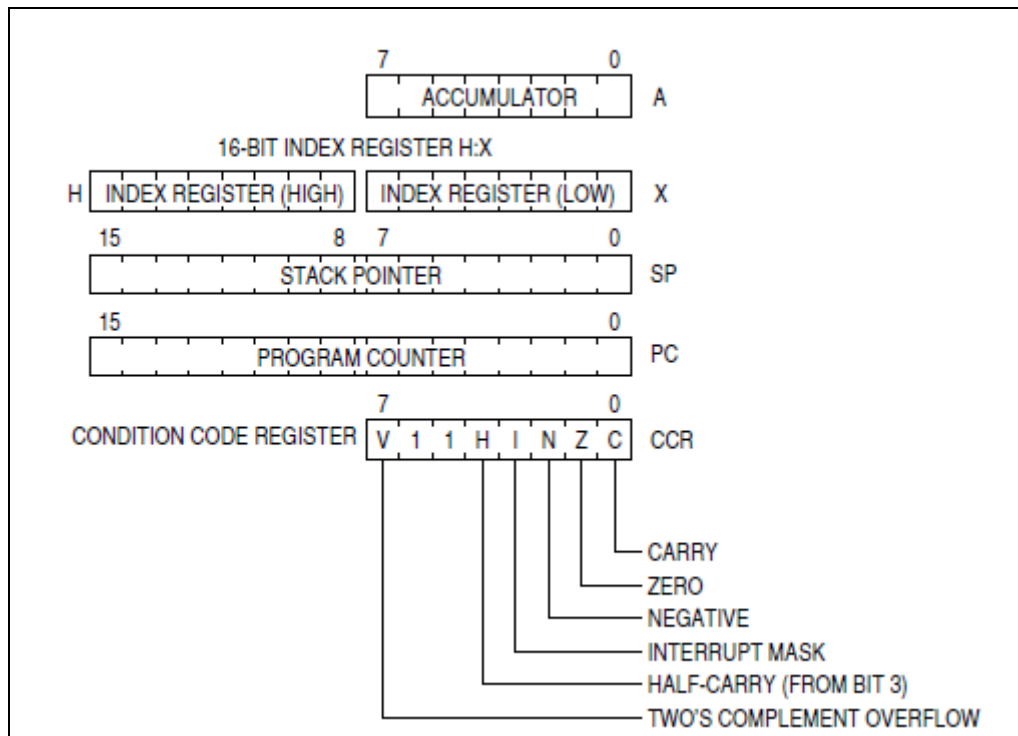
Dále se práce zabývá srovnáním navržených funkcí s funkcemi aritmetických operací jazyka C. Výsledky porovnání jsou zobrazeny v přehledných grafech za účelem vytvoření ucelenějšího pohledu na výkon funkcí. Testování proběhlo v simulačním režimu vývojového prostředí.

Na konci práce je uveden program, který vytvořenou knihovnu úspěšně používá.

I. TEORETICKÁ ČÁST

1 MIKROPROCESOR FREESCALE HCS08

Mikroprocesor HCS08 patří do rodiny výkonných a levných 8-mi bitových kontrolérů společnosti Freescale. Jádro těchto mikroprocesorů běží na frekvenci 40MHz. Procesor obsahuje rychlé vnitřní registry, tyto jsou vyobrazeny na obrázku níže.

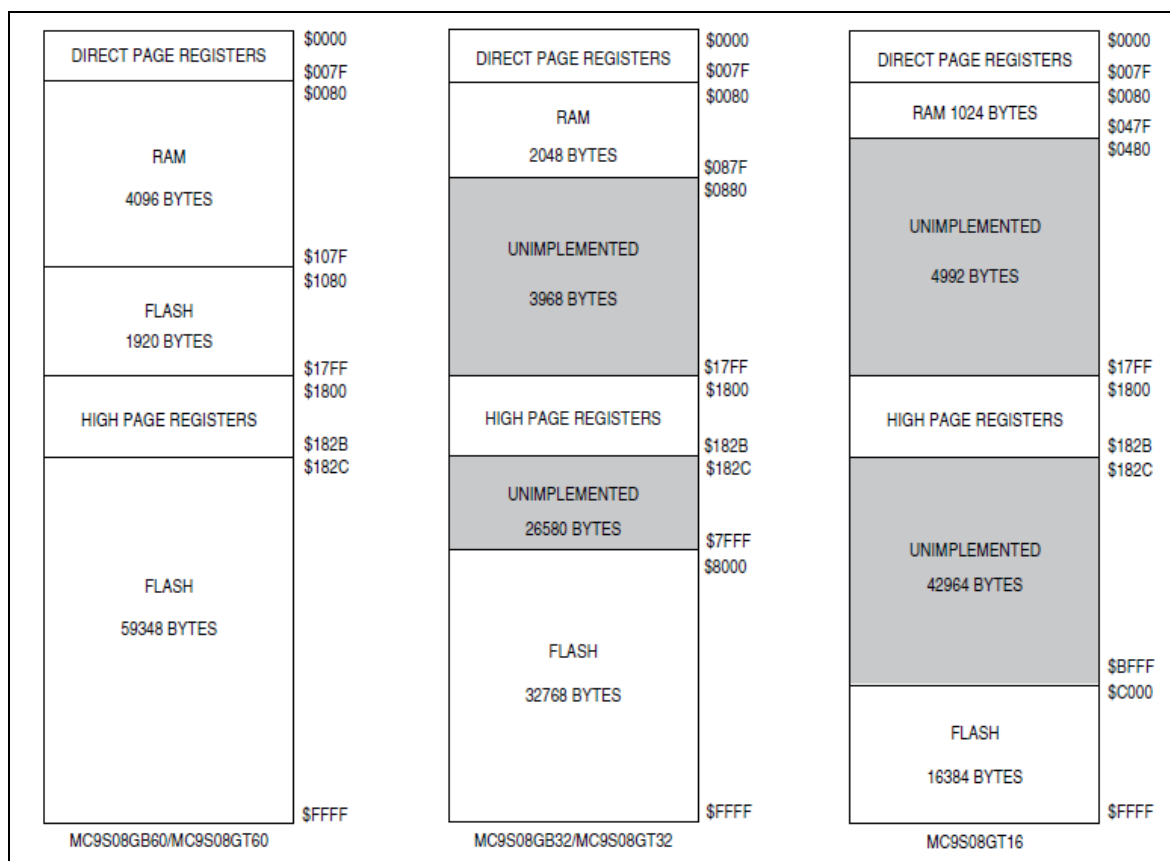


Obr. 1. Vnitřní registry mikroprocesoru HCS08

- Registr A je víceúčelový 8-mi bitový registr nazývaný akumulátor (Accumulator).
- Registr H:X je 16-ti bitový registr, který se skládá ze dvou 8-mi bitových a lze je použít samostatně. Jeho největší výhodou je ovšem použití obou registrů dohromady jako 16-ti bitový ukazatel na adresu do paměti. Registr H ukládá horní byte adresy a registr X dolní byte.
- Registr SP (Stack Pointer) je 16-ti bitový ukazatel na následující volnou adresu v automatickém zásobníku typu LIFO (Last-In-First-Out). Tento zásobník může být umístěn kdekoli v oblasti RAM paměti.
- Registr PC (Program Counter) je 16-bitový registr ukládající adresu další instrukce kódu, která má být načtena. Tuto hodnotu mění instrukce skoků, větvení a volání.
- Registr CCR (Condition Code Register) je 8-mi bitový registr indikující výsledek právě dokončené instrukce hodnotou příslušných bitů.

Paměť mikroprocesoru se skládá z oblasti RAM paměti, FLASH paměti pro program, registrů pro stav a řízení a vstupně-výstupních registrů. Registry se dělí na 3 skupiny:

- registry s přímým přístupem (adresy: \$0000 až \$007F),
- registry s nepřímým přístupem (adresy: \$1800 až \$182B),
- neměnné (statické) registry (na adresách: \$FFB0 až \$FFBF).



Obr. 2. Rozložení paměti v mikroprocesoru HCS08

Registry s přímým přístupem jsou nejčastěji používány, proto jsou umístěny do prvních 128 bytů paměti. Jedná se o většinu vstupně-výstupních a řídicích registrů.

Registry s nepřímým přístupem se nepoužívají často, proto jsou odsunuty až za oblast přímého přístupu do paměti.

Mikropocesory HCS08 přebírají instrukční sadu od mikroprocesorů HC08 a podporují navíc instrukci BGND.

Kompletní instrukční sada mikroprocesoru Freescale HCS08 je detailně popsána v podrobném manuálu procesoru CPU08 [1] nebo v krátkém přehledu v manuálu mikrokontroléru [5].

V dalších kapitolách této práce budou zmíněny a popsány jen nejdůležitější instrukce mikroprocesoru.

1.1 Základní instrukce pro operaci s registry

1.1.1 Instrukce LDA, LDX, LDHX

Instrukce LDA načte byte z požadované oblasti do registru A. U této instrukce lze využívat přímé i nepřímé adresování, načítání ze zásobníku nebo okamžitou hodnotu zadanou jako operand instrukce. Výhodou této instrukce je, že po jejím vykonání je ihned upraven registr CCR následovně:

- $V = 0$,
- $Z = 1$ pokud načtený byte má nulovou hodnotu, jinak $Z = 0$,
- $N = 1$, pokud je nejvyšší bit čísla též 1.

Tímto je umožněno provádět okamžité větvení bez nutnosti dodatečného porovnání.

Obdobnými instrukcemi jsou LDX a LDHX. Instrukce LDX má stejnou funkci jako LDA, jediným rozdílem je, že načítá do registru X. LDHX pracuje se 16-ti bitovým rozsahem, tzn. načítá rozsah o velikosti dva byty do registrů H a X.

1.1.2 Instrukce STA, STX, STHX

STA a STX slouží pro uložení aktuální hodnoty vnitřních registrů A a X do paměti na místo, které je určeno operandem instrukce a to v několika adresovacích režimech.

Instrukce STHX ukládá na určené místo v paměti obsah celého registru H:X, tedy celé dva byty. Instrukce podporuje pouze přímou adresaci nebo využití hodnoty SP registru pro uložení do zásobníku.

1.1.3 Instrukce PSHA, PSHX, PSHH

Pomocí těchto instrukcí se vloží hodnota registru A, X nebo H na vrchol zásobníku. Hodnota registru SP se sníží o jedničku, aby ukazovala na další volný byte. Všechny tři instrukce pracují s jedním bytem registru A, X nebo H, tedy nelze vložit obsah H:X jedinou instrukcí.

1.1.4 Instrukce PULA, PULX, PULH

Opakem instrukcí pro vložení na zásobník jsou instrukce pro sejmutí vrcholu zásobníku. Byte ze zásobníku je vložen do odpovídajícího registru a hodnota SP zvýšena o jedničku. I tyto instrukce pracují s jedním bytem, takže nelze jedinou instrukcí naplnit registr H:X.

1.2 Instrukce aritmetických operací

1.2.1 Instrukce ADD, ADC

Instrukce aritmetického sčítání ADD přičítá číslo z paměti určené operandem k registru A. ADC přičte k součtu registru A a čísla z paměti ještě bit C z registru CCR. Podle výsledku je následně upraven stav registru CCR.

1.2.2 Instrukce SUB, SBC

Aritmetické odčítání, SUB a SBC, odečte od registru A číslo z paměti určené operandem. SBC odečítá také bit C registru CCR. Po provedení těchto instrukcí je upraven stav registru CCR.

1.2.3 Instrukce MUL

Jedná se o aritmetické násobení čísel v registru A a X. Výsledek je pak uložen v obou těchto registrech, kdy v registru A je uložen nižší byte výsledku a v X vyšší byte výsledku. Po vynásobení se v registru CCR nulují bity H a C.

1.2.4 Instrukce DIV

Instrukce aritmetického celočíselného dělení DIV dělí číslo, které je umístěno nižším bytem v registru A a vyšším bytem v registru H, číslem v registru X. Výsledek je umístěn do registru A a zbytek po dělení do registru H.

1.2.5 Instrukce INC, INCA, INCX

Instrukce inkrementace INC zvětší číslo v paměti udané operandem o jedničku. INCA zvětšuje o jedničku číslo v registru A a INCX číslo v registru X.

1.2.6 Instrukce DEC, DECA, DECX

DEC dekrementuje (sníží o jedničku) číslo v paměti zadané operandem, DECA sníží o jedničku číslo v registru A a DECX číslo v registru X.

1.3 Instrukce pro porovnávání

1.3.1 Instrukce CMP, CPX, CPHX

Byte v paměti určený operandem instrukce porovnává CMP s obsahem registru A, CPX porovnává s obsahem registru X. CPHX porovnává dva byty v paměti s obsahem registrů H:X. Výsledkem porovnání je úprava stavu registru CCR, konkrétně bitů V, N, Z nebo C.

1.4 Instrukce MOV

Instrukce MOV je speciální instrukce pouze pro paměťovou oblast, která přenesení obsah jednoho bytu v paměti do druhého. Ve funkcích matematické knihovny je využita především k vynulování akumulátorů, protože přesunutí čísla 0 na specifikovaný byte paměti trvá o jeden cyklus procesoru méně než nulování pomocí instrukce CLR.

2 ARITMETICKÉ OPERACE S BINÁRNÍMI ČÍSLY

Zadáním práce je vytvořit knihovnu pro celočíselnou aritmetiku binárních čísel různých typů. Čísla mohou být typu Byte (1B) až typu Long (4B) anebo neznaménkové i znaménkové čísla. Z tohoto důvodu nejsou aritmetické operace s nimi vždy triviální a je potřeba prověřit případné úskalí.

2.1 Sčítání binárních čísel

Binární čísla je možné sčítat stejným způsobem, jakým se sčítají čísla desítková. V případě dekadických čísel, pokud se sčítají čísla s více číslicemi, se v případě výsledku většího než devět musí provádět tzv. přenos (anglicky Carry). U binárních čísel je to stejné:

A	B	A + B
0	0	0
1	0	1
0	1	1
1	1	10

K přenosu jedničky do vyššího řádu tedy dochází tehdy, je-li výsledkem součtu dvou čísel hodnota větší nebo rovna 10b. Podle tohoto pravidla se sčítají postupně všechny bity dvou čísel, jak je vidět z příkladu níže:

$$\begin{array}{r}
 01011110 \\
 01000001 \\
 \hline
 10011111
 \end{array}$$

Při sčítání různých čísel je nutno nejdříve rozlišit jejich typ. Celkově se lze ve sčítání setkat se čtyřmi rozdílnými případy:

- sčítání neznaménkových čísel,
- sčítání kladných znaménkových čísel nebo kladného a neznaménkového čísla,
- sčítání záporných znaménkových čísel,
- sčítání různých čísel, z nichž jedno je záporné.

2.1.1 Sčítání neznaménkových čísel

U sčítání neznaménkových čísel se nenarazí na větší problémy. Sčítají se odpovídající byty a přičítává případný přenos.

Například:

$$44477 + 1241 = 45718$$

V binární podobě to vypadá následovně:

$$\begin{array}{r} 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1 \\ \underline{0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1} \\ \underline{1\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0} \end{array}$$

2.1.2 Sčítání kladných znaménkových čísel

U kladných znaménkových čísel se postupuje stejně jako u neznaménkových, ale navíc se hlídá znaménkový bit výsledku.

Číslo 112 je vyjádřeno jako:

$$\begin{array}{l} 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0 \leftarrow \text{hodnota čísla, 112} \\ \uparrow \\ \text{Znaménkový bit, +} \end{array}$$

Když se k němu přičte číslo 57, které je vyjádřeno takto:

$$\begin{array}{l} 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1 \leftarrow \text{hodnota čísla, 57} \\ \uparrow \\ \text{Znaménkový bit, +} \end{array}$$

Výsledek:

$$\begin{array}{r} 0\ 1\ 1\ 1\ 0\ 0\ 0\ 0 \\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1 \\ \underline{1\ 0\ 1\ 0\ 1\ 0\ 0\ 1} \end{array}$$

Správný výsledek by měl být 169, ovšem současný rozsah 8-mi bitů má maximální kladnou hodnotu +127, takže číslo 169 přeteče na hodnotu -87. Proto pokud dojde k přetečení do znaménkového bitu, musí se samotný znaménkový bit posunout a rozšířit rozsah.

$$\begin{array}{ccc} 1\ 0\ 1\ 0\ 1\ 0\ 0\ 1 & \longrightarrow & 0\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 1 \\ \underbrace{\hspace{1.5cm}} & & \underbrace{\hspace{1.5cm}} \\ 8\ \text{bitů} & & 9\ \text{bitů} \end{array}$$

2.1.3 Sčítání záporných znaménkových čísel

Záporná znaménková čísla jsou vyjádřena dvojkovým doplňkem, který umožňuje jednoduše sčítat libovolná (záporná i kladná) čísla v tomto tvaru. Jediný problém nastává v situacích, kdy mají čísla různý počet bitů.

Například:

Číslo -1594 1 1 1 1 1 0 0 1 1 1 0 0 0 1 1 0 16 bitů
 A číslo -77 1 0 1 1 0 0 1 1 8 bitů

Tato dvě čísla se sečtou, ale prostý součet povede ke špatnému výsledku. Seřadí-li se čísla nejnižšími byty k sobě, potom lze pozorovat, že se znaménkové bity nachází na jiných pozicích:

```

  1 1 1 1 1 0 0 1 1 1 0 0 0 1 1 0
    1 0 1 1 0 0 1 1
  -----
  1 1 1 1 1 0 1 0 0 1 1 1 0 0 1
  
```

Výsledkem je číslo -1415 namísto správného -1671. Řešením problému je rozšířit menší číslo jedničkami zleva tak, aby se počet bitů vyrovnal, pak lze obě čísla jednoduše sečíst:

```

  1 1 1 1 1 0 0 1 1 1 0 0 0 1 1 0
  1 1 1 1 1 1 1 1 1 0 1 1 0 0 1 1
  -----
  1 1 1 1 1 1 0 1 0 0 1 1 1 0 0 1
  
```

Nyní je výsledek správně: -1671. Přebývajících jedničky na začátku čísla je možno odstranit až na jednu, která zastane funkci znaménkového bitu, takže výsledek se může zkrátit na níže uvedené:

```

  1 0 1 0 0 1 1 1 1 0 0 1
  
```

2.1.4 Sčítání různých čísel, z nichž jedno je záporné

Při sčítání různých čísel se vyskytne více problémů, prvním je již zmíněný problém s rozdílnou délkou záporných čísel. Proto se i zde musí srovnávat rozsahy záporného čísla s kladným. Kladné číslo může mít menší počet bitů než záporné, ale ne naopak.

Dalším problémem je rozdílnost rozsahů znaménkových a neznaménkových čísel. Ukázkou je součet čísel 211 a -31:

1 1 0 1 0 0 1 1	211	
1 0 0 0 0 1	-31	číslo se doplní jedničkami zleva
1 1 0 1 0 0 1 1	211	
1 1 1 0 0 0 0 1	-31	
<u>1 1 0 1 1 0 1 0 0</u>	<u>-76</u>	

Správný výsledek je však 180. Problém spočívá v úrovni znaménkového bitu čísla B, kde se nachází datový bit čísla A. Řešením je rozšíření obou čísel před sčítáním o jeden bit,

čímž se posune znaménkový bit čísla B. Přetečení do vyššího bitu je pak nutno následujícím ignorovat:

$$\begin{array}{r}
 011010011 \\
 \underline{111100001} \\
 1\underline{010110100} \leftarrow \text{přetečení ignorováno}
 \end{array}$$

Výsledek je tedy ve dvojkovém doplňku a je roven +180.

2.2 Odčítání binárních čísel

Klasický algoritmus odčítání pracuje s principem takzvaného vypůjčování, kdy je při rozdílu 0 – 1 vypůjčena jednička z následujícího bitu menšence (čísla A). Vlevo v tabulce je popsáno základní pravidlo binárního odčítání Booleovy algebry.

A	B	A – B
0	0	0
1	0	1
0	1	1
1	1	0

Princip vypůjčení jedničky:

$$\begin{array}{r}
 10 \\
 -\underline{1} \\
 1
 \end{array}
 \longrightarrow
 \begin{array}{r}
 00 \\
 -\underline{1} \\
 01
 \end{array}$$

Nejdříve se odečte 0 – 1, podle pravidla se sepíše jednička, pak se z následujícího vyššího bitu jednička zase odečte (vypůjčí) a konečný výsledek je tedy 1.

Druhým způsobem odčítání je inverze menšitele na dvojkový doplněk a sečtení obou čísel. Tento algoritmus je hojně využíván. Způsob přiblíží následující příklad:

$$\begin{array}{r}
 150 \qquad 10010110 \\
 -81 \qquad -01010001
 \end{array}$$

Zde se musí navíc ještě číslo 150 rozšířit o jeden bit, aby bylo ve správném tvaru dvojkového doplňku. Číslo 81 se invertuje na -81 a čísla se sečtou:

$$\begin{array}{r}
 010010110 \\
 \underline{110101111} \\
 1\underline{001000101}
 \end{array}$$

Jak bylo popsáno v sekci sčítání, přenos nad rozsah je ignorován a výsledkem je tedy znaménkové číslo 69.

Pravidla algoritmu, podle kterých je převedeno odčítání na sčítání, jsou následující:

$$\begin{array}{ll}
 A - B & = A + (-B) & -A - B & = -A + (-B) \\
 A - (-B) & = A + B & -A - (-B) & = -A + B
 \end{array}$$

2.3 Násobení binárních čísel

Násobení se řídí základy Booleovy algebry viz tabulka níže:

A	B	A * B
0	0	0
1	0	0
0	1	0
1	1	1

Z tabulky vyplývají již zřejmá pravidla, násobí-li se cokoli nulou, výsledek je vždy nula a násobí-li se jedničkou, číslo se nemění. Dalším pravidlem při násobení čísel je, že 0. bitem čísla B se násobí originální číslo A, prvním bitem čísla B se násobí číslo A posunutě o jedno místo doleva, druhým bitem čísla B se násobí číslo A posunutě o dvě místa doleva atd. Obecně tedy platí:

Každým bitem čísla B se násobí číslo A posunutě o tolik míst doleva, kolikátý je bit v čísle B, začínaje od 0.

Tuto definici lze převést na jednoduchý algoritmus:

1. Před začátkem násobení se výsledek vynuluje.
2. Vezme se nultý bit (první zprava) čísla B, pokud je tento bit roven 1, přičítá se číslo A do výsledku, pokud je roven 0, nepřičítá se nic navíc.
3. Posune se číslo A doleva přidáním 0 zprava.
4. Pokud má číslo B další bity, provádění algoritmu se vrací na bod 2, jinak skončí.

Celý algoritmus podrobně demonstuje práce na příkladu:

18 * 5

Výsledek: 0
A: 1 0 0 1 0
B: 1 0 1

1. krok:

Vezme se 0. bit z čísla B:

B: 1 0 **1** ← je roven 1, přičte se tedy A k výsledku

Výsledek: 1 0 0 1 0

Číslo A se posune doleva:

A: 1 0 0 1 0 0 ← vloží se 0

2. krok:

Vybere se další bit z čísla B:

B: 1 0 1 ← je roven 0, nepřičte se A

Výsledek: 1 0 0 1 0

Číslo A se posune doleva:

A: 1 0 0 1 0 0 0 ← vloží se 0

3. krok:

Další bit z čísla B:

B: 1 0 1 ← je roven 1, přičte se tedy A k výsledku

Výsledek: 1 0 1 1 0 1 0

Číslo A se posune doleva:

A: 1 0 0 1 0 0 0 0 ← vloží se 0

4. krok:

Číslo B již nemá další bity – tedy výpočet končí. Výsledek je roven 90, což je považováno za výsledek správný.

Tento algoritmus platí pro neznaménková čísla, případně pro kladná znaménková čísla, ale po skončení násobení je nutné přidat nulu jako znaménkový bit. Záporná čísla se před násobením musejí převést na kladné a po násobení případně tento výsledek převést na záporný.

2.4 Dělení binárních čísel

Při celočíselném dělení binárních čísel se nejčastěji používá algoritmus posuvů a odčítání. Tento algoritmus je přenesenou formou algoritmu pro klasické dekadické dělení. Ve zjednodušené podobě se dá znázornit na následující straně takto:

$$\begin{array}{r}
 1111101 \div 101 = \underline{11001} \\
 \underline{-101} \\
 101 \\
 \underline{-101} \\
 0101 \\
 \underline{-101} \\
 0
 \end{array}$$

Nejprve je potřeba zjistit počet bitů obou čísel, protože počet bitů čísla B udává kolik bitů zleva se vybere pro odčítání z čísla A. Pokud má tedy číslo A menší počet bitů než číslo B, je možné rovnou prohlásit, že výsledek dělení je 0 a zbytek je přímo číslo A:

Například:

$$10 \div 110 \sim 2 \div 6$$

I podle dekadického zápisu je jasné, že při celočíselném dělení je výsledek 0 a zbytek 2. Naprosto stejně je tomu i u čísel binárních. Proto je vhodné tento stav kontrolovat a v takovýchto případech se dělením nezabývat.

Přechod k samotnému algoritmu:

Číslo A = 2463 se dělí číslem B = 188

$$\begin{array}{rcl}
 2463 & = & 00001001 \ 10011111 \\
 188 & = & 10111100 \\
 & & \underbrace{\hspace{1.5cm}} \\
 & & 8 \text{ bitů}
 \end{array}$$

Spočítají se bity čísla B: 8 bitů. To je rozsah, který se vybere z čísla A od prvního jedničkového bitu zleva:

$$\begin{array}{c}
 0000 \color{red}{1001} \color{red}{1001} 1111 \\
 \underbrace{\hspace{1.5cm}} \\
 \text{Rozsah 8 bitů}
 \end{array}$$

Od vybraného rozsahu se odečte číslo B:

$$\begin{array}{r}
 10011001 \\
 \underline{10111100} \\
 \underline{11011101} \leftarrow \text{při odčítání došlo k „podtečení“}
 \end{array}$$

Při odečtení nastávají dva možné případy:

- číslo B je větší než vybraný rozsah, dojde k podtečení,
- číslo B je menší než vybraný rozsah, k podtečení nedojde.

V prvním případě se vloží do výsledku dělení 0 zprava, zvětší se o jedničku vybíraný rozsah z čísla A a číslo B se posune o jeden bit doprava (doplňuje se nulami zleva). V druhém případě lze vložit do výsledku 1, za vybraný rozsah se považuje rozdíl původního rozsahu a čísla B a přidá se k němu následující bit z čísla A. Číslo B stejně jako v prvním případě se posunují o jeden bit doprava. V příkladu nastala první možnost, proto změny budou následující:

Výsledek: 0 ← vloží se 0
 A: 0 0 0 0 1 0 0 1 1 0 0 1 1 1 1
 Rozsah 9 bitů

B: 0 → 0 1 0 1 1 1 1 0 0

Nyní se cyklus opakuje až do chvíle, kdy vybíraný rozsah bitů obsáhne celé číslo A.

1 0 0 1 1 0 0 1 1
 0 1 0 1 1 1 1 0 0
 ———
 1 1 1 0 1 1 1 ← při odečítání k „podtečení“ nedošlo

Provedené úpravy:

Výsledek: 0 1 ← vloží se 1
 A: 0 0 0 0 1 0 0 1 1 0 0 1 1 1 1
 Původní rozsah
 Vybraný rozsah: 1 1 1 0 1 1 1 1 ← následující bit
 B: 0 → 0 0 1 0 1 1 1 1 0 0

Po dalších třech opakováních už je znám výsledek. Zbytkem po dělení je číslo, které zůstalo jako vybraný rozsah po posledním cyklu:

Výsledek: 0 1 1 0 1 = 13
 Zbytek: 1 0 0 1 1 = 19

Výsledek lze snadno ověřit:

$$13 * 188 + 19 = 2444 + 19 = 2463$$

Tento algoritmus platí pro neznaménková čísla, případně čísla znaménková kladná. Záporná čísla se musí převést na kladná a uložit si příznak, podle kterého se následně invertuje i výsledek na záporný.

II. PRAKTICKÁ ČÁST

3 NÁVRH MATEMATICKÉ KNIHOVNY

Následující kapitoly praktické části této bakalářské práce se zabývají popisem vytvořené knihovny, porovnáním výkonu (rychlosti) jejích funkcí a demonstračním příkladem využití navrhované knihovny v jazyce ASM. Práce začíná shrnutím požadavků na tuto knihovnu:

- aritmetické operace musí být schopnost sčítání, odčítání, násobení a dělení 2 čísel,
- čísla mohou být typu Byte, Word (2B) i Long (4B),
- použitá čísla mohou být neznaménková i znaménková (záporná čísla jsou vyjádřena ve tvaru dvojkového doplňku),
- převod funkce z formátu ASCII znaků na číslo a naopak.

Knihovna musí splňovat tyto požadavky a zároveň nalézt kompromis mezi rychlostí a paměťovou náročností. Protože čísla mohou být větší než jeden byte, neexistují vhodné registry pro práci s nimi. Možností je čísla vložit do zásobníku a pracovat s jejich byty pomocí nepřímého adresování, ale toto řešení s sebou přináší určité zpomalení, neboť většina instrukcí trvá o cyklus déle než u přímého adresování. Kvůli tomuto rozhodnutí jsou v bakalářské práci použity paměťové „akumulátory“ o velikosti 4B (nazvané ACCA a ACCB). Tyto akumulátory jsou umístěny v paměti, a proto operace s nimi trvají déle než se skutečnými registry procesoru. Každý z těchto akumulátorů ukládá jedno číslo, které je zarovnáno doprava (nejnižší byte čísla je na nejnižším bytu akumulátoru). Typy čísel ukládají pomocné proměnné ACCAT a ACCBT o velikosti jednoho bytu a tím určují jak s číslem pracovat a jak je vyjádřeno. Například číslo 11011010b může představovat více hodnot. V neznaménkovém tvaru je to hodnota 218. Ale ve tvaru dvojkového doplňku se jedná o hodnotu -38. Neexistuje způsob jak tyto hodnoty rozlišit bez explicitního zadání tvaru. Pro tento účel práce definuje šest typů, které jednoduše a jednoznačně určují velikost a tvar čísla. Jsou to následující typy:

Tab. 1. Typy určující velikost a tvar čísla

Název typu	Hodnota	Počet bitů	Tvar	Minimum	Maximum
tBYTE	\$01	8	neznaménkový	0	255
tSBYTE	\$09	8	znaménkový	-128	127
tWORD	\$02	16	neznaménkový	0	65535
tSWORD	\$0A	16	znaménkový	-32768	32767
tLONG	\$04	32	neznaménkový	0	4294967295
tSLONG	\$0C	32	znaménkový	-2147483648	2147483647

Číslo tedy bude jednoznačně určeno hodnotou uloženou ve čtyřbytovém akumulátoru a jednobytovém typu. Jak je vidět, hodnota typu je složena z počtu bytů čísla a příznaku pro znaménkové číslo. To umožňuje jednoduše získat velikost čísla nebo zjistit jestli je znaménkové či naopak. Velikost čísla se dá zjistit tak, že se načte jeho typ a provede se logický součin s bitovou maskou \$07, výsledkem bude velikost čísla. Typ čísla však zabírá pouze 4 bity, takže horní 4 bity bytů *ACCAT* a *ACCBT* jsou volné pro další příznaky, které budou při výpočtech potřeba.

Zmíněnou bitovou masku knihovna používá velmi často, proto je vhodné ji zanést do programu jako konstantu. Knihovna používá dvě bitové masky, *mSIZE*, jak již bylo zmíněno dříve, je používána pro získání rozsahu čísla z jeho typu. Druhá maska *mSIGN* je používána pro získání hodnoty nejvyššího 7. bitu v bytu, tedy nejčastěji pro zjištění zápornosti čísla.

Kromě akumulátorů *ACCA* a *ACCB* a jejich typů, se deklarují další pomocné akumulátory, které vyžadují algoritmy násobení a zejména dělení. Tento krok vede samozřejmě ke zrychlení výpočtu na úkor zabrané paměti. Celkem je zde tedy 23B rezervovaných pro data (včetně akumulátorů *ACCA* a *ACCB*).

V následujících podkapitolách práce rozvede jednotlivé funkce a jejich využití. Pro lepší vysvětlení tématu se práce nezabývá popisem zdrojových kódů. Zaměřuje se přímo na jednotlivé algoritmy aritmetických operací implementovaných v jazyce ASM znázorněných ve schématech (dále jen Obr.) a jednoduchých nečíslovaných tabulkách.

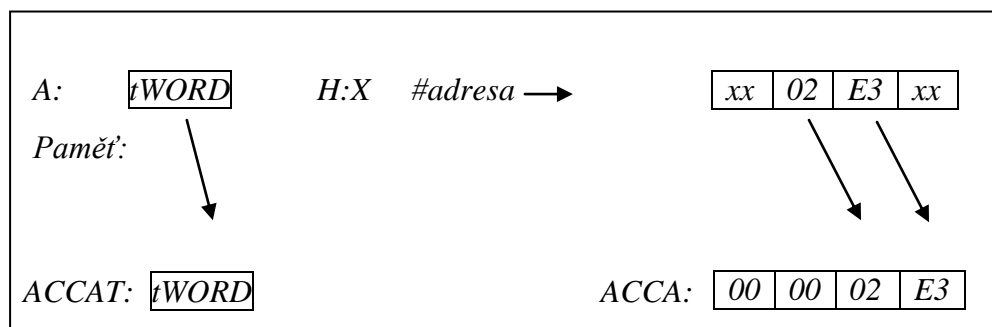
3.1 Funkce pro načítání dat do akumulátorů

3.1.1 Funkce *GETA* a *GETB*

Vstupy:

- Registr *A* – typ načítaného čísla.
- Registr *H:X* – 16-ti bitová adresa nejvyššího bytu načítaného čísla.

Tato funkce plní daty akumulátor *ACCA*. Typ načítaného čísla je předán v registru *A* a adresa počátku dat je předána v registru *H:X*. Funkce uloží typ čísla do *ACCAT*, zjistí si počet bytů, které má přenést. Nalezne poslední byte dat a přenesse ho na poslední byte akumulátoru *ACCA*. Pak postupuje zpět a kopíruje byte po bytu. Jakmile přenesse všechny byty, vynuluje zbylé byty akumulátoru *ACCA*.



Obr. 3. Schéma funkcí GETA a GETB

Funkce GETB je naprosto stejná, ale načítá do akumulátoru ACCB.

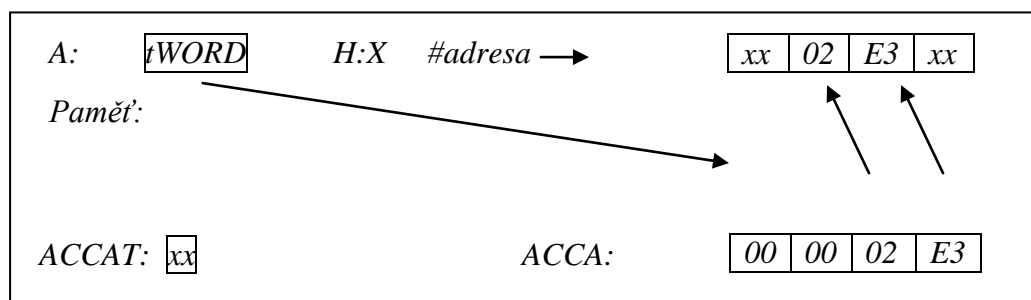
3.2 Funkce pro ukládání dat z akumulátorů

3.2.1 Funkce PUTA a PUTB

Vstupy:

- Registr A – typ ukládaného čísla.
- Registr H:X – 16-ti bitová adresa nejvyššího bytu v paměti, kam se má zapisovat.

Funkce *PUTA* přijímá v registrech *H:X* adresu, na kterou má data z akumulátoru ukládat. V registru *A* přijímá externí typ čísla. Pokud je *A* nulové, použije se typ *ACCAT*, jinak se použije typ specifikovaný v *A*. Funkce na zadanou adresu přenese nejvyšší byte akumulátoru *ACCA*, poté pokračuje byte po bytu dále, dokud nepřenese všechny byty čísla. Z toho vyplývá, že před ukládáním dat, je potřeba mít na určené adrese vyhrazen dostatečný počet bytů. To může být někdy problém a právě z tohoto důvodu existuje možnost zadat externí typ. Pokud by číslo mělo 2B, ale vyhrazen by byl pouze byte, dojde k přepsání i následujícího bytu, což většinou bude vést k chybě programu. Jestliže se zadá externí typ jako byte, bude přenesen pouze nejnižší byte čísla a k chybě nedojde. Došlo-li při výpočtu k přetečení nebo je-li číslo větší než externí typ, dojde k určité ztrátě dat. Proto nejlepší možností je vyhradit pro číslo celé 4B a také zadat externě typ long. Výsledkem bude dokonalá kopie akumulátoru *ACCA*.

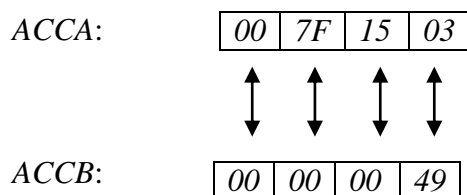


Obr. 4. Schéma funkcí PUTA a PUTB

3.3 Funkce pro prohození akumulátoru a funkce pro porovnávání

3.3.1 Funkce SWAB

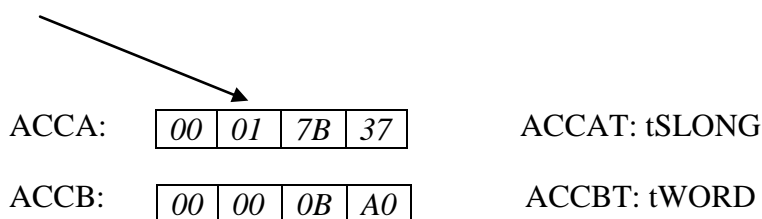
Funkce *SWAB* prohodí obsah jednotlivých akumulátorů a jejich typů. Prohodí mezi sebou všechny 4 byty akumulátorů, je to rychlejší než dodatečné testy, kolik bytů skutečně prohodit.



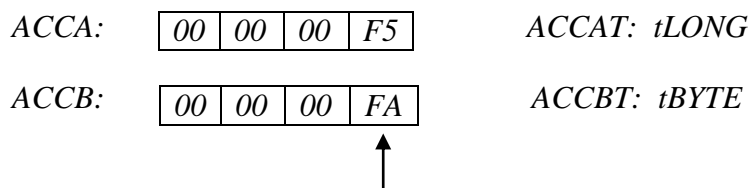
3.3.2 Funkce CMPAB

Tato funkce slouží pro porovnání čísel v akumulátorech *ACCA* a *ACCB*. Jejím výstupem je změna příznaků v registru *CCR* obdobným způsobem jako to dělá instrukce *CMP*. Funkce *CMPAB* musí nejdříve rozlišit, jaké typy čísel porovnává, jestli záporná nebo kladná. Prověří byty většího čísla, jestli jsou 0 u kladných nebo \$FF u záporných. Pokud ne, může okamžitě rozhodnout, které číslo je větší:

Tento byte není 0, což znamená, že *ACCA* je větší než *ACCB*.



Pokud nelze rozhodnout pomocí přesahujících bytů nebo mají čísla shodný rozsah, porovnává se dále byte po byte, dokud se neporovnají všechny byty nebo nedojde k rozhodnutí.



Při porovnání posledních bytů, dojde k rozhodnutí, číslo *ACCB* je větší než číslo *ACC*.

Jestliže stále není možné rozhodnout, které číslo je větší, jsou čísla stejná.

Tab. 2. Hodnoty CCR pro jednotlivé případy

Případ	Registr CCR		
	Bit N	Bit Z	Bit C
$ACCA = ACCB$	0	1	0
$ACCA > ACCB$	1	0	1
$ACCA < ACCB$	0	0	0

3.4 Aritmetické funkce

3.4.1 Funkce ADDAB

Vstupy:

- $ACCA$ – sčítanec,
- $ACCB$ – sčítanec,
- $ACCAT$ – typ sčítance $ACCA$,
- $ACCBT$ – typ sčítance $ACCB$.

Výstupy:

- $ACCA$ – výsledek,
- $ACCAT$ – typ výsledku.

Při sčítání čísel v akumulátorech je nutno nejdříve rozlišit jeden ze čtyř známých případů (viz teoretická část). To se provede kontrolou typu, pokud mají obě čísla neznaménkový typ, pak se sečtou neznaménkové čísla k tomu určenou podprocedurou. Je-li jedno z čísel znaménkové, prozkoumá se, zda-li jsou čísla kladná, záporná či různá. Pro každý z těchto případů je navržena jedna podprocedura. Celkem tedy 4 samostatné podprocedury (pro každý případ jedna).

3.4.1.1 Sčítání neznaménkových čísel *ADDAB_UNSGN*

U sčítání neznaménkových čísel práce pro výrazné zrychlení sečítá všechny byty akumulátorů $ACCA$ a $ACCB$. Jde sice o jistou redundanci při sčítání malých čísel (např. obě čísla typu *tBYTE*), ale ta je vyvážena podstatným zrychlením v případě větších čísel a absencí zbytečných testů, kolik bytů je již sečteno a kolik teprve čeká na sečtení.

Další částí této podprocedury je zjištění, kolik bytů je skutečně obsazeno a v případě potřeby zvětšit typ výsledku. Sčítá-li se číslo typu *tBYTE* v $ACCA$ a číslo typu *tLONG*

v *ACCB*, může mít výsledek prakticky jakoukoli velikost (1 až 4 byty). Existují tři spolehlivá řešení:

1. Je-li typ *ACCBT* rozsahem větší než *ACCAT*, překopíruje se *ACCBT* do *ACCAT*, v případě přetečení při výpočtu se dodatečně zvětší rozsah.
2. Při sčítání každého bytu se kontroluje typ *ACCAT* a v případě potřeby se rozšíří – velmi pomalé řešení.
3. Po skončení sčítání se projdou byty od nejvyššího k nejnižšímu a podle toho, kdy se narazí na nenulový byte, se nastaví typ *ACCAT*, pokud je nový typ rozsahem větší než původní.

Podprocedura implementuje 3. způsob, který je poměrně rychlý. Tímto je tedy zabezpečeno, že po každém sčítání neznaménkových čísel bude mít výsledek správný typ.

3.4.1.2 Sčítání kladných znaménkových čísel *ADDAB_0NEG*

U kladných znaménkových čísel je třeba si dávat pozor na přetečení dat do znaménkového bitu. Proto se nejdříve zjistí rozsahy obou čísel, následně se sčítá potřebný počet bytů. V případě, že by došlo k přetečení nad rámec rozsahu, se tento rozsah zvětší. Pokud dojde k přetečení jedničky do znaménkového bytu, který je v tomto případě vždy 0, opět je nutné rozsah zvětšit:

Číslo 86 je vyjádřeno jako:

0 1 0 1 0 1 1 0 ← hodnota čísla, 86
 ↑
 Znaménkový bit, +

Když k němu se přičte číslo 53, které je vyjádřeno takto:

0 0 1 1 0 1 0 1

Výpočet:

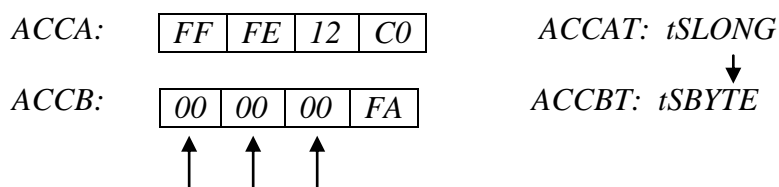
0 1 0 1 0 1 1 0
 0 0 1 1 0 1 0 1
 1 0 0 0 1 0 1 1

Zde došlo právě k přetečení do znaménkového bitu, což se musí ošetřit zvětšením rozsahu, jinak by výsledek byl -117, a ne správných 139.

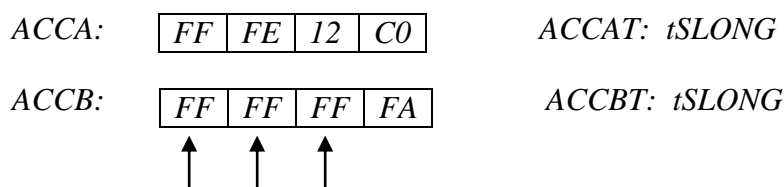
1 0 1 0 1 0 0 1 → 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1
 1 byte (tSBYTE) 2 byty (tSWORD)

3.4.1.3 Sčítání záporných znaménkových čísel *ADDAB_2NEG*

Jak bylo popsáno v teoretické části, je nutné záporná čísla převést na stejný typ. To lze udělat velmi jednoduchým způsobem. Všechny byty, které číslu chybí k doplnění do rozsahu většího čísla, nahradíme jejich jedničkovým doplňkem, tedy z nuly se stanou jedničky a znaménkový bit se přesune:



Zde jsou byty nulové, změníme je tedy jedničkovým doplňkem.



Číslo je nyní správně rozšířeno na typ *tSLONG*

Jakmile jsou obě čísla srovnána na stejný rozsah, je možné je běžným způsobem sečíst. Opět jejich sečtením může dojít k přetečení, takže je nutno hlídat a případně ošetřit zvětšením rozsahu. Rozsah ovšem není potřeba zvětšovat, pokud se na posledním bitu nejvyššího bytu nachází jednička. V takovém případě je možné nebo spíše vhodné přetečení ignorovat:

$$\begin{array}{r}
 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0 \\
 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1 \\
 \hline
 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 1
 \end{array}$$

Zde se přenos ignoruje, nezmění to hodnotu výsledku.

3.4.1.4 Sčítání různých čísel, z nichž jedno je záporné *ADDAB_ANEG* a *ADDAB_BNEG*

Zde se musí opět srovnat záporné číslo na stejný rozsah jako číslo kladné. Ovšem je nutné dát pozor na neznaménková čísla, která se znaménkovými nelze vždy jednoduše sečíst. Je-li jedno z čísel neznaménkové, a jeho nejvyšší bit je roven 1, pak tomuto číslu musí být zvětšen rozsah. Teprve pak můžeme záporné číslo srovnat s jeho rozsahem a čísla sečíst. Typ výsledku bude znaménkový a to i v případě, že součet čísel zůstane kladný.

3.4.2 Funkce SUBAB

Vstupy:

- ACCA – menšenec,
- ACCB – menšitel,
- ACCAT – typ menšence ACCA,
- ACCBT – typ menšitele ACCB.

Výstupy:

- ACCA – výsledek,
- ACCAT – typ výsledku.

Funkce pro odečtení nahrazuje odčítání inverzí znaménka u čísla *ACCB* a následným sečtením s číslem *ACCA*:

<i>ACCA</i> :	<table border="1" style="display: inline-table;"><tr><td>FF</td><td>FE</td><td>12</td><td>C0</td></tr></table>	FF	FE	12	C0	<i>ACCAT</i> : <i>tSLONG</i>
FF	FE	12	C0			
<i>ACCB</i> :	<table border="1" style="display: inline-table;"><tr><td>00</td><td>00</td><td>00</td><td>C3</td></tr></table>	00	00	00	C3	<i>ACCBT</i> : <i>tBYTE</i>
00	00	00	C3			

Inverze znaménka ve tvaru dvojkového doplňku se provede záměnou všech bytů čísla za jejich jedničkové doplňky a přičtením jedničky k nejnižšímu bytu. Pokud má neznaménkové číslo v *ACCB* nejvyšší bit roven 1, musí se jeho rozsah zvětšit už při inverzi:

<i>ACCA</i> :	<table border="1" style="display: inline-table;"><tr><td>00</td><td>07</td><td>64</td><td>2F</td></tr></table>	00	07	64	2F	<i>ACCAT</i> : <i>tSLONG</i>
00	07	64	2F			
<i>ACCB</i> :	<table border="1" style="display: inline-table;"><tr><td>00</td><td>00</td><td>FF</td><td>3D</td></tr></table>	00	00	FF	3D	<i>ACCBT</i> : <i>tWORD</i>
00	00	FF	3D			

Následně se obě čísla sečtou, k tomu se využije již hotová funkce pro sčítání čísel, která si všechny problémy při sčítání pohlídá a ošetří.

3.4.3 Funkce MULAB

Vstupy:

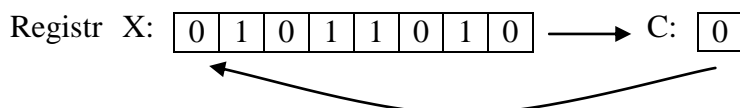
- ACCA – činitel,
- ACCB – činitel,
- ACCAT – typ činitele ACCA,
- ACCBT – typ činitele ACCB.

Výstupy:

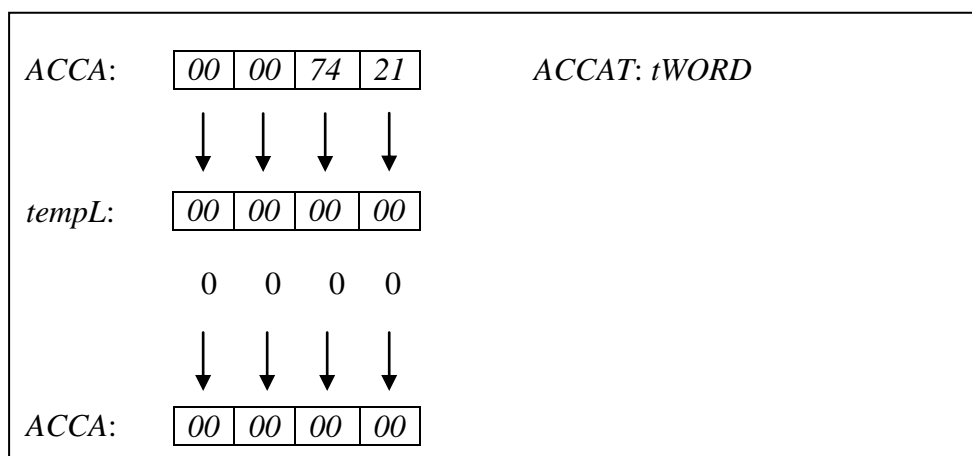
- ACCA – výsledek,

- ACCAT – typ výsledku.

Funkce *MULAB* násobí čísla v akumulátorech a výsledek uloží do akumulátoru *ACCA*. Násobení platí pro kladná čísla, takže je potřeba záporná čísla převést na kladná pomocí znaménkové inverze. Pro násobení je využit algoritmus využívající bitových posunů, zde lze s výhodou využít instrukce pro rotaci bytu přes *Carry bit*. Zejména vhodná je instrukce *RORX*, která rotuje registr *X* přes *Carry*. Princip je následující:

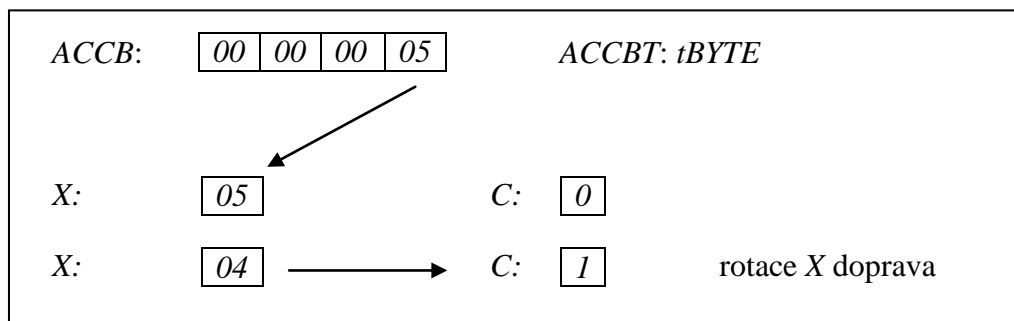


Při násobení je nejdříve číslo z *ACCA* zkopírováno do pomocného akumulátoru *tempL* a akumulátor *ACCA* je vynulován.



Obr. 5. Schéma přenosu čísla do *tempL*

Nyní jsou postupně načítány jednotlivé byty čísla *ACCB* do registru *X*, který je rotován přes *Carry*.



Obr. 6. Schéma algoritmu násobení pomocí rotací čísla *ACCB*

Nyní plně vyplývá výhoda rotování přes *Carry*, použijeme zde instrukci *BCC*, která skočí na návěští, pokud je $C = 0$. V případě, že je $C = 1$, pokračuje program dále přičtením akumulátoru *tempL* k *ACCA*. Je-li $C = 0$, je toto přičtení přeskočeno.

$$\begin{array}{rcl}
 \text{ACCA:} & & \boxed{00} \boxed{00} \boxed{00} \boxed{00} \\
 & + & \begin{array}{c} \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \\ \boxed{00} \boxed{00} \boxed{74} \boxed{21} \end{array} \\
 \text{tempL:} & &
 \end{array}$$

Posuneme akumulátor *tempL* doleva, použitím instrukce *ROL*:

$$\text{tempL:} \quad \boxed{00} \boxed{00} \boxed{E8} \boxed{42} \quad \leftarrow 0$$

Toto je celý jeden cyklus algoritmu. Opakuje se tak dlouho, dokud má číslo *ACCB* další bity. Jako výhodnější, jednodušší a rychlejší pro malé rozsahy se ukázalo nastavit počet opakování na počet bitů rozsahu.

3.4.4 Funkce DIVAB

Vstupy:

- *ACCA* – dělenec,
- *ACCB* – dělitel,
- *ACCAT* – typ dělence *ACCA*,
- *ACCBT* – typ dělitele *ACCB*.

Výstupy:

- *ACCA* – výsledek,
- *ACCAT* – typ výsledku,
- *ACCB* – zbytek,
- *ACCBT* – typ zbytku.

Funkce *DIVAB* dělí číslo v *ACCA* číslem v *ACCB*, výsledek bude uložen v *ACCA* a zbytek po dělení v *ACCB*. Protože algoritmus pro dělení čísel pracuje jen s neznaménkovými čísly, respektive znaménkovými kladnými, musí být záporná čísla převedena na kladná pomocí znaménkové inverze. Po převedení je nastaven příznak (7. bit v *ACCAT* nebo *ACCBT*). Pokud jsou obě čísla záporná, vynulují se příznaky a výsledek bude kladný. Pokud je jen jedno z čísel záporné, vynulují se příznaky na sedmých bitech a nastaví se příznak na 6. bitu *ACCAT*. Tento bit se na konci dělení kontroluje, a pokud je roven jedničce, výsledek se převede na záporné číslo.

Pro algoritmus dělení je důležitý počet bitů obou čísel. Každý typ má pevně určen počet bitů, tudíž by bylo možno vzít právě toto číslo. Avšak to může velmi často vést k nadměrně redundantnímu výpočtu. Bylo by například číslo 101b typu tLONG, byla by jeho délka určena na 32 bitů, což by vedlo ke zdoluhavému výpočtu plného nulových výsledků. Proto je vhodnější zjistit skutečnou délku čísla, počítanou od prvního výskytu jedničky.

Prvním krokem pro zjištění skutečné velikosti je vyhledání prvního bytu od nejvyššího k nejnižšímu, který není nulový. Jakmile je nalezen první nenulový byte, všechny nižší byty jsou považovány za součást čísla. V nalezeném bytu je nyní nutné zjistit skutečný počet bitů, k tomu lze velmi výhodně využít instrukci pro rotaci doleva přes Carry – ROL. Celý byte se rotuje doleva tak dlouho, dokud se v Carry nenachází jednička nebo bylo dosaženo osmi rotací (pro zamezení nekonečné smyčky). Zjištěný počet bitů se přičte k osminásobku počtu nižších bytů a tím získáme celkový počet bitů čísla. Celá tato operace má obrovskou výhodu v tom, že pomůže jednoduše vyčlenit čísla, pro které nemá algoritmus dělení smysl. Není-li nalezen žádný nenulový byte, pak je číslo rovno nule a dělení nulou nelze provést. Je-li velikost čísla jen dva bity, pak je možné snadno zjistit, jedná-li se o jedničku, dvojku nebo trojku. Dělení jedničkou dává výsledek rovný dělenci. Dělení dvojkou dává výsledek o jeden bit posunutý doprava a zbytek je právě bit, který byl posunut ven. Pro tyto dvě čísla je tedy vhodné dělení optimalizovat a celý algoritmus neprovádět:

$$127 \div 2 = 63 \text{ a zbytek je } 1.$$

Binárně:

$$01111111 \div 10$$

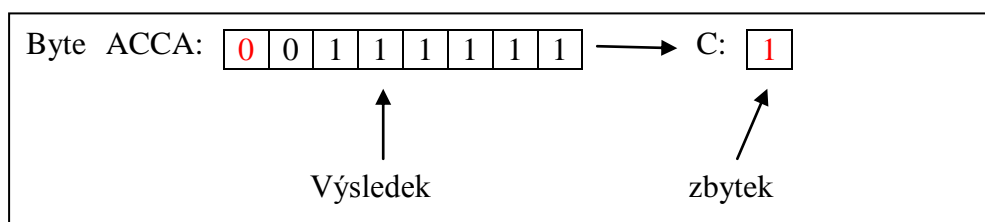
Byte ACCA:

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

 C:

0

Byte čísla ACCA rotujeme přes C pomocí ROR:

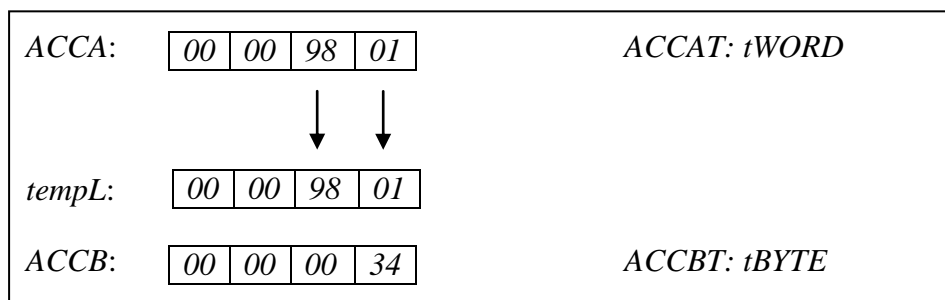


Obr. 7. Schéma rotace čísla přes C

Před začátkem samotného dělení se provede porovnání dělence a dělitele a to z důvodu vyčlenění dalších případů, kdy je výsledek znám dopředu. Jde o případ, kdy se dělí číslo

stejným číslem, takže výsledek je roven jedné. A všechny případy, kdy je dělitel větší než dělenec, pak je výsledek vždy nula a zbytek je roven dělenci. Pokud dojde k některému z vyjmenovaných případů, nemá smysl začínat složitý algoritmus dělení, ale rovnou nastavit výsledek.

Nyní k samotnému dělení. Číslo *ACCA* se překopíruje do pomocného akumulátoru, ve kterém se pomocí rotace vyrovná k levému okraji:



Obr. 8. Schéma vytvoření kopie *ACCA* při dělení

Rotací se číslo srovná k levému okraji:

tempL:

98	01	00	00
----	----	----	----

Nyní je potřeba přenést několik prvních bitů z pomocného akumulátoru do akumulátoru *ACCA*. K tomu je nepřekonatelně výhodná instrukce rotace přes *C*. Po každé rotaci akumulátoru *tempL* doleva zůstane v *C* jeho nejvyšší bit. Tento bit se rotuje opět směrem doleva do *ACCA*. Celý proces je nutno opakovat tolikrát, kolik má druhé číslo bitů. Jakmile je číslo přeneseno, vytvoří se kopie čísla *ACCA*, odečte se *ACCB* od *ACCA* a kontroluje se, jestli došlo k přetečení (*C* = 1).

Došlo-li k přetečení, pak číslo *ACCA* obsahuje číslo *ACCB* 0krát. Takže je vynulován bit *C* a rotován do výsledku (momentálně v pomocném akumulátoru *divR*). Do *ACCA* je vrácena jeho původní hodnota.

Nedošlo-li k přetečení, pak *ACCA* obsahuje číslo *ACCB* 1krát a proto je nastaven bit *C* na 1 a rotován do výsledku. V *ACCA* zůstává rozdíl a kopie původního stavu se nevyužije.

Následně je rotován další bit z akumulátoru *tempL* do *ACCA*, vytvoří se opět kopie *ACCA* a pak se odečte *ACCB* od *ACCA* atd. Postup se opakuje tak dlouho dokud není přeneseno do *ACCA* celé číslo z *tempL*, zde se využije počet bitů čísla *ACCA* získaný na začátku.

Po skončení výpočtu se zbytek nachází v *ACCA*, proto se přenese do *ACCB* a výsledek, který byl doposud v akumulátoru *divR*, se přenese do *ACCA*. Pokud je nutno obrátit znaménko výsledku (nastaven 6. bit *ACCAT*), obrátí se známou znaménkovou inverzí.

3.5 Doplnkové funkce

3.5.1 Funkce STR2INT

Vstupy:

- *A* – délka řetězce v bytech,
- *H:X* – adresa prvního bytu řetězce.

Výstup:

- *ACCA* – převedené číslo.

Funkce *STR2INT* převádí řetězec začínající na adrese udané v *H:X* o počtu bytů udaném v registru *A*. Převod skončí předčasně, pokud funkce narazí na znak, který není číslo v ASCII tvaru. Je-li na prvním místě znaménko -, je toto číslo konvertováno na záporné.

3.5.2 Funkce INT2STR

Vstupy:

- *ACCA* – číslo, které má být převedeno
- *H:X* – adresa prvního bytu, kam má být řetězec uložen.

Výstup:

- *ACCA* – převedené číslo.

Funkce převádí číslo uložené v *ACCA* na řetězec ASCII hodnot. Využívá k tomu dělení 10, kdy je zbytek roven číslici na daném místě. Následně jsou číslice uloženy do paměti na místo zadané v registru *H:X*.

3.5.3 Funkce GETSA a GETSB

GETSA vrací v registru *A* počet bytů čísla v *ACCA*, obdobně *GETSB* vrací počet bytů čísla v *ACCB*.

4 SROVNÁNÍ S FUNKCEMI JAZYKA C

Každá z vytvořených funkcí má pro různá čísla různou dobu výpočtu. Proto je vhodné vytvořit srovnání, ve kterém je jasně vidět závislost doby výpočtu na typu čísel.

Pro porovnání aritmetických operací byl vytvořen jednoduchý program v jazyce C, který provede aritmetickou operaci se dvěma čísly zadaného typu. A zde nastává problém. Jestliže byly proměnné definovány (nastaveny na konkrétní hodnotu), pak jsou všechny aritmetické operace vypočítány kompilátorem a do zdrojového kódu je dosazen přímo výsledek operace. Tudíž při vykonávání kódu nedochází k žádným aritmetickým operacím!

Proto je nutné hodnoty proměnným přiřadit až za běhu programu. Toho lze nejjednodušší docílit prostým generátorem náhodných čísel. Takto upravený program již opravdu prováděl aritmetické operace za chodu, ovšem stále zde zůstává určitá výhoda kompilovaného programu jazyka C. Onou výhodou je silná typovost jazyka, což znamená, že každá proměnná má deklarovaný typ a podle tohoto typu se s ní pracuje.

Matematická knihovna v Assembleru ovšem tuto typovost nemá. Zmíněné akumulátory ACCA a ACCB mohou obsahovat čísla různých typů a vytvořené funkce jsou více či méně jednotné pro různé typy. Studium výsledného kódu programu, napsaném v jazyce C, lze dojít ke zjištění, že pro každou kombinaci typů proměnných je vytvořen optimální kód, kterým je aritmetická operace nahrazena, případně je tento kód volán jako podprogram.

Například sčítání dvou proměnných typu Integer (velikost 2B) je ve výsledném kódu nahrazeno přímým sečtením čtyř odpovídajících bytů. Dojde-li k přetečení, pak záleží na typu proměnné, kam je výsledek sčítání přiřazen. Je-li takéž typu Integer, přetečení je ignorováno, což přináší další zrychlení, ale také ztrátu informace. K rozhodnutí jaký algoritmus pro konkrétní aritmetickou operaci použít a jak zacházet s výsledkem dochází při kompilaci programu a ne za jeho běhu. Tato rozhodnutí ovšem tvoří podstatnou část vytvořených knihovních funkcí a nelze ji zanedbat. Z toho vyplývá, že funkce knihovny a jazyka C nelze přímo srovnat, ovšem pro představu o rychlosti jednotlivých funkcí je srovnání provedeno. Nelze však činit směřodátné závěry o rychlosti či pomalosti jednotlivých funkcí.

Operandy

Na začátku srovnání s jazykem C je nutné uvést základní pojmy a zkratky, které se budou vyskytovat v následujících kapitolách jak v tabulkách, tak i v grafech. Jde zejména o zkratky použitých typů operandů.

Tab. 3. Odpovídající typy operandů jazyka C a knihovny v jazyce ASM

Typ v matematické knihovně	Typ v jazyce C	Použitá zkratka
tBYTE	unsigned char	UB
tSBYTE	signed char	SB
tWORD	unsigned	UINT
tSWORD	int	INT
tLONG	unsigned long	ULONG
tSLONG	long	LONG

4.1 Sčítání

Pro porovnání rychlosti sčítání je použit jednoduchý program v jazyce C, který vygeneruje dvě čísla zadaného typu a pak čísla sečte. Při vykonávání programu byl měřen počet cyklů před sečtením a po něm. Rozdíl dává počet cyklů, které samotné sečtení zabralo. Naměřeny byly následující hodnoty:

Tab. 4. Sčítání v jazyce C

Sčítání v C				
Typ	Počet cyklů	Hodnoty	Výsledek	Typ výsledku
UB + UB	24	232 + 44	276	UB
UB + SB	35	58 + (-53)	6	SB
SB + SB	46	(-113) + (-102)	-215	SB
UINT + UB	25	18266 + 186	18452	UINT
UINT + SB	36	11689 + (-21)	11668	UINT
INT + SB	37	(-4956) + (-105)	-5061	INT
UINT + UINT	29	15676 + 3304	18980	UINT
UINT + INT	30	7494 + (-9854)	-2360	INT
INT + INT	30	(-1969) + (-15745)	-17714	INT
ULONG + UB	246	67969 + 200	68169	ULONG
ULONG + SB	263	74227 + (-111)	74116	LONG

Typ	Počet cyklů	Hodnoty	Výsledek	Typ výsledku
LONG + SB	263	$(-74453) + (-82)$	-74535	LONG
ULONG + UINT	29	$69942 + 14715$	84657	ULONG
ULONG + INT	265	$89287 + (-22035)$	67252	LONG
LONG + INT	265	$(-93517) + (-9087)$	-102604	LONG
ULONG + ULONG	228	$93236 + 70937$	164173	ULONG
ULONG + LONG	228	$89345 + (-79842)$	9503	LONG
LONG + LONG	228	$(-98370) + (-79973)$	-178343	LONG

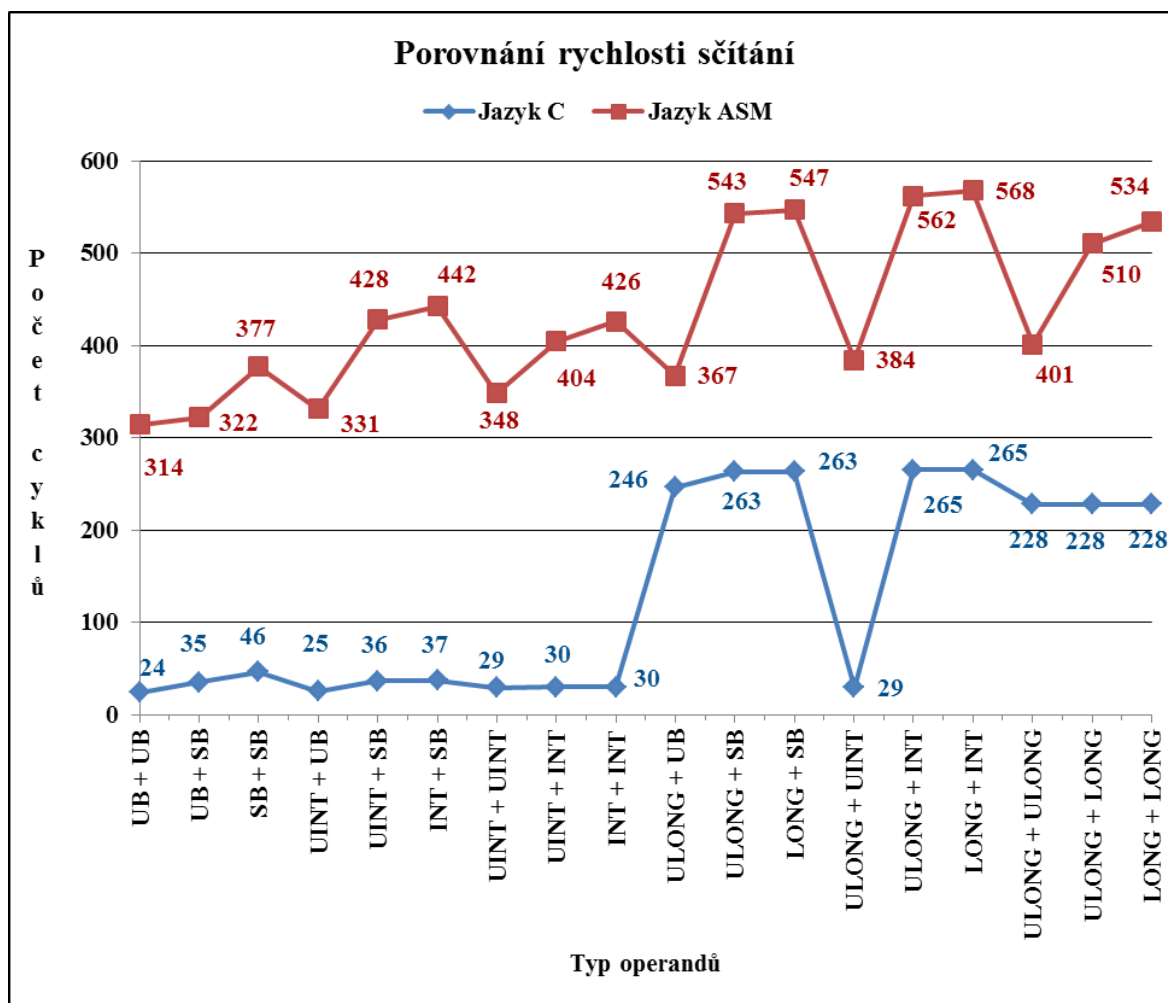
Náhodná čísla byla zaznamenána a použita pro stejná měření u funkce ADDAB vytvořené knihovny. Do registru A je načten typ operandu, do registru H:X je načtena adresa uloženého čísla, pak je volána funkce GETA, která naplní akumulátor ACCA. Době běhu této funkce je součástí celkového výpočtu, stejně tak doba běhu GETB pro naplnění akumulátoru ACCB a PUTA pro získání výsledku z akumulátoru ACCA.

Tab. 5. Sčítání v jazyce ASM

Sčítání v ASM							
Typ	Počet cyklů					Hodnoty	Výsledek
	GETA	GETB	ADDAB	PUTA	Celkem		
UB + UB	56	56	141	61	314	$232 + 44$	276
UB + SB	56	56	169	41	322	$58 + (-53)$	5
SB + SB	56	56	204	61	377	$(-113) + (-102)$	-215
UINT + UB	73	56	141	61	331	$18266 + 186$	18452
UINT + SB	73	56	238	61	428	$11684 + (-21)$	11668
INT + SB	73	56	252	61	442	$(-4956) + (-105)$	-5061
UINT + UINT	73	73	141	61	348	$15676 + 3304$	18980
UINT + INT	73	73	197	61	404	$7494 + (-9854)$	-2360
INT + INT	73	73	219	61	426	$(-1969) + (-15745)$	-17714
ULONG + UB	90	56	127	94	367	$67969 + 200$	68169
ULONG + SB	90	56	303	94	543	$74227 + (-111)$	74116
LONG + SB	90	56	307	94	547	$(-74453) + (-82)$	-74535
ULONG + UINT	90	73	127	94	384	$69942 + 14715$	84657
ULONG + INT	90	73	305	94	562	$89287 + (-22035)$	67252

Typ	Počet cyklů					Hodnoty	Výsledek
	GETA	GETB	ADDAB	PUTA	Celkem		
LONG + INT	90	73	311	94	568	(+93517) + (-9087)	-102604
ULONG + ULONG	90	90	127	94	401	93236 + 70937	164173
ULONG + LONG	90	90	236	94	510	89345 + (-79842)	9503
LONG + LONG	90	90	260	94	534	(-98370) + (-79973)	-178343

Hodnoty cyklů z obou výše uvedených tabulek jsou zobrazeny v následujícím přehledném grafu. Je vidět, že funkce ADDAB je znatelně pomalejší než funkce sčítání v jazyce C. Pokud by však byla do grafu zanesena i doba, která je operaci sčítání věnována v době kompilace C programu, pak by byly rozdíly v počtu cyklů téměř minimální. V grafu je také patrné, že sčítání neznaménkových operandů je podstatně rychlejší než sčítání znaménkových operandů.



Graf 1. Porovnání rychlosti sčítání dle jazyků

4.2 Odčítání

Stejně jako u sčítání je pro srovnání algoritmu odčítání vytvořen primitivní program, který vygeneruje dvě náhodná čísla a odečte jedno od druhého. Měřen je počet cyklů před a po provedení operace odčítání. Hodnoty jsou uvedeny níže v tabulce:

Tab. 6. Odčítání v jazyce C

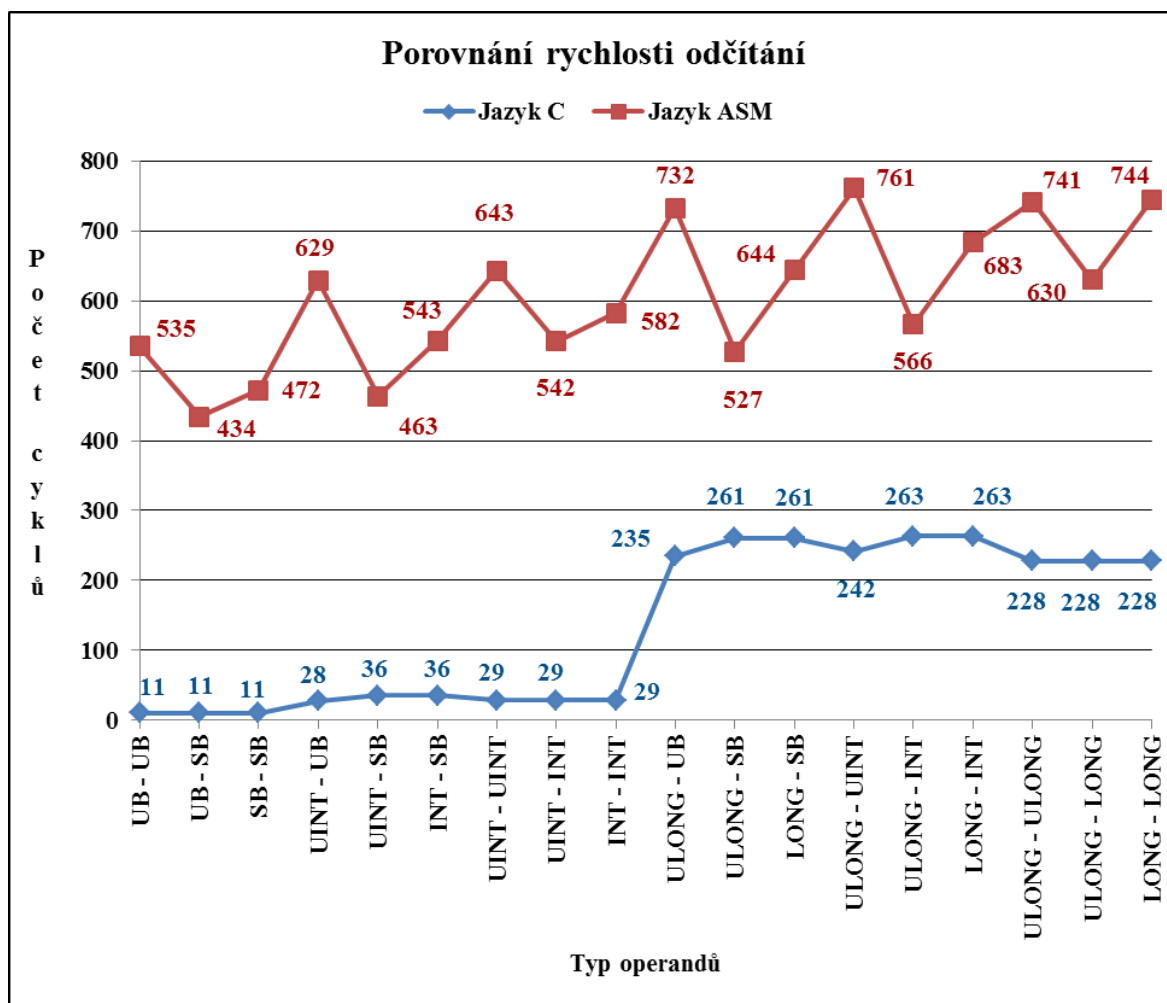
Odčítání v C				
Typ	Počet cyklů	Hodnoty	Výsledek	Typ výsledku
UB – UB	11	235 – 120	115	UB
UB – SB	11	61 – (-123)	-72	SB
SB – SB	11	(-85) – (-73)	-12	SB
UINT – UB	28	18492 – 40	18452	UINT
UINT – SB	36	22722 – (-81)	22803	UINT
INT – SB	36	(-2096) – (-41)	-2055	INT
UINT – UINT	29	4788 – 12126	58198	UINT
UINT – INT	29	22441 – (-23625)	-19470	INT
INT – INT	29	(-22287) – (-21035)	-1252	INT
ULONG – UB	235	91633 – 76	91557	ULONG
ULONG – SB	261	75672 – (-124)	75796	LONG
LONG – SB	261	(-76518) – (-76)	-76442	LONG
ULONG – UINT	242	80748 – 18996	61752	ULONG
ULONG – INT	263	90506 – (-17886)	108392	LONG
LONG – INT	263	(-91352) – (-15296)	-76056	LONG
ULONG – ULONG	228	94736 – 72437	22299	ULONG
ULONG – LONG	228	95582 – (-69847)	165429	LONG
LONG – LONG	228	(-81313) – (-79757)	-1556	LONG

Náhodné hodnoty jsou zapsány a použity pro srovnání rychlosti funkce SUBAB matematické knihovny. Čísla jsou uložena do paměti, následně načtena do akumulátorů pomocí funkcí GETA a GETB. Po odečtení funkcí SUBAB je výsledek získán z akumulátoru ACCA funkcí PUTA viz Tab. 7.

Tab. 7. Odčítání v jazyce ASM

Odčítání v ASM							
Typ	Počet cyklů					Hodnoty	Výsledek
	GETA	GETB	ADDAB	PUTA	Celkem		
UB – UB	56	56	382	41	535	235 – 120	115
UB – SB	56	56	261	61	434	61 – (-123)	184
SB – SB	56	56	319	41	472	(-85) – (-73)	-12
UINT – UB	73	56	439	61	629	18492 – 40	18452
UINT – SB	73	56	273	61	463	22722 – (-81)	22803
INT – SB	73	56	353	61	543	(-2096) – (-41)	-2055
UINT – UINT	73	73	436	61	643	4788 – 12126	-7338
UINT – INT	73	73	302	94	542	22441 – (-23625)	46066
INT – INT	73	73	375	61	582	(-22287) – (-21035)	-1252
ULONG – UB	90	56	492	94	732	91633 – 76	91557
ULONG – SB	90	56	287	94	527	75672 – (-124)	75796
LONG – SB	90	56	404	94	644	(-76518) – (-76)	-76394
ULONG – UINT	90	73	504	94	761	80748 – 18996	61752
ULONG – INT	90	73	309	94	566	90506 – (-17886)	108392
LONG – INT	90	73	426	94	683	(-91352) – (-15296)	-76056
ULONG – ULONG	90	90	467	94	741	94736 – 72437	22299
ULONG – LONG	90	90	356	94	630	95582 – (-69847)	165429
LONG – LONG	90	90	470	94	744	(-81313) – (-79757)	-1556

Přímé srovnání počtu cyklů v grafu níže ukazuje, že ani zde vytvořená funkce matematické knihovny SUBAB nemůže rychlostí konkurovat předpřipravené optimalizované operaci jazyka C. U odčítání je vidět, že nejrychlejší kombinací operandů je odčítání záporného čísla od neznaménkového čísla. Tento rozdíl se totiž převádí na součet neznaménkového a kladného znaménkového čísla. Přesto je operace odčítání pomalejší než sčítání a to především kvůli provedení znaménkové inverze a také kvůli dvojité režii – nejprve se projde logickou částí odčítání, provede se inverze znaménka a pak se musí projít celou funkcí sčítání.



Graf 2. Porovnání rychlosti odčítání dle jazyků

4.3 Násobení

Stejným způsobem je porovnáváno násobení, avšak zde dochází k určitým problémům při násobení čísel v jazyce C. Násobení záporným číslem vedlo někdy ke špatným výsledkům. Z tohoto důvodu a také kvůli omezení maximálního rozsahu na čtyři byty je v tabulce níže menší počet výsledků. Jedná se nejspíše o problém s typy čísla (přetečení rozsahu Integer).

Tab. 8. Násobení v jazyce C

Násobení v C				
Typ	Počet cyklů	Hodnoty	Výsledek	Typ výsledku
UB * UB	20	192 * 247	47424	UINT
UB * SB	111	214 * (-82)	-17548	INT
SB * SB	67	(-71) * (-53)	3763	INT
UINT * UB	117	4946 * 117	54394	ULONG

Typ	Počet cyklů	Hodnoty	Výsledek	Typ výsledku
UINT * SB	135	22599 * (-57)	22577	LONG
INT * SB	211	(-17975) * (-44)	4468	LONG
UINT * UINT	113	31114 * 30248	39312	ULONG
UINT * INT	113	997 * (-21131)	34985	LONG
INT * INT	190	(-17804) * (-3452)	-13360	LONG
ULONG * UB	639	121999 * 8	975992	ULONG
ULONG * SB	656	123410 * (-126)	-15549660	LONG
LONG * SB	656	(-127640) * (-52)	6637280	LONG
ULONG * UINT	655	119800 * 5257	629788600	ULONG
ULONG * INT	658	108915 * (-3808)	-414748320	LONG
LONG * INT	658	(-113145) * (-44)	323481555	LONG

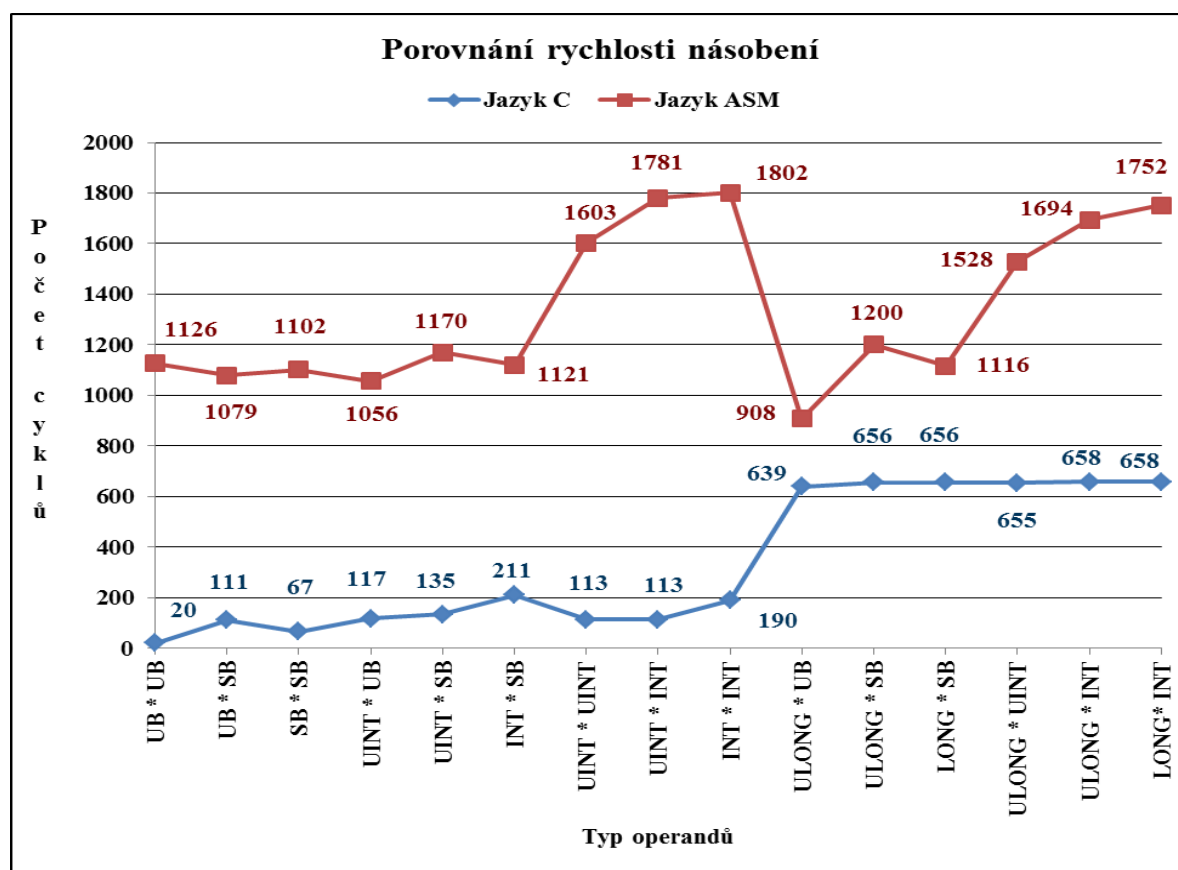
Pátý řádek v předchozí tabulce má špatný výsledek, jedná se o výše zmíněný problém. Zdá se, že algoritmus násobení jazyka C výsledek ořízl na velikost 2B, přestože výsledek byl velikosti 4B. Stejná čísla byla použita pro testování rychlosti funkce MULAB. Tato čísla byla naplněna do akumulátorů pomocí funkcí GETA a GETB. Po násobení funkcí MULAB byl získán výsledek funkcí PUTA.

Tab. 9. Násobení v jazyce ASM

Násobení v ASM							
Typ	Počet cyklů					Hodnoty	Výsledek
	GETA	GETB	ADDAB	PUTA	Celkem		
UB * UB	56	56	923	91	1126	192 * 247	47424
UB * SB	56	56	909	58	1079	214 * (-82)	-17548
SB * SB	56	56	932	58	1102	(-71) * (-53)	3763
UINT * UB	73	56	836	91	1056	4946 * 117	578682
UINT * SB	73	56	950	91	1170	22599 * (-57)	-1288143
INT * SB	73	56	901	91	1121	(-17975) * (-44)	790900
UINT * UINT	73	73	1366	91	1603	31114 * 30248	941136272
UINT * INT	73	73	1544	91	1781	997 * (-21131)	-21067607

Typ	Počet cyklů					Hodnoty	Výsledek
	GETA	GETB	ADDAB	PUTA	Cel- kem		
INT * INT	73	73	1565	91	1802	$(-17804) * (-3452)$	614594408
ULONG * UB	90	56	671	91	908	$121999 * 8$	975992
ULONG * SB	90	56	963	91	1200	$123410 * (-126)$	-15549660
LONG * SB	90	56	879	91	1116	$(-127640) * (-52)$	6637280
ULONG * UINT	90	73	1274	91	1528	$119800 * 5257$	629788600
ULONG * INT	90	73	1440	91	1694	$108915 * (-3808)$	-414748600
LONG * INT	90	73	1498	91	1752	$(-113145) * (-2859)$	323481555

Zajímavostí je řádek *ULONG * UB* u obou tabulek, kde je největší použité číslo násobeno nejmenším. U jazyka C tento výpočet trval dlouho (4. nejdelší čas), naopak v případě knihovny jazyka ASM trval výpočet dobu nejkratší (908 cyklů). Je to dáno tím, že počet opakování algoritmu funkce MULAB je přímo závislý na velikosti činitele ACCB.



Graf 3. Porovnání rychlosti násobení dle jazyků

4.4 Dělení

U dělení je opět nižší počet výsledků v tabulce, protože i zde docházelo k chybám. Jazyk C si neporadil s několika případy dělení, pak byl výsledek roven 0 a zbytek nesmyslné hodnotě. Jednalo se o dělení větších čísel menším záporným číslem. I tyto výpočty jsou uvedeny v tabulce:

Tab. 10. Dělení v jazyce C

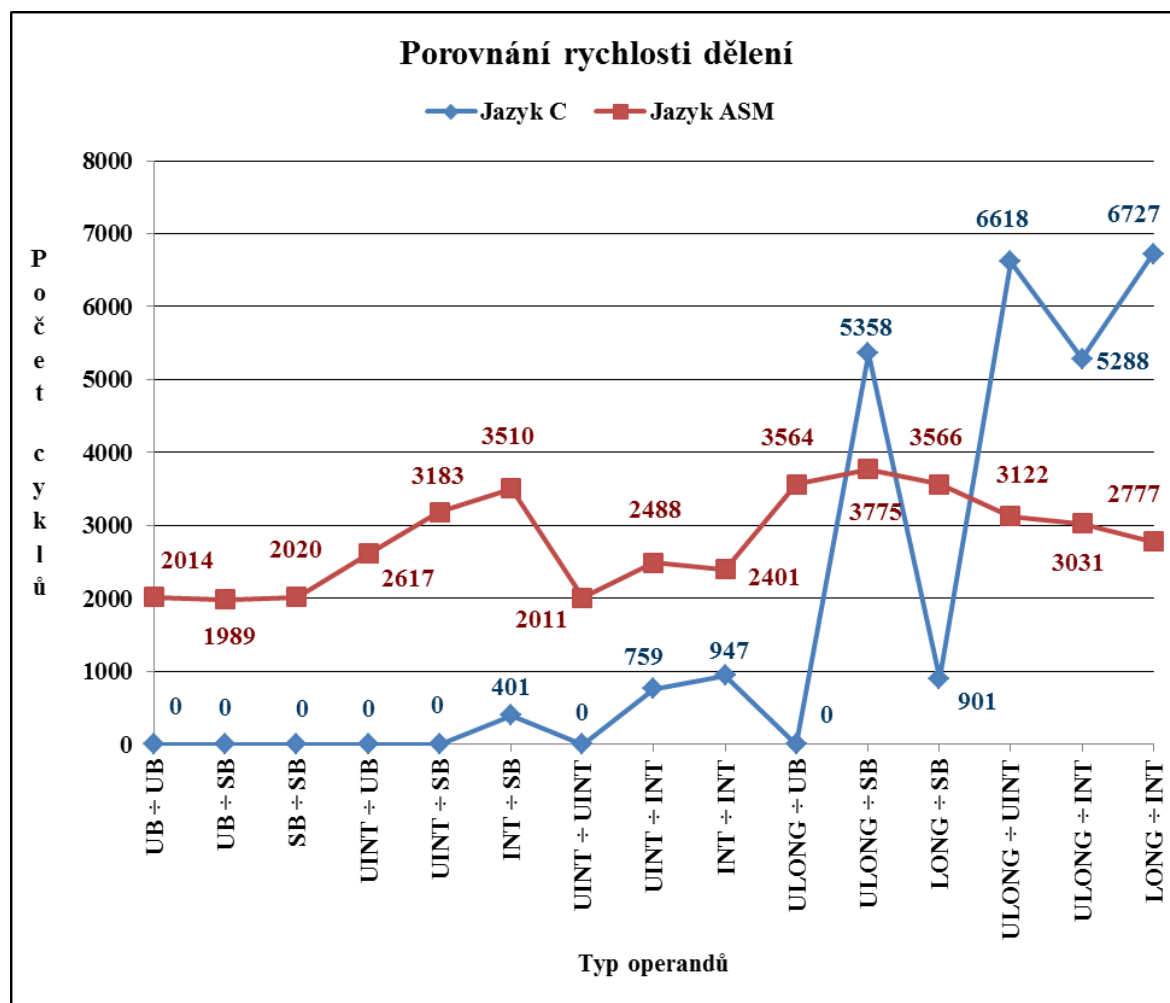
Dělení v C				
Typ	Počet cyklů	Hodnoty	Výsledek	Zbytek
UB ÷ UB	36	45 ÷ 12	3	9
UB ÷ SB	365	56 ÷ (-16)	-3	8
SB ÷ SB	411	(-78) ÷ (-55)	1	-23
UINT ÷ UB	174	14478 ÷ 173	83	119
UINT ÷ SB	441	15324 ÷ (-15)	0	141
INT ÷ SB	401	(-22880) ÷ (-15)	1525	-5
UINT ÷ UINT	815	32131 ÷ 18117	1	14014
UINT ÷ INT	759	32525 ÷ (-6919)	0	32525
INT ÷ INT	947	(-6354) ÷ (-2742)	2	-870
ULONG ÷ UB	623	632377 ÷ 137	4615	122
ULONG ÷ SB	5358	626568 ÷ (-84)	0	62668
LONG ÷ SB	901	(-123126) ÷ (-93)	1323	-87
ULONG ÷ UINT	6618	603839 ÷ 2578	234	587
ULONG ÷ INT	5288	622338 ÷ (-12488)	0	32514
LONG ÷ INT	6727	(-103555) ÷ (-13467)	7	-9286

Stejná čísla byla použita pro testování funkce DIVAB, přesto k žádným problémům nedošlo a výsledky byly správné (Tab. 11. na následující straně):

Tab. 11. Dělení v jazyce ASM

Dělení v ASM									
Typ	Počet cyklů						Hodnoty	Výsledek	Zbytek
	GETA	GETB	ADDAB	PUTA	PUTB	Celkem			
UB ÷ UB	56	56	1826	38	38	2014	45 ÷ 12	3	9
UB ÷ SB	56	56	1801	38	38	1989	56 ÷ (-16)	-3	8
SB ÷ SB	56	56	1832	38	38	2020	(-78) ÷ (-55)	1	-23
UINT ÷ UB	73	56	2412	38	38	2617	14478 ÷ 173	83	119
UINT ÷ SB	73	56	2938	58	58	3183	15324 ÷ (-15)	-1021	9
INT ÷ SB	73	56	3265	58	58	3510	(-22880) ÷ (-15)	1525	-5
UINT ÷ UINT	73	73	1749	58	58	2011	32131 ÷ 18117	1	14014
UINT ÷ INT	73	73	2226	58	58	2488	32525 ÷ (-6919)	252	4949
INT ÷ INT	73	73	2139	58	58	2401	(-6354) ÷ (-2742)	2	-870
ULONG ÷ UB	90	56	3302	58	58	3564	632377 ÷ 137	4615	122
ULONG ÷ SB	90	56	3513	58	58	3775	626568 ÷ (-84)	-7459	12
LONG ÷ SB	90	56	3304	58	58	3566	(-123126) ÷ (-93)	1323	-87
ULONG ÷ UINT	90	73	2843	58	58	3122	603839 ÷ 2578	234	587
ULONG ÷ INT	90	73	2752	58	58	3031	622338 ÷ (-12488)	207	10426
LONG ÷ INT	90	73	2498	58	58	2777	(-103555) ÷ (-13467)	1525	-5

Zde přibyl do tabulky další sloupec cyklů, jedná se o funkci PUTB, která vrací číslo z akumulátoru ACCB. To je v tomto případě zbytek po dělení, který je součástí výsledku a tudíž počet cyklů pro jeho získání musíme do tabulky zahrnout taktéž.



Graf 4. Porovnání rychlosti dělení dle jazyků

5 PŘÍKLAD VYUŽITÍ KNIHOVNY

Tento praktický příklad demonstruje možné využití navrhované knihovny. Program počítá faktoriál čísla v registru A. Výsledek uloží do paměti na místo, zadané v registru H:X.

Faktoriál je násobek klesající aritmetické řady, s diferencí rovnou jedné a končící 1. Například faktoriál 5:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Programu je číslo předáno v registru A, ten nejprve porovná, zda je číslo rovno 0 nebo 1, protože v takovém případě je výsledek automaticky roven 1. Je-li číslo větší než 1 program je uloží do pomocné proměnné, do registru H:X načte adresu této proměnné, do registru A načte typ tBYTE a volá funkci GETA pro načtení čísla do akumulátoru ACCA.

Pak program zahajuje smyčku. Načte číslo zpět do A, sníží jej o jedničku a není-li toto číslo rovno 1, uloží ho zpět do paměti, načte typ tBYTE do registru A a volá funkci GETB pro načtení čísla do akumulátoru ACCB. Následně zavolá funkci MULAB, která vynásobí čísla v obou akumulátorech a výsledek ponechá v akumulátoru ACCA.

Program opakovaně snižuje číslo a násobí jím výsledek, dokud není číslo rovno 1. Pak program skončí a výsledek uloží na adresu, kterou na začátku dostal v registru H:X.

Program a knihovna jsou přiloženy k bakalářské práci na CD-ROM.

ZÁVĚR

Cílem bakalářské práce bylo implementovat podporu aritmetických funkcí pro čísla různých typů, ve znaménkovém i neznaménkovém tvaru. Vytvořená knihovna má za úkol rozšířit aritmetické možnosti mikroprocesoru HCS08, aby bylo možno na tomto mikroprocesoru provádět složitější a časově náročnější výpočty.

Navržená knihovna tyto výpočty umožňuje a usnadňuje tím práci při psaní programů, které tyto výpočty potřebují. V jiném případě by autoři takových programů museli výpočty provádět sami, v rámci svého programu, což by mnohdy znamenalo nepřiměřenou a zdlouhavou práci navíc. Demonstrativní příklad na konci praktické části ukazuje, jaké zjednodušení knihovna poskytuje program pro výpočet faktoriálu. Bez existence matematické knihovny by byl jednoduchý algoritmus výpočtu zastíněn velkým množstvím kódu starajícím se o násobení čísel větších než jeden byte.

Pro návrh knihovny byly použity všeobecně známé algoritmy aritmetických operací pro více bytová čísla, které byly při implementaci upraveny na nejrychlejší možný výpočet. Při implementaci místy docházelo k situacím, kdy mikroprocesor nenabízěl vhodnou instrukci, která by danou operaci značně urychlila. Proto bylo nutno v těchto situacích použít zbytečně dlouhý kód, který zvyšoval dobu výkonu aritmetických operací. Přesto však se podařilo dosáhnout poměrně dobrých výsledků, vzhledem ke srovnání s optimalizovanými funkcemi jazyka C. Aritmetické operace jazyka C jsou při kompilaci nahrazeny optimálním kódem, který pracuje s čísly přímo na zásobníku, kde jsou proměnné uloženy. Tento optimální kód je téměř pro každý druh operandů jiný a tedy nemusí ztrácet čas, zjišťováním s jakými operandy pracuje a vybíráním vhodné procedury pro danou aritmetickou operaci. Na druhou stranu knihovna v jazyce ASM tuto možnost nemá, ale poskytuje výhody v univerzálnosti výpočtu.

Pro sčítání knihovna využívá jen čtyři procedury, mezi kterými se volí podle sčítaných operandů. Toto řešení znamená určité zpomalení, ovšem značnou úsporu délky kódu celé knihovny. Funkce odčítání obsahuje jen základní režii a zbytek přenechává na funkci sčítání, což taktéž přináší znatelnou úsporu. Daty knihovna obsazuje pouhých 23 bytů paměti, přes dva tisíce řádků dlouhý kód zabírá 2887 bytů paměti. Zde knihovna jasně vede oproti programu v C, který zabírá kódem 12925 bytů a daty 2150 bytů paměti. Navržená knihovna splnila požadované parametry a může sloužit jako školní pomůcka při výuce programování mikroprocesorů.

ZÁVĚR V ANGLIČTINĚ

The aim of this thesis was to implement arithmetic functions to support various types of numbers in the signed or unsigned shape. Created library's task is to extend the arithmetic abilities of HCS08 microcontroller to allow the microprocessor to perform more complex and time-consuming calculations.

Proposed library provides these calculations and makes writing of programs that need such calculations easier. In another case, the authors of such programs need to perform their own calculations, as part of its program, which would impose a disproportionate and often tedious extra work. Demonstrative example of the practical end of the work, the library offers simplicity to program to calculate the faktoriál of a number. Without a math library a simple algorithm of calculations would be overshadowed by the large amount of code, taking care to multiply numbers larger than one byte.

For the design of libraries have used the well-known algorithms of arithmetic operations for multiple byte numbers, which have been adapted for implementation at the fastest possible calculation. The implementation places there were situations where a microprocessor does not offer adequate instruction, which would greatly speed up the operation. It was therefore necessary in these situations, use the code too long to increase the performance of arithmetic operations. Nevertheless, library managed to achieve quite good results, due to comparison with the optimized C arithmetic operations, that are replaced at compile time optimal by code that works directly with the numbers on the stack, where the variables are stored. This code is optimal for almost every other kind of operand and therefore does not waste time identifying with what works operands and collection procedures for the appropriate arithmetic operation. On the other hand, library in ASM does not have this option, but provides benefits in the universality of the calculation.

For addition the library uses only four treatments, among which are selected according counted operands. This solution is a slowdown, however, considerable savings in code length. Subtraction function contains only the basic direction and the rest is up to the addition function, which also brings appreciable savings. Data Library occupies only 23 bytes of memory, over two thousand lines of code long takes 2887 bytes of memory. Here the library is a clear leader against the C program, which occupies 12,925 bytes, and code data 2150 bytes of memory. The proposed library meets the required parameters and can serve as teaching aids for teaching programming microprocessors.

SEZNAM POUŽITÉ LITERATURY

- [1] Freescale Semiconductor. *CPU08 Central Processor Unit Reference Manual*. 2001. [online]. [cit. 2011-05-25]. Dostupný z WWW: <http://www.freescale.com/files/microcontrollers/doc/ref_manual/CPU08RM.pdf>.
- [2] BURKHARD, M. *C pro mikrokontroléry*. Praha: BEN – technická literatura, 2003. 280 s. ISBN 80-7300-077-6.
- [3] VÁŇA, V. *Začínáme s mikrokontroléry Motorola HC08 Nitron*. Praha: BEN – technická literatura, 2003. 96 s. ISBN 80-7300-124-1.
- [4] Freescale Semiconductor: *HCS08 Family Reference Manual*. 1. verze. 2003. [online]. [cit. 2011-05-14]. Dostupný z WWW: <http://www.freescale.com/files/microcontrollers/doc/ref_manual/HCS08RMV1.pdf>.
- [5] Freescale Semiconductor. *MC9S08GB/GT Data Sheet*. 2.3. verze. 2004. [online]. [cit. 2011-05-25]. Dostupný z WWW: <http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S08GB60.pdf>.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

\$FF Číslo v hexadecimálním tvaru.

10b Číslo v binárním tvaru.

ASM Jazyk Assembler.

B Byte.

SEZNAM OBRÁZKŮ A GRAFŮ

Obrázky:

Obr. 1. Vnitřní registry mikroprocesoru HCS08	11
Obr. 2. Rozložení paměti v mikroprocesoru HCS08	12
Obr. 3. Schéma funkcí GETA a GETB	27
Obr. 4. Schéma funkcí PUTA a PUTB	27
Obr. 5. Schéma přenosu čísla do tempL	33
Obr. 6. Schéma algoritmu násobení pomocí rotací čísla ACCB	33
Obr. 7. Schéma rotace čísla přes C	35
Obr. 8. Schéma vytvoření kopie ACCA při dělení	36

Grafy:

Graf 1. Porovnání rychlosti sčítání dle jazyků	41
Graf 2. Porovnání rychlosti odčítání dle jazyků	44
Graf 3. Porovnání rychlosti násobení dle jazyků	46
Graf 4. Porovnání rychlostí dělení dle jazyků	49

SEZNAM TABULEK

Tab. 1. Typy určující velikost a tvar čísla.....	25
Tab. 2. Hodnoty CCR pro jednotlivé případy.....	29
Tab. 3. Odpovídající typy operandů jazyka C a knihovny v jazyce ASM	39
Tab. 4. Sčítání v jazyce C	39
Tab. 5. Sčítání v jazyce ASM	40
Tab. 6. Odčítání v jazyce C.....	42
Tab. 7. Odčítání v jazyce ASM.....	43
Tab. 8. Násobení v jazyce C	44
Tab. 9. Násobení v jazyce ASM	45
Tab. 10. Dělení v jazyce C.....	47
Tab. 11. Dělení v jazyce ASM.....	48