Vektorové instrukční sady a použití assembleru v gcc

Martin Bruchanov – BruXy
bruxy@regnet.cz
http://bruxy.regnet.cz

23. března 2009

1. Instrukční sady SIMD procesorů x86

1.1. MultiMedia eXtension – MMX

V roce 1997 uvedla firma Intel novou instrukční sadu typy SIMD (Single Instruction, Multiple Data – jedna instrukce zpracovává víc dat), nazvanou MMX (MultiMedia eXtension). Rozšíření přineslo 57 nových instrukcí a 4 datové typy, které se ukládají v 64bitových registrech 0 až 7. V

registrech je možné uložit 64 bitový integer (quadword), 2× 32 bitové integery (doubleword), 4× 16 bit. integery (word) nebo 8× 8 bit. integery (byte).

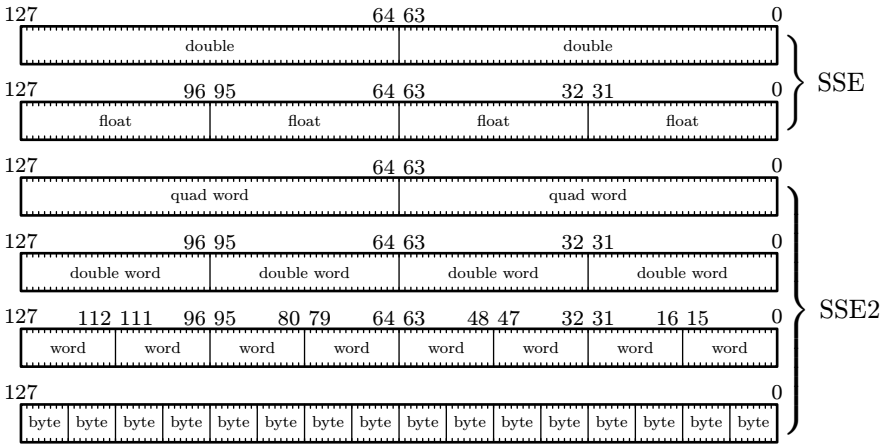
Nevýhodné je, že celočíselné registry 0 až 7 sdílí prostor s registry FPU `st0` až `st7`, což znemožňuje společné použití znaménkových a MMX instrukcí. A pokud programátor chce využít oboje musí mezi jednotlivými úseky kódu schovávat obsah registrů do paměti.

Firma AMD ve svých procesorech uvedla rozšíření MMX nazvané 3Dnow!, které přidává pro MMX podporu pro provádění operace v pohyblivé řadové čárce. Obsahuje 21 nových instrukcí pro SIMD operace v FP i celočíselné aritmetice, přenos dat do L1 cache (prefetching) a rychlejší přepínání mezi využitím MMX a FP instrukcí (instrukce `femms`). Registr může pojmout dvě 32bitová desetinná čísla v formátu IEEE-754 single precision.

1.2. Streaming SIMD Extensions – SSEx

Intel v procesoru Pentium III uvedl další SIMD rozšíření s názvem SSE (Streaming SIMD Extensions) v roce 1999 a za další dva roky SSE2 v procesoru Pentium IV, tato sada je dostupná i v procesorech AMD od řady Athlon XP/MP. V současné době je uvedena i sada SSE3 v procesorech Xeon a Intel Core 2 a jsou ohlášeny i SSE4 a SSE5.

SSE přineslo celkem 70 instrukcí. Registry mají 128 bitů, je jich celkem 8 `xmm0` až `xmm7` (v 64bitových modelech navíc ještě `xmm8` až `xmm15`). Mohou obsahovat 2× 64 bit. double nebo 4× 32 bit. float. Oproti MMX jsou registry v hardware řešeny samostatně, takže je možno je bez problémů kombinovat s operacemi v x87 FPU. Navíc přibyl řídící a stavový registr `mxcsr` (viz příloha 3.



Obrázek 1.: Uložení datových typů v registrech `xmm`.

Rozšíření SSE2 přidává ještě dalších 144 instrukcí jak pro operace v pohyblivé řadové čárce, tak i pro celočíselné operace a dále instrukce pro obsluhu cache. Další rozšíření spočívá v možnosti využít také MMX intstrukce pro operace s `xmm` registry.

Tato sada přišla s Pentiem IV v roce 2001. V SSE2 přibyla oproti sadě SSE podpora 64-bitových čísel v plovoucí řádové čárce (double precision) a dále podpora celočíselných operací s 64, 32, 16 a 8 bitovými celými čísly zabalenými v 128-bitových registrech `xmm`.

Přehled datových typů, které umí SSE2 zpracovávat a jsou uložené v registrech `xmm` je na obr. 1.

2. Využití SSE intrukcí

Možnosti, které `gcc` nabízí pro využití pokročilých instrukčních sad, jsou dvě:

1. ruční vkládání úseků kódu,
2. využití rozšíření pro vektorové operace, které `gcc` obsahuje od verze 3.x. Jazyk C na rozdíl od jiných jazyků nemá přímou podporu pro práci s vektorovými datovými typy.

2.1. Ruční propojení C a assembleru

Možností využití instrukční sady přímým vkládání úseků kódu demonstrují následující dva příklady. V unixovém světě je pro psaní assembleru obvyklejší syntaxe AT&T (viz příloha 4), ale `gcc` umožňuje využít i syntaxi Intel rozšířenou v OS Windows/DOS.

Pro přesun mezi pamětí a registry má SSE několik instrukcí, např. `movaps` a `movups` pro přesun ze zarovnané adresy, resp. nezarovnané.

Zarovnání proměnných je důležité pro rychlý přístup do paměti. Čtení z nezarovnané paměti obvykle vyžaduje dva čtecí cykly, než se požadovaná data dostanou do registru. Architektura x86 dovoluje číst i z nezarovnané paměti, ale čtení je pomalejší, Některé architektury dokonce ani nezarovnané čtení z paměti neumožňují a při takovém požadavku generují chybové výjimky, případně výjimka vyvolá nějakou softwarovou rutinu, která nezarovnaná data načte, ale bohužel tohle řešení může být pomalé.

Aby kompilátor adresy proměnných zarovnal na adresu dělitelnou beze zbytku šestnácti je nutné k proměnné přidat `__attribute__ (( aligned(16) ))`. V případě čtení z nezarovnané adresy instrukcí `movaps` je vyvolána výjimka a program havaruje s hlášením Segmentation fault.

Příklad syntaxe AT&T

Překlad: `gcc -msse2 -o vect_asm vect_asm.c`

```
typedef int xmm_reg[4] __attribute__ (( aligned(16) ));

int main(int argc, char *argv[])
{
       xmm_reg A = {1, 2, 3, 4};
       xmm_reg B = {4, 3, 2, 1};
       xmm_reg C;
```

```
        __asm__ (
                        "movaps %1, %%xmm0 \n\t"
                        "movaps %2, %%xmm1 \n\t"
                        "addps  %%xmm1, %%xmm0\n\t"
                        "movaps %%xmm0, %0 \n\t"
                        : "=m" (C[0]) /* %0, output */
                        : "m" (A[0]) /* %1 */, "m" (B[0]) /* %2, input */
                        : "xmm0", "xmm1" // clobered registers
        );

        printf("Vector: %u, %u, %u, %u\n", C[0], C[1], C[2], C[3]);
        return 0;
}
```

Příklad syntaxe Intel

Překlad: `gcc -msse2 -masm=intel -o vect_asm vect_asm.c`

Díky volbě `-masm=intel` generuje `gcc` kód v syntaxi Intel. Překladač GAS předpokládá název registru uvozený znakem „%“, ovšem programátor tuto vlastnost může dočasně potlačit direktivou `.intel_syntax noprefix` a při ukončení bloku `__asm__` opět zapnout pomocí `.intel_syntax prefix`.

Při použití intelovské syntaxe je důležité kompilovat s volbou `-masm=intel`. Pokud se pouze uvede direktiva `.intel_syntax`, pak kompilátor pro vložení adresovaných registrů použije syntaxi AT&T a překlad skončí hlášením assembleru o syntaktické chybě.

```
typedef int xmm_reg[4] __attribute__ (( aligned(16) ));

int main(int argc, char *argv[])
{
        xmm_reg A = {1, 2, 3, 4};
        xmm_reg B = {4, 3, 2, 1};
        xmm_reg C;

        __asm__ (
                        ".intel_syntax noprefix \n\t"
                        "movaps xmm0, %1        \n\t"
                        "movaps xmm1, %2        \n\t"
                        "addps  xmm0, xmm1      \n\t"
                        "movaps %0,   xmm0      \n\t"
                        ".intel_syntax prefix   \n\t"
                        : "=m" (C[0]) /* %0, output */
                        : "m" (A[0]) /* %1 */, "m" (B[0]) /* %2, input */
                        : "xmm0", "xmm1", "memory"   // clobered registers
        );
```

```
        printf("Vector: %u, %u, %u, %u\n", C[0], C[1], C[2], C[3]);
        return 0;
}
```

2.2. Rozšíření gcc pro operace nad vektory

Definice datového typu vektor se provádí podobně jako u ostatních typů direktivou `typedef`. Je nutno zadefinovat počet elementů vektoru, např. pro vektor o rozměrech 4 × `int`:

```
typedef int v4si __attribute__ (( vector_size(4 * sizeof(int)) ));
```

Jediným omezením je fakt, že počet prvků vektoru musí být mocnina dvou. S takto definovaným vektorovým datovým typem je pak možno pracovat stejně jako se skalárním. Kompilátor `gcc` zpracovává tyto vektorové operátory: `+`, `-`, `*` (násobení prvků na stejných pozicích), `/`, unární mínus, `^` (bitový XOR), `|` (bit. OR), `&` (bit. AND), `~` (bit. negace).

Výhodou je, že takto zadefinované vektorové operace poskytují abstrakci nad hardware a kód je bez problémů možno přeložit pro nejrůznější architektury (Intel MMX, Intel SSE Motorola AltiVec, Sun VIS, atd.). Přeložit kód je možné i pro ty architektury, které žádné SIMD rozšíření neobsahují (kompilátor operaci převede na provádění jednotlivých skalární operací).

Hlavní nevýhodou tohoto řešení je, že zdrojový kód přestává být snadno přenositelný mezi jednotlivými kompilátory.

Příklad, součet dvou vektorů

Následující zdrojový kód demonstruje možnosti vektorových operací.

```
typedef int v4si __attribute__ (( vector_size(4 * sizeof(int)) ));

int main(int argc, char *argv[])
{
        v4si A = {1, 2, 3, 4};
        v4si B = {4, 3, 2, 1};
        v4si C;

        C = A + B;

        return 0;
}
```

Nejprve provedeme překlad bez specifikace instrukčního souboru: `gcc -ggdb3 -o vector vector.c`. Díky volbě `-ggdb3` při pozdějším disassemblování kódu uvidíme provázanost jazyka C a instrukcí. Disassemblování provedeme příkazem `objdump -dS -M intel vector`:

```
       v4si A = {1, 2, 3, 4};
80483a7:       c7 45 b8 01 00 00 00    mov    DWORD PTR [ebp-0x48],0x1
80483ae:       c7 45 bc 02 00 00 00    mov    DWORD PTR [ebp-0x44],0x2
80483b5:       c7 45 c0 03 00 00 00    mov    DWORD PTR [ebp-0x40],0x3
80483bc:       c7 45 c4 04 00 00 00    mov    DWORD PTR [ebp-0x3c],0x4
       v4si B = {4, 3, 2, 1};
80483c3:       c7 45 c8 04 00 00 00    mov    DWORD PTR [ebp-0x38],0x4
80483ca:       c7 45 cc 03 00 00 00    mov    DWORD PTR [ebp-0x34],0x3
80483d1:       c7 45 d0 02 00 00 00    mov    DWORD PTR [ebp-0x30],0x2
80483d8:       c7 45 d4 01 00 00 00    mov    DWORD PTR [ebp-0x2c],0x1
       v4si C;

       C = A + B;
80483df:       8b 55 b8                mov    edx,DWORD PTR [ebp-0x48]
80483e2:       8b 45 c8                mov    eax,DWORD PTR [ebp-0x38]
80483e5:       8d 0c 02                lea    ecx,[edx+eax]
80483e8:       8b 55 bc                mov    edx,DWORD PTR [ebp-0x44]
80483eb:       8b 45 cc                mov    eax,DWORD PTR [ebp-0x34]
80483ee:       8d 1c 02                lea    ebx,[edx+eax]
80483f1:       8b 55 c0                mov    edx,DWORD PTR [ebp-0x40]
80483f4:       8b 45 d0                mov    eax,DWORD PTR [ebp-0x30]
80483f7:       8d 34 02                lea    esi,[edx+eax]
80483fa:       8b 55 c4                mov    edx,DWORD PTR [ebp-0x3c]
80483fd:       8b 45 d4                mov    eax,DWORD PTR [ebp-0x2c]
8048400:       8d 04 02                lea    eax,[edx+eax]
8048403:       89 4d d8                mov    DWORD PTR [ebp-0x28],ecx
8048406:       89 5d dc                mov    DWORD PTR [ebp-0x24],ebx
8048409:       89 75 e0                mov    DWORD PTR [ebp-0x20],esi
804840c:       89 45 e4                mov    DWORD PTR [ebp-0x1c],eax
```

Nyní pro překlad budeme specifikovat použití instrukční sady SSE2: `gcc -msse2 -ggdb3 -o vector vector.c`. Disassemblovaný kód je následující:

```
       v4si A = {1, 2, 3, 4};
80483a5:       66 0f 6f 05 c0 84 04    movdqa xmm0,XMMWORD PTR ds:0x80484c0
80483ac:       08
80483ad:       66 0f 7f 45 c8          movdqa XMMWORD PTR [ebp-0x38],xmm0
       v4si B = {4, 3, 2, 1};
80483b2:       66 0f 6f 05 d0 84 04    movdqa xmm0,XMMWORD PTR ds:0x80484d0
80483b9:       08
80483ba:       66 0f 7f 45 d8          movdqa XMMWORD PTR [ebp-0x28],xmm0
       v4si C;

       C = A + B;
80483bf:       66 0f 6f 45 d8          movdqa xmm0,XMMWORD PTR [ebp-0x28]
80483c4:       66 0f fe 45 c8          paddd  xmm0,XMMWORD PTR [ebp-0x38]
80483c9:       66 0f 7f 45 e8          movdqa XMMWORD PTR [ebp-0x18],xmm0
```

Pokud by vektor v příkladu byl větší než je rozsah registrů SSE2 nebo daná operace nemá svůj ekvivalent v podobě jedné instrukce, kompilátor přeloží kód tak, že se výraz rozloží do dílčích operací, ale opět s využitím instrukcí SSE2.

2.2.1. Práce s vektory

S vektorovými datovými typy `gcc` pracuje obdobným způsobem jako s ostatními. Je možno je použít jako argumenty funkcí, jako návratové hodnoty apod. V případě přiřazování je možno vektor přetypovat, ale pouze na vektor stejné velikosti. Operace není možná na vektorech rozdílné velikosti nebo jiného znaménkového typu. V tomto případě je nutné přetypovat.

2.2.1.1. Přístup k prvkům

Zvláštní postup vyžaduje přístup k jednotlivým prvkům vektoru. Nabízí se dva možné přístupy. První možnost je přetypovat ukazatel na vektor a k položkám přistupovat pomocí pointerové aritmetiky:

```
v4si A = { 1, 2, 3, 4 };  // vektor o velikosti 4
int  *S;                  // ukazatel na integer / pole integerů

S = (int *) &A;           // S ukazuje na začátek vektoru A

printf("Vector: %u %u %u %u \n", S[0], S[1], S[2], S[3]);
```

Druhý způsob využívá pro uložení vektoru datový typ union, který se podobá struktuře. Rozdíl spočívá v tom, že paměťová velikost unionu je rovná velikosti největší položky a jednotlivé položky se v paměti „překrývají“.

Pro vektor o velikosti čtyři 32bitové celočíselné prvky je definice následující:

```
typedef union {
       v4si v;
       int  s[4];
} vector;
```

K jednotlivým prvkům vektoru je pak možné přistupovat přes pole `s` uvnitř unionu.

```
vector A.v = {{ 1, 2, 3, 4 }};

printf("Vector: %u %u %u %u \n", A.s[0], A.s[1], A.s[2], A.s[3]);
```

2.2.2. Funkce pro volání instrukcí

Následující „zabudované“ funkce umožňují přímé volání instrukcí dané sady a jsou dostupné při překladu se specifikací dané instrukční sady a procesoru. Pro tyto účely slouží přepínače `gcc`: `-mmmx`,

`-msse`, `-msse2`, `-m3dnow -march=athlon`, `-msse3`, bez nichž překladač ohlásí chybu o nedefinované funkci.

Pro SSE a SSE2 má kompilátor definovány vektorové typy, které mají všechny velikost 128 bitů / 16 bytů (`vector_size(16)`), ale liší se typem a počtem uložených elementů, vit tab 1.

Tabulka 1.: Vektorové datové typy.

Rozměr vektoru	Rozměr prvku	Počet prvků	Označení	C typ vektoru	C typ prvku
128 b	32 b	4	packed single-precision float (SSE)	`v4sf`	`float`
128 b	64 b	2	packed double-precision float (SSE)	`v2df`	`double`
128 b	128 b	1	celé slovo (SSE2)	`di`	
128 b	64 b	2	packed quad word (SSE2)	`v4di`	`long long`
128 b	32 b	4	packed double word (SSE2)	`v4si`	`int`
128 b	16 b	8	packed word (SSE2)	`v8hi`	`short`
128 b	8 b	16	packed byte (SSE2)	`v16qi`	`char`

Předchozí příklad pro sečtení dvou vektorů upravený s použitím zabudované funkce:

```
v4si A = {1, 2, 3, 4};
v4si B = {4, 3, 2, 1};
v4si C;

C = __builtin_ia32_paddd128(A, B); // C = A + B;
```

Kompletní výčet funkcí obsahuje manuál `gcc` [4] a je vhodné se podívat i do příslušných hlavičkových souborů dodávaných s `gcc`. Pro SSE2 je to `xmmintrin.h`. Reference některých použitých instrukcí jsou v příloze 3.

Kompilátor `gcc` také dovoluje použít zabudované funkce v notaci kompilátoru ICC firmy Intel. Mapování funkci je uvedeno v `xmmintrin.h`.

2.2.3. Řízení datového toku cache

SSE obsahuje instrukce pro řízení toku dat mezi hlavní pamětí a cache. Pomocí intrukcí `PREFETCHx` je možné označit adresu paměti před jejím použitím a tím snížit zdržení výpadky cache před použitím paměťového úseku. Minimální velikost paměťového úseku je 32 bytů. Pro označovanou adresu není kladen nárok na zarovnání.

```
void __builtin_prefetch (const void *addr, ...)
void __builtin_prefetch (const void *addr, rw, locality)
```

Dva volitelné parametry se zadávají v čase kompilace:

- `rw` – načtení adresy pro zápis (hodnota 1) nebo pro čtení (0), hodnota 0 je přednastavená.
- `locality` – možné hodnoty 0 až 3. Hodnota 0 znamená, že data nemají žádnou dočasnou lokalitu a není třeba je ponechávat v cache po přístupu. Hodnota 3 vyjadřuje naopak to, že data mají vysoký stupeň lokality a měla by zůstat ve všech úrovních cache, pokud je to možné. Hodnoty 1 a 2, znamenají nižší stupeň lokality. Přednastavená hodnota je 3.

2.3. Testování rychlosti kódu

Pro představu uvádím jednoduchý příklad funkce pro rozdíl dvou obrazů. První úsek kódu je běžná implementace, která postupně projde pixely obou obrazů (`datasize` $= M \times N$ je bezezbytku dělitelné 16) a provede jejich rozdíl:

```
for(i = 0; i < datasize; ++i)
{
       int a = A->data[i], b = B->data[i];
       out->data[i] = (unsigned char) a - b;
}
```

S použitím vektorového rozšíření a vestavěných funkcí pro načtení, uložení (`loaddqu` a `storedqu`), a řízení toku cache vypadá následovně:

```
v16qi vA, vB, vC;
__builtin_prefetch(A->data, 0, 1);   // čtení
__builtin_prefetch(B->data, 0, 1);   // čtení
__builtin_prefetch(out->data, 1, 1); // zápis
for(i = 0; i < datasize ; i += 16)
{
       __builtin_prefetch(&A->data[i+16], 0, 1);      // prefetch pro následující
       __builtin_prefetch(&B->data[i+16], 0, 1);      // cyklus
       __builtin_prefetch(&out->data[i+16], 1, 1);
       vA = __builtin_ia32_loaddqu((char *) &A->data[i]);
       vB = __builtin_ia32_loaddqu((char *) &B->data[i]);
       vC = vA - vB;
       __builtin_ia32_storedqu(&out->data[i], vC);
}
```

Vyzkoušel jsem jak rychle dokáží obě implementace odečíst dva snímky v rozlišení 3712×3712 (13 MB), pro několik nastavení (`gcc 4.3.0`).

Pro měření rychlosti vykonání funkce jsem použil `gprof` (The GNU Profiler) a zaznamenával střední hodnotu pro 5 běhů. Není ani tak důležitý čas operace jako spíše hodnota relativního zrychlení.

1. Běžná implementace, bez optimalizace — 260 ms, zrychlení 0 %.
2. Běžná implementace, `-O3` — 130 ms, zrychlení 50 %.
3. Běžná implementace, rozbalení cyklů `-funroll-loops`, `-O3` — 50 ms, zrychlení 81 %.
4. Pomocí vekt. rozšíření, bez optimalizace — 40 ms, zrychlení 85 %.
5. Pomocí vekt. rozšíření s přednačítáním do cache, bez optimalizace — 40 ms, zrychlení 85 %.
6. Pomocí vekt. rozšíření, `-O3` — 30 ms, zrychlení 89 %.
7. Pomocí vekt. rozšíření, rozbalení cyklů `-funroll-loops`, `-O3` — 20 ms, zrychlení 92 %.

Žádná změna v rychlosti obou vektorových variant s přednačítáním a bez je dána tím, že se uplatňuje princip lokality a díky tomu, že data obrazu leží v paměti hned za sebou, jsou přednačítána automaticky. Ačkoliv je v každém cyklu několik taktů ztraceno na provádění intrukcí `prefetch` výsledek měření to neovlivnilo.

Pro testování, jsem předchozí kód přepsal přímo do inline assembleru, vestavěné funkce jsou totiž omezeny svým datovým typem. Protože data obrazu jsou zarovnána na 16, je možné použít přímo instrukci `movups`. Změřená rychlost kódu byla 30 ms, tedy stejná jako v případě 6. Zde se kód po kompilaci `-funroll-loops` nezrychlil, kompilátor na funkci s vloženým assembler není schopen uplatnit optimalizace na rozbalení cyklů.

3. Vektorové operace SSE2 jako funkce GCC

- `v16qi __builtin_ia32_paddb128 (v16qi, v16qi)`
  - Instrukce: PADDB – Packed Add.
  - Funkce: Sečíst registry po byty.
  - Operace:
    `DEST[7..0] ← DEST[7..0] + SRC[7..0];`
- `v16qi __builtin_ia32_psubb128 (v16qi, v16qi)`
  - Instrukce: PSUBB – Packed Subtract.
  - Funkce: Odečíst registry po bytech.
  - Operace:
    `DEST[7..0] ← DEST[7..0] + SRC[7..0];`
- `v16qi __builtin_ia32_pavgb128 (v16qi, v16qi)`
  - Instrukce: PAVGB – Packed Average.
  - Funkce: Průměr registrů po bytech.
  - Operace:
    `SRC[7-0] ← (SRC[7-0] + DEST[7-0] + 1) >> 1;`
- `v16qi __builtin_ia32_pcmpeqb128 (v16qi, v16qi)`

  - Instrukce: PCMPEQB – Packed Compare for Equal.
  - Funkce: Porovnání prvků dvou vektorů.
  - Operace:

    ```
    IF DEST[7..0] = SRC[7..0] THEN DEST[7..0] ← 0xFF;
    ELSE DEST[7..0] ← 0x0;
    ```

- `v16qi __builtin_ia32_pcmpgtb128 (v16qi, v16qi)`
  - Instrukce: PCMPGTB – Packed Compare for Greater Than.
  - Funkce: Porovnání prvků „větší než“ $\geq$.
  - Operace:

    ```
    IF DEST[7..0] > SRC[7..0] THEN DEST[7..0] ← 0xFF;
    ELSE DEST[7..0] ← 0x0;
    ```

- `v16qi __builtin_ia32_pmaxub128 (v16qi, v16qi)`
  - Instrukce: PMAXUB – Packed Unsigned Integer Byte Maximum.
  - Funkce: Vrátí maximální hodnotu pro každou dvojici prvků.
  - Operace:

    ```
    IF DEST[7..0] > SRC[7..0] THEN DEST[7-0] ← DEST[7-0];
    ELSE DEST[7-0] ← SRC[7-0];
    ```

- `v16qi __builtin_ia32_pminub128 (v16qi, v16qi)`
  - Instrukce: PMINUB – Packed Unsigned Integer Byte Minimum.
  - Funkce: Vrátí minimální hodnotu pro každou dvojici prvků.
  - Operace:

    ```
    IF DEST[7-0] < SRC[7-0] THEN DEST[7-0] ← DEST[7-0];
    ELSE DEST[7-0] ← SRC[7-0];
    ```

- `v16qi __builtin_ia32_punpckhbw128 (v16qi, v16qi)`
  - Instrukce: PUNPCKHBW – Unpack High Packed Data.
  - Funkce: Proloží prvky horních polovin dvou vektorů.
  - Operace:

    ```
    DEST[7..0] ← ← DEST[71..64]; DEST[15..8] ← SRC[71..64];
    DEST[23..16] ← DEST[79..72]; DEST[31..24] ← SRC[79..72];
    DEST[39..32] ← DEST[87..80]; DEST[47..40] ← SRC[87..80];
    DEST[55..48] ← DEST[95..88]; DEST[63..56] ← SRC[95..88];
    DEST[71..64] ← DEST[103..96]; DEST[79..72] ← SRC[103..96];
    DEST[87..80] ← DEST[111..104]; DEST[95..88] ← SRC[111..104];
    DEST[103..96] ← DEST[119..112]; DEST[111..104] ← SRC[119..112];
    DEST[119..112] ← DEST[127..120]; DEST[127..120] ← SRC[127..120];
    ```

- `v16qi __builtin_ia32_punpcklbw128 (v16qi, v16qi)`

  - Instrukce: PUNPCKLBW – Unpack Low Packed Data.
  - Funkce: Proloží prvky dolních polovin dvou vektorů.
  - Operace:

```
DEST[7..0] ← DEST[7..0]; DEST[15..8] ← SRC[7..0];
DEST[23..16] ← DEST[15..8]; DEST[31..24] ← SRC[15..8];
DEST[39..32] ← DEST[23..16]; DEST[47..40] ← SRC[23..16];
DEST[55..48] ← DEST[31..24]; DEST[63..56] ← SRC[31..24];
DEST[71..64] ← DEST[39..32]; DEST[79..72] ← SRC[39..32];
DEST[87..80] ← DEST[47..40]; DEST[95..88] ← SRC[47..40];
DEST[103..96] ← DEST[55..48]; DEST[111..104] ← SRC[55..48];
DEST[119..112] ← DEST[63..56]; DEST[127..120] ← SRC[63..56];
```

- `v16qi __builtin_ia32_packsswb128 (v16qi, v16qi)`
  - Instrukce: PACKSSWB – Pack with Signed Saturation.
  - Funkce: Sbal 8 wordů se znaménkem ze zdrojového a 8 wordů z cílového operandu do 16 znaménkových bytů cílového registru. Pokud je znaménková hodnota wordu větší nebo menší než je rozsah znaménkového bytu, pak v případě přetečení je maximální hodnota `0x7F` a při podtečení `0x80`.
  - Operace:

```
DEST[7-0] ← SatSigW2SignB(DEST[15-0]);
DEST[15-8] ← SatSigW2SignB(DEST[31-16]);
DEST[23-16] ← SatSigW2SignB(DEST[47-32]);
DEST[31-24] ← SatSigW2SignB(DEST[63-48]);
DEST[39-32] ← SatSigW2SignB(DEST[79-64]);
DEST[47-40] ← SatSigW2SignB(DEST[95-80]);
DEST[55-48] ← SatSigW2SignB(DEST[111-96]);
DEST[63-56] ← SatSigW2SignB(DEST[127-112]);
DEST[71-64] ← SatSigW2SignB(SRC[15-0]);
DEST[79-72] ← SatSigW2SignB(SRC[31-16]);
DEST[87-80] ← SatSigW2SignB(SRC[47-32]);
DEST[95-88] ← SatSigW2SignB(SRC[63-48]);
DEST[103-96] ← SatSigW2SignB(SRC[79-64]);
DEST[111-104] ← SatSigW2SignB(SRC[95-80]);
DEST[119-112] ← SatSigW2SignB(SRC[111-96]);
DEST[127-120] ← SatSigW2SignB(SRC[127-112]);
```

- `v16qi __builtin_ia32_packuswb128 (v16qi, v16qi)`

  - Instrukce: PACKUSWB – Pack with Unsigned Saturation.
  - Funkce: Sbal 8 wordů se znaménkem ze zdrojového a 8 wordů se znaménkem z cílového registru do registru obsahujícím 16 bytů bez znaménka. Pokud je hodnota wordu mimo rozsah hodnot jednoho bytu je přetečená hodnota uložena jako `0xff` a podtečená jako `0x00`.
  - Operace:

```
DEST[7..0] ← SatSigW2UnsignB(DEST[15..0]);
DEST[15..8] ← SatSigW2UnsignB(DEST[31..16]);
DEST[23..16] ← SatSigW2UnsignB(DEST[47..32]);
DEST[31..24] ← SatSigW2UnsignB(DEST[63..48]);
DEST[39..32] ← SatSigW2UnsignB(DEST[79..64]);
DEST[47..40] ← SatSigW2UnsignB(DEST[95..80]);
DEST[55..48] ← SatSigW2UnsignB(DEST[111..96]);
DEST[63..56] ← SatSigW2UnsignB(DEST[127..112]);
DEST[71..64] ← SatSigW2UnsignB(SRC[15..0]);
DEST[79..72] ← SatSigW2UnsignB(SRC[31..16]);
DEST[87..80] ← SatSigW2UnsignB(SRC[47..32]);
DEST[95..88] ← SatSigW2UnsignB(SRC[63..48]);
DEST[103..96] ← SatSigW2UnsignB(SRC[79..64]);
DEST[111..104] ← SatSigW2UnsignB(SRC[95..80]);
DEST[119..112] ← SatSigW2UnsignB(SRC[111..96]);
DEST[127..120] ← SatSigW2UnsignB(SRC[127..112]);
```

- `void __builtin_ia32_maskmovdqu (v16qi, v16qi)`
  - Instrukce: MASKMOVDQU – Mask Move of Double Quadword Unaligned.
  - Funkce: Uloží vybrané byty ze zdrojového operandu do paměti.
  - Operace:

```
IF (MASK[7] = 1) THEN DEST[DI/EDI] ← SRC[7-0] ELSE nemění paměť;
IF (MASK[15] = 1) THEN DEST[DI/EDI+1] SRC[15-8] ELSE nemění paměť;
...
IF (MASK[127] = 1) THEN DEST[DI/EDI+15] SRC[127-120] ELSE nemění paměť; FI;
```

- `int __builtin_ia32_pmovmskb128 (v16qi)`
  - Instrukce: PMOVMSKB – Move Byte Mask to General-Purpose Register.
  - Funkce: Vytvoří bitovou masku z nejvíce významových bitů každého bytu a uloží výsledek do nižšího slova cílového operandu.
  - Operace:

```
r32[0] ← SRC[7], r32[1] ← SRC[15], ...
```

- `v2di __builtin_ia32_psadbw128 (v16qi, v16qi)`

  - Instrukce: PSADBW – Packed Sum of Absolute Differences.
  - Funkce: Zjistí součet absolutních rozdílů prvků registru.
  - Operace:

    ```
    TEMP0 ← ABS(DEST[7-0] - SRC[7-0]);
    ...
    TEMP15 ABS(DEST[127-120] - SRC[127-120]);
    DEST[15-0] SUM(TEMP0...TEMP7);
    DEST[63-6] 000000000000H; DEST[79-64] SUM(TEMP8...TEMP15);
    DEST[127-80] 000000000000H;
    ```

- `v16qi __builtin_ia32_loaddqu (const char *)`
  - Instrukce: Load Double Quad Unaligned.
  - Funkce: Přesune double quad z nezarovnané paměti do registru.
  - Operace:

    ```
    DEST ← ADDR
    ```

- `void __builtin_ia32_storedqu (char *, v16qi)`
  - Instrukce: Store Double Quad Unaligned.
  - Funkce: Uloží double quad v registu do paměti.
  - Operace:

    ```
    ADDR ← SRC
    ```

3.1. Vektorové operace s 16bit. prvky

Instrukce v předchozí části mají své ekvivalenty pro zpracování 16bit. slov (word, `v8hi`). Navíc jsou k dispozici instrukce pro balení a prohazování prvků a posuvy.

```
v8hi __builtin_ia32_paddw128 (v8hi, v8hi)
v8hi __builtin_ia32_psubw128 (v8hi, v8hi)
v8hi __builtin_ia32_pavgw128 (v8hi, v8hi)
v8hi __builtin_ia32_pcmpeqw128 (v8hi, v8hi)
v8hi __builtin_ia32_pcmpgtw128 (v8hi, v8hi)
v8hi __builtin_ia32_pmaxsw128 (v8hi, v8hi)
v8hi __builtin_ia32_pminsw128 (v8hi, v8hi)
v8hi __builtin_ia32_punpckhwd128 (v8hi, v8hi)
v8hi __builtin_ia32_punpcklwd128 (v8hi, v8hi)
v8hi __builtin_ia32_packssdw128 (v8hi, v8hi)
v4si __builtin_ia32_pmaddwd128 (v8hi, v8hi)
v8hi __builtin_ia32_pmullw128 (v8hi, v8hi)
v8hi __builtin_ia32_pmulhw128 (v8hi, v8hi)
v8hi __builtin_ia32_pmulhuw128 (v8hi, v8hi)
v8hi __builtin_ia32_pshuflw (v8hi, int)
```

```
v8hi __builtin_ia32_pshufhw (v8hi, int)
v8hi __builtin_ia32_psllw128 (v8hi, v2di)
v8hi __builtin_ia32_psrlw128 (v8hi, v2di)
v8hi __builtin_ia32_psraw128 (v8hi, v2di)
v8hi __builtin_ia32_psllwi128 (v8hi, int)
v8hi __builtin_ia32_psrlwi128 (v8hi, int)
v8hi __builtin_ia32_psrawi128 (v8hi, int)
```

3.2. Význam bitů stavového registru `mxcsr`

Tabulka 2.: Význam bitů stavového registru `mxcsr`.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	MM	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FZ	RC		PM	UM	OM	ZM	DM	IM	DZ	PE	UE	OE	ZE	DE	IE

- $\texttt{IE}_0$ Invalid-Operation Exception 0
- $\texttt{DE}_1$ Denormalized-Operand Exception 0
- $\texttt{ZE}_2$ Zero-Divide Exception 0
- $\texttt{OE}_3$ Overflow Exception 0
- $\texttt{UE}_4$ Underflow Exception 0
- $\texttt{PE}_5$ Precision Exception 0
- $\texttt{DZ}_6$ Denormals are Zeros 0
- $\texttt{IM}_7$ Invalid-Operation Exception Mask 1
- $\texttt{DM}_8$ Denormalized-Operand Exception Mask 1
- $\texttt{ZM}_9$ Zero-Divide Exception Mask 1
- $\texttt{OM}_{10}$ Overflow Exception Mask 1
- $\texttt{UM}_{11}$ Underflow Exception Mask 1
- $\texttt{PM}_{12}$ Precision Exception Mask 1
- $\texttt{RC}_{13-14}$ Floating-Point Rounding Control 00
- $\texttt{FZ}_{15}$ Flush-to-Zero for Masked Underflow 0
- $\texttt{RZ}_{16}$ Reserved 0
- $\texttt{MM}_{17}$ Misaligned Exceptuion Mask 0
- Ostatní bity rezervovány

4. Porovnání syntaxe assembleru AT&T a Intel

4.1. Úvod

Ve světě unixových počítačů je zažitá syntaxe jazyka symbolických adres (assembleru), která pochází z dílen společnosti AT&T Bell Labs, kde byl vytvořen první operační systém UNIX. Tato syntaxe se nazývá AT&T/386 a je obvyklá i pro některé další rodiny procesorů.

Další typ syntaxe vznikl ve firmě Intel, a proto se jí říká syntaxe intelovská, která je dobře známá vývojářům na platformě x86.

Na linuxu je rozšířený assembler `gas` – GNU Assembler, který je součástí sady `binutils`. Také assemblerový kód, který produkují ostatní programy z kolekce překladačů GNU (`gcc`, `g++`, `objdump`,...) je právě určený pro další zpracování pomocí `gas`.

4.2. Srovnání syntaxe

4.2.1. Základní instrukce

Základní rozdíl obou syntaxí je v pořadí operandů:

- AT&T: `instrukce` *`zdroj, cíl`*
- Intel: `instrukce` *`cíl, zdroj`*

Dále se liší způsob zadávání registrů a přímých operandů:

- AT&T:
  - registry: `%eax`, `%ebx`, `%ecx`, `%ax`, ...
  - přímý operand je uvozen znakem „$“: `$128`, `$0x01`, ...
- Intel:
  - registry: `eax`, `ebx`, `ecx`, `ax`, ...
  - přímý operand: `128`, `0x01`, ...

Velikost dat je u AT&T rozlišena suffixem:

- AT&T: např. `movb` (byte – 8bit), `movw` (word – 16bit), `movl` (long – 32bit)
- Intel specifikuje velikost dat u operandu pomocí direktiv `byte ptr`, `word ptr`, `dword ptr`

Instrukce vyžadující suffix: `adc`, `add`, `and`, `bound`, `bsf`, `bsr`, `bt`, `btc`, `btr`, `bts`, `call`, `cmp`, `cmpxchg`, `dec`, `div`, `idiv`, `imul`, `in`, `inc`, `lar`, `lds`, `les`, `lfs`, `lgs`, `lss`, `lea`, `mov`, `mul`, `neg`, `not`, `or`, `out`, `pop`, `push`, `rcl`, `rcr`, `rol`, `ror`, `sal`, `sar`, `sbb`, `shl`, `shr`, `shld`, `shrd`, `sub`, `test`, `xadd`, `xbts`, `xchg`, `xor`.
Příklady:

AT&T	Intel
`pushl $128`	`push 128`
`addl $128, %eax`	`add eax, 128`
`xorl %eax, %eax`	`xor eax, eax`
`movl %ebx, %eax`	`mov eax, ebx`
`movb foo, %eax`	`mov eax, dword ptr foo`

4.2.1.1. Výpočet efektivní adresy

- AT&T: `segment:přírůstek(báze, index, násobící_faktor)`
- Intel: `segment:[báze + index * násobící_faktor + přírůstek]`

Složky výpočtu offsetové části adresy:

- přírůstek (displacement) – přímá adresa
- báze – obsah registru s bází
- index – obsah indexového registru
- násobící faktor (scaling factor) – indexový registr je možné násobit 2, 4 nebo 8

Příklady:

AT&T	Intel
`0x804838e`	`[0x804838e]`
`(%eax)`	`[eax]`
`(%eax,%ebx)`	`[eax+ebx]`
`(%ecx,%ebx,2)`	`[ecx+ebx*2]`
`(,%ebx,2)`	`[ebx*2]`
`-10(%eax)`	`[eax-10]`
`movl 4(%ebp), %eax`	`mov eax, [ebp+4]`
`addl (%eax,%eax,4), %ecx`	`add ecx, [eax + eax*4]`
`movl array(,%eax,4), %eax`	`mov eax, dword ptr [4*eax + array]`
`movw array(%ebx,%eax,4), %ecx`	`mov ecx, dword ptr [ebx + 4*eax + array]`

4.2.2. Výjimky v názvech instrukcí

4.2.2.1. Instrukce skoku

Dlouhá volání a skoky používají různá vyjádření segmentu a offsetu:

- AT&T: `lcall` / `ljmp` *`$segment`*, *`$offset`*
- Intel: `call` / `jmp far` *`segment`*`:`*`offset`*

4.2.2.2. Znaménkové rozšíření

Intrukce pro znaménkové rozšíření mají odlišný název:

Rozšíření	AT&T	Intel
AL → AH	`cbtw`	`cbw`
AX → EAX	`cwtl`	`cwde`
AX → DX:AX	`cwtd`	`cwd`
EAX → EDX:EAX	`cltd`	`cdq`
EAX → RAX	`cltq`	`cdqe`
RAX → RDX:RAX	`cqto`	`cqo`

4.2.2.3. Přesuny se znaménkovým rozšířením

Instrukce `movsx` – přesun čísla se znaménkovým rozšířením a instrukce `movzx` – přesun čísla neznaménkově mají v AT&T ekvivalenty `movs`... a `movz`... které jsou navíc opatřeny suffixem, který udává velikost obou operandů: `bl` – byte → long, `bw` – byte → word, `bq` – byte → quadruple word, `wl` – word → long word , `wq` – word → quadruple word, `lq` – long → quadruple word.

5. Vkládání kódu assembleru v GCC

Klíčové slovo pro vkládání assembleru je `__asm__` nebo `asm`. Pokud se uveden `__asm__ __volatile__` kompilátor se nebude snažit kód v tomto bloku optimalizovat.

```
asm __volatile__("instrukce\n"
  : předávání výstupních proměnných                (volitelné)
  : předávání vstupních proměnných                 (volitelné)
  : seznam modifikovaných registrů a paměti        (volitelné)
);
```

GCC posílá instrukce assembleru jako řetězec, takže pro správné formátování je nutné přidat na konec `\n`. Proto aby se v řetězci předešlo dezinterpretaci mezi formátovacími značkami (`%u`, `%d`, `%0`, `%1`, ...) a intrukcí s prefixem „`%`“, zapisuje se znak procento jako „`%%`“.

Modifikátory:

- `=` – operand pouze pro zápis
- `+` – operand pro čtení i zápis
- `&` – pro vstup a výstup se alokuje jiný registr
- `m` – operand v paměti
- `o` – operand v paměti s adresací posunu
- `r` – obecný registr
- `g` – libovolný obecný registr, adresa nebo přímý operand
- `0, 1, 2, ...9` –
- `p` – operand je platná adresa

Specifikace registrů i386:

- `R` – Některý ze základních celočíselných registrů (`%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%ebp`, `%esp`) i
- `a` – registr `%eax`
- `b` – registr `%ebx`
- `c` – registr `%ecx`

- `d` – registr `%edx`
- `S` – registr `%esi`
- `D` – registr `%edi`
- `A` – registrový pár `%edx:%eax` (typ `long long`)
- `f` – některý ze zásobníkových FP registrů 80387
- `t` – vrchol zásobníku 80387, registr `%st0`
- `u` – registr pod vrcholem zásobníku 80387 `%st1`
- `y` – libovolný MMX registr
- `x` – libovolný SSE registr
- `I` – celočíselná konstanta v rozsahu 0 až 31, pro 32bitový posuv
- `J` – celočíselná konstanta v rozsahu 0 až 63, pro 64bitový posuv
- `K` – znaménková 8bit. celočíselná konstanta
- `M` – posuv 0, 1, 2 nebo 3 pro instrukci `lea`
- `N` – neznaménková 8bit. celočíselná konstanta (pro instrukce `in` a `out`)
- `G` – 80387 konstanta s plovoucí čárkou
- `C` – standardní konstanta SSE s plovoucí čárkou

5.1. Příklady

Pro vstup i výstup je použita jedna instrukce:

```
asm ("incl %0" :"=a" (var) : "0" (var));
```

6. Literatura

[1] Dandamundi, S. P.: Guide to Assembly Language Programming in Linux, Springer, 2005.
[2] Blum, R.: Professional Assembly Language, Wiley Publishing, Inc., 2005.
[3] Leiterman, J.: 32/64-BIT 80x86 Assembly Language Architecture, Wordware Publishing, 2005.
[4] Leiterman, R. S. J.: Using the GNU Compiler Collection, GNU Press, Free Software Foundation, 2008.
[5] Chisnall, D. (2007). Vector Programming with GCC. informIT.