1 Úvod

Architektura ARM (Advanced Risc Machines) v dnešní době představuje velmi efektivní způsob použití jednočipových RISC mikrokontrolérů v rozmanitých zařízeních. Parametry jako rozměry čipu (cca 0.5-4mm^2), spotřeba na 1MHz (cca 0.15-0.3 mW/MHz) a napájení jádra (cca 1.2-5V) jsou hlavním důvodem použití těchto procesorů v celém spektru moderní elektroniky.

2 Architektura ARM

V současné době architektura ARM představuje 75% usazených RISC mikroprocesorů na trhu. Partneři ARMu již vyrobili díky vlastnostem, jako je kopatibilita jednotlivých verzí[1], dobrý výkon na 1 watt a hustotu kódu přes miliardu mikroprocesorů s jádrem ARM.
Rozšířením instrukční sady ARM se vyvinula instrukční sada Thumb. Ta vyžaduje pouze 16bitovou šířku operačního kódu instrukce a tím i poloviční aktivitu datové sběrnice, což má za následek nižší výkonovou spotřebu a samozřejmě úsporu paměti programu.
Sada instrukcí Thumb je podmnožina instrukcí ARM, zabalená do poloviční šířky operačního kódu. Znamená to, že veškeré Thumb instrukce se pro vlastní vykonávání (exekuci) instrukčního slova rozbalí na 32 bitovou šířku - tedy na patřičnou instrukci sady ARM. Vývojáři tak mohou kombinovat instrukční sadu ARM a Thumb tak, jak je možné pro úsporu paměti a času. Vlastní rozbalení Thumb instrukce na instrukci ARM nepotřebuje žádný instrukční cykl navíc.
Stojí za zmíňku, že pro zlepšení výkonu a úspory paměti byla na základě sadvy Thumb vyvinuta instrukční sada Thumb-2.
ARM Jazelle je technologie pro podporu Javy. V době několika operačních systémů dává volnost vývojářům spouštět Java programy na libovolné platformě.
S vývojem multimédií byla vyvinuta SIMD (Single-Instruction Multiple-Data) technologie, která umí efektivně používat velká datová pole. Jádra ARM s implementovanou podporou SIMD jsou používána v softwarových oblastech jako audio a video kodeky, kde se díky SIMD extezi může zvýšit výkon až 4x beze změny příkonu jádra.
Jádro ARM může bý také rozšířeno o některé 16 nebo 32bitové aritmetciké operace tedy o DSP rozšíření. Opět se zde šetří energie, čas a koplikovanost systému s dvěmi jádry (řízení + DSP). Aplikace jádra ARM s rozšířením

[1] S tím i možnost použít návrh programu z dřívější doby

DSP zahrnují např. zařízení pro ukládání velkých dat (řízení HDD) a automatické navádění satelitu.
Technologie ARM TrustZone poskytuje zabezpečení pro otevřené operační systémy - tzv. open OS. Zaručuje, že data která zákazník využívá jsou řádně zabezpečena. Typickým představitelem je GSM bankovnictví a placené GSM služby.
Technologie ARM Intelligent Energy Manager (IEM) řeší pomocí složitějších adaptivních algoritmů dynamické předpovídání vytíženosti CPU a dle toho nastaví jeho výkon. Znamená to výraznou úsporu el. energie, která se uplatní hlavně v bateriemi napájených zařízeních, jako jsou mobilní telefony.
Pro multiprocesorový systém jádra byla vyvinuta architektura ARM OptimoDE. Jedná se o VLIW[2] datově konfigurovatelnou architekturu zaměřenou na non-stop zpracování dat. OptimoDE je zatím jednou z nejlepších kombinací pokročilé úrovně procesorů, robustní funkčnosti, spotřeby a integrace na křemík (klidně i spolu s jinými jádry ARM).

3 Jádra CPU archtektury ARM

ARM nabízí 16/32bitová RISC jádra zabudovaná do celkového systému, která se dělí dle rodin: ARM7, ARM9, ARM9E, ARM10, ARM11 a SecurCore. Každá z těchto rodin je výkonná s malou spotřebou. Výběr rodiny záleží na konkrétní aplikaci, kterou je nutno navrhnout.

3.1 ARM7 Thumb

Tuto rodinu představují mikroprocesorová jádra ARM7TDMI a ARM7TDMI-S, procesorová makrobuňka s vlastni cache ARM720T a ARM7EJ-S jádro s ARM Jazelle rozšířením. Sada instrukcí Thumb je optimalizována pro použití rozsáhlejších SoC (System on Chip) návrhů.
Pro představu jádro ARM7TDMI zabírá pouze 0.53mm^2 v 0.18μ CMOS technologii a spotřeba je kolem 0.25mW/MHz.
ARM720T představuje konpletní procesor s 8kB chache, write bufferem a funkcemi pro správu paměti MMU (Memory Managment Unit). Tato podpora virtuální paměti podporuje operační systémy, jako je Linux, Symbian OS a Windows CE.
Výrobci používají rodinu ARM7 pro výrobu síťových a bezdrátových zařízení (modemy, síťové karty, mobilní telefony ‚PDA[3]). Vývojáři komentují rodinu

[2]Very Long Instruction Word - množina různých instrukcí je za kódována do jedné kopmplexní instrukce

[3]z angl. Personal Digital Assisntant

ARM7 jako cílenou pro použití v budoucích bezdrátových "smart - chytrých" zařízeních.
V současné době mají na rodinu ARM7 licenci následující firmy: ATMtek, Agilent, AMI Semiconductor, Epson, Ericsson, Fujitsu, Global UniChip, Intel, Kawasaki, Lucent, Micronas, Mitsubishi Electric, Mobilian, Oak Technology, Parthus, Sanyo, Silicon Wave, SiS, ST Microelectronics, Toshiba, Triscend a Yamaha.

3.2 ARM9 Thumb

Rodinu ARM9 tvoří procesorové jádro ARM9TDMI a procesorové makrobuňky s cache ARM920T a ARM940T. Výkon je 220MIPS na taktovací frekvenci 200MHz a 0.18μ nebo 0.13μ technologii. Typická velikost čipou je pro ARM920T jádro s 0.13μ technologií 2.1mm^2 včetně cache a spotřeba pro 1.2V napájení je cca 0.2 mW/MHz.
Všechny jádra z rodiny ARM9 podporují Thumb instrukční sadu a jsou zpětně komptibilní s rodinou ARM7. Typická zařízení např: automatické řízení pohybu, automatizace, bezpečnostní systémy, set-top boxy[4], high-end tiskárny, PDA, "smart" telefony, kódování-dekódování MP3 a MPEG4.
Licence: ADMtek, AMI Semiconductor, Matsushita, OKI, Resonext, TSMC a ZTEIC.

3.3 Rodina ARM9E

Tato rodina sloučených procesorů je tvořena procesorovými makrobuňkami s cache ARM926EJ-S, ARM946E-S a ARM966E-S. Díky rozšíření o DSP aritmetické operace je výhodná všude tam, kde je potřeba sloučení DSP a řídícího CPU. Rodina ARM9E coby jednočipové řešení v aplikacích normalně používajicích víceprocesorový systém tak šetří energií a cenou výrobku. Jádra krom DSP rozšíření instrukční sady disponují samozřejmě rozšířenou sadou instrukcí Thumb. Jádro ARM9EJ navíc disponuje ARM Jazelle technologií pro HW podporu Javy.
ARM vyvíjí též floating-point koprocesory. ARM VFP9-S je vektorový floating-point koprocesor, který se vyznačuje vysokým výkonem, nízkou spotřebou a nízkými rozměry (cca 1.5mm^2).
Pro názornost koprocesor ARM VFP9-S se skládá z cca 95000 logických hradel, jádro ARM966E-S pak z 65000 hradel. V 0.18$^\mu$ technologii bude implementace jádra ARM966E-S včetně 16kb instrukční a 16kb datové cache

[4]Krabička pro využívání některých služeb internetu pomocí televize - umístění na televizi, proto ten název

zabírat cca 4mm^2. ARM VFP9-S koprocesor pak nepřidá více než 1.5mm^2. Uplatnění rodiny ARM9E je pak v aplikacích jako je řízení HDD čí DVD, kódování řeči, hans-free řešení rozhraní, ABS[5], modemy, PDA, "smart" telefony, MP3 přehrávače, řečové rozpoznávání a syntéza řeči.
Licence: Agilent, Fujitsu, Intel, Intersil, Lucent, LSI Logic, Marvell, NEC, Oak Technology, Pixim, Philips, PrairieComm, Sanyo, Samsung, ST Microelectronics, Toshiba a TSMC.

3.4 Technologie StrongARM a XScale

Procesory StrongARM představují ideální řešení pro přenosnou komunikační a spotřební elektroniku. Tyto procesory, jež ARM vyvinul společně s Digital Equipment nyní vyrábí Intel a lze se s nimi setkat v palmtopech a PDA, např: Compaq iPAQ H3600 Pocket PC, the Hewlett Packard Jornada Handheld PC, Java technology-based Palmtop computers ...
Intel nabízí StrongARM v podobě základního procesoru SA110 a procesorů pro malá zařízení velikosti palmtopů SA1110 a SA1111. Na začátku nového tisíciletí Intel ohlásil blízké uvedení na trh mikroarchitektury XScale, která krom standartní Thumb sady instrukcí obsahuje i DSP rozšíření instrukční sady. Mikroarchitekturu XScale uvedl na trh v únoru 2002 v podobě procesorů PXA250 a PXA210. PXA250 je taktována hodinovým kmitočtem až 400MHz, PXA210 pak 200MHz.

[5]Anti-lock Braking Systems

4 C pro ARM efektivně

4.1 Úvod

C překladače pro instrukční sady ARM a Thub jsou schopné produkovat opravdu kvalitní strojový kód. Nelze však zajistit vysokou efektivnost výsledného kódu bez seznámení programátora alespoň se základními vlastnostmi struktury procesoru ARM. Tato část je detailně popsána v [2] a popisuje názorně na příkladech jednotlivé případy, na kterých může programátor zvýšit rychlost a snížit velikost svého programu.

4.2 Nastavení překladače

4.2.1 Volba jádra procesoru

Kvůli co nejlepší optimalizaci kódu na instrukce které dané jádro podporuje je tento parametr nutné nastavit. Například `-proc ARM7` pro rodinu ARM7[6] nebo `-proc StrongARM1` pro procesory SA11xx.
Parametrem `-arch N` se nechá zvolit nižší architektura než 7. Je výhodné volit 4, resp 4T pokud není známo na kterém jádře ARM program poběží.

4.3 Dělení

Instrukční sada ARM neposkytuje instrukci pro celočíselné dělení. Dělení je tak implementováno voláním funkcí (`__rt_sdiv` pro znaménkové a `__rt_udiv` pro neznaménkové dělení) standartní C knihovny. S ohledem na velikosti hodnot čitatele a jmenovatele 32-bitové dělení zabírá 20 + 4.3N cyklů (konstantní čas plus čas na každý bit dělení).
Protože je dělení časově náročnou operací, je velmi vhodné se jí vyhnout všude, kte je to možné. Např. `(x/y)>z` může být přepsáno na `x>(z*y)` pokud je `y` kladné.

4.3.1 Dělení a zbytek

V některých případech potřebujeme celočíselný podíl (x/y) a zbytek (x%y). Pro jedno volání dělicí rutiny je nutné vyjádřit oba operátory "spolu". Například:

[6]Nutno poznamenat, že u rodiny ARM7 se podpora thumb instrukcí zapíná parametrem `-proc ARM7T`, což může být díky poloviční šířce operačního kódu instrukce nmohem efektivnější

```
int combined_div_mod (int a, int b)
{ return (a / b) + (a % b);
}
```

Se přeloží jako:

```
combined_div_mod
    STMDB sp!,{lr}
    MOV a3,a2
    MOV a2,a1
    MOV a1,a3
    BL __rt_sdiv
    ADD a1,a1,a2
    LDMIA sp!,{pc}
```

4.3.2 Dělení a zbytek mocninou dvěma

Pokud je dělitel mocnina dvou, překladač použije posuv pro dělení. Proto je dobré kde to jde volit např. 128 místo 100. Následující kousek programu pro znaménkové a neznaménkové dělení ukazuje díky vyhnutí se volání externí funkce jednoduchost algoritmu:

```
typedef unsigned int uint;
uint div16u (uint a)
{ return a / 16;
}
int div16s (int a)
{ return a / 16;
}
```

Se přeloží jako:

```
div16u
    MOV a1,a1,LSR #4
    MOV pc,lr div16s
    CMP a1,#0
    ADDLT a1,a1,#&f
    MOV a1,a1,ASR #4
    MOV pc,lr
```

4.3.3 Dělení a zbytek pro modulo aritmetiku

Vše je nejlépe vidět na následujících příkladech čítačů minut nebo hodin:

```
uint counter1 (uint count)
{ return (++count % 60);
}

uint counter2 (uint count)
{ if (++count >= 60) count = 0;
return (count);
}
```

Se přeloží jako:

```
counter1
    STMDB sp!,{lr}
    ADD a2,a1,#1
    MOV a1,#&3c
    BL __rt_udiv
    MOV a1,a2
    LDMIA sp!,{pc}

counter2
    ADD a1,a1,#1
    CMP a1,#&3c
    MOVCS a1,#0
    MOV pc,lr
```

Odtud je vidět, že je opravdu nejlepší se dělení vyhnout, pokud existuje jiná cesta.

4.3.4 Dělení konstantou

V tomto případě stojí zato si napsat vlastní funkci, která je pro danou konstantu rychlejší, než rutina ze standartní knihovny. ARM C knihovna má standartně implementováno dělení 10 pro obě znaménkové reprezentace. V případě použití pevné řádové čárky při reprezentaci čísel je pak vhodné použít násobení převrácenou honotou dělitele. Instrukce násobení je v součástí instrukční sady.
Vygenerování programu pro dělení konstantou pro obě instrukční sady je možné vyhleadt na stránkách společnosti ARM `examples/explasm/div.c` a `examples/thumb/div.c`

4.4 Podmíněné vykonávání instrukcí

Všechny instrukce v instr. sadě ARM jsou podmíněné. Mají na to vyhrazené první čtyři bity v 32 bitovém instrukčním slově. Ty pokud odpovídají nastavení prvním čtyřem podmínkovým flag bitům v registru CPSR[7], instrukce se vykoná. Typické použití začíná porovnávací instrukcí. Např.:

```
CMP x, #0
MOVGE y, #1
MOVLT y, #0
```

Výše uvedený příklad ušetří 2 skokové instrukce a tím přibližně 2.5 ARM7 cyklů.
Podmíněné vykonávání je často používáno ve větvení programu (if, else), při výpočtu výrazů s relacemi ($<, ==, >$ apod.), nebo s booleovskými operátory (&&,!, apod.). Podmíněný výkon instrukce se neprovádí, pokud se někde v těle nachází skok do podprogramu.
Je tedy velmi výhodné mít co nejjednodušší těla příkazů `if` a `else`. Následující příklad ukazuje jak překladač používá podmíněné vykonávání příkazů, když jsou podmínky seskupeny:

```
int g(int a, int b, int c, int d)
{ if (a > 0 && b > 0 && c < 0 && d < 0) /* grouped conditions */
return a + b + c + d;
return -1;
}
g
    CMP a1,#0
    CMPGT a2,#0
    BLE |L000024.J4.g|
    CMP a3,#0
    CMPLT a4,#0
    ADDLT a1,a1,a2
    ADDLT a1,a1,a3
    ADDLT a1,a1,a4
    MOVLT pc,lr
|L000024.J4.g|
    MVN a1,#0
    MOV pc,lr
```

[7]Current Program Status Registers

4.5 Booleovské výrazy

4.5.1 Kontrola pozice

Běžně jde o to zjistit, zda se nějaká proměnná nachází uvnitř dané oblasti. Např. kontrola, zda se grafický bod nachází uvnitř daného okna:

```
bool PointInRect1(Point p, Rectangle *r)
{ return (p.x >= r->xmin && p.x < r->xmax &&
      p.y >= r->ymin && p.y < r->ymax);
}
```

```
PointInRect1
    LDR a4,[a3,#0]
    CMP a1,a4
    BLT |L000034.J5.PointInRect1|
    LDR a4,[a3,#4]
    CMP a4,a1
    BLE |L000034.J5.PointInRect1|
    LDR a1,[a3,#8]
    CMP a2,a1
    BLT |L000034.J5.PointInRect1|
    LDR a1,[a3,#&c]!
    CMP a2,a1
    MOVLT a1,#1
    MOVLT pc,lr
|L000034.J5.PointInRect1|
    MOV a1,#0
    MOV pc,lr
```

Ukažme rychlejší možnost: Výraz (`x` >= `min` && `x` < `max`) může být transformován na (`unsigned`)(`x` − `min`) < (`max` − `min`), což je extra výhodné, pokud je `min` nulové.

```
bool PointInRect2(Point p, Rectangle *r)
{ return ((unsigned) (p.x - r->xmin) < r->xmax &&
      (unsigned) (p.y - r->ymin) < r->ymax);
}
PointInRect2
    LDR a4,[a3,#0]
    SUB a1,a1,a4
    LDR a4,[a3,#4]
    CMP a1,a4
```

```
    LDRCC a1,[a3,#8]
    SUBCC a1,a2,a1
    LDRCC a2,[a3,#&c]!
    CMPCC a1,a2
    MOVCS a1,#0
    MOVCC a1,#1
    MOV pc,lr
```

Pozn: Dnešní verze překladačů by již měly dělat tuto optimalizaci automaticky

4.5.2 Porovnávání s nulou

Příznakové bity - flagy ARMu se vždy nastaví po instrukci CMP (compare). Mohou být též nastaveny i jinými instrukcemi jako MOV, ADD, AND, MUL, což jsou základní aritmetické a logické instrukce (instrukce pro *dataprocessing*). Pokud operační kód instrukce dataprocessingu má nastaven bit pro ovlivňování flagů (za jméno instrukce se přidá písmeno S), N a Z flagy jsou nastavené stejně jako by byl výsledek porovnán s nulou. N, resp. Z flag indikuje záporný, resp. nulový výsledek. Například

```
ADD R0, R0, R1
CMP R0, #0
```

vytvoří stejné N a Z flagy jako:

```
ADDS R0, R0, R1
```

N a Z flagy korespondují se znaménkovými relačními operátory `x` < 0, `x` $>= 0$, `x` $== 0$, `x`$! = 0$ a neznaménkovými `x` $== 0$, `x`$! = 0$ (nebo `x` > 0) v C.
Při každém použití relačního operátoru generuje kompilátor instrukci CMP. Pro výše uvedené operátory platí v případě předcházení aritmetické operace před relací výjímka. Například:

```
int g(int x, int y)
{ if (x + y < 0)
    return 1;
  else
    return 0;
}
g
    ADDS a1,a1,a2
    MOVPL a1,#0
    MOVMI a1,#1
    MOV pc,lr
```

Jazyk C nemá žádný koncept pro flagy carry a overflow, není tak možné tyto flagy testovat přímo bez použití inline assembleru. Existuje však možnost jak to obejít. Například:

```
int sum(int x, int y)
{ int res;
  res = x + y;
  if ((unsigned) res < (unsigned) x) /* carry set? */
    res++;
    return res;
}
sum
    ADDS a2,a1,a2
    ADC a2,a2,#0
    MOV a1,a2
    MOV pc,lr
```

4.6 Smyčky

Smyčky jsou často používanou záležitostí ve většině programů. Většina času určená výkonu programu je právě pohlcena uvnitř smyček, a proto stojí za to věnovat pozornost na jejich efektivní psaní.

4.6.1 Ukončení smyčky

Pokud není rozvážně uděláno ukončení smyčky, může dojít k významnému prodloužení smyčky a tím i několikanásobně celého programu. Dobré je psát smyčky čítající k nule s jednoduchou podmínkou ukončení. Následující příklady počítají n!, první používá inkrementující smyčku, druhá dekrementující.

```
int fact1 (int n)
{ int i, fact = 1;
  for (i = 1; i <= n; i++) fact *= i;
  return (fact);
}
int fact2 (int n)
{ int i, fact = 1;
  for (i = n; i != 0; i--) fact *= i;
  return (fact);
}
```

```
fact1
    MOV a3,#1
    MOV a2,#1
    CMP a1,#1
    BLT |L000020.J5.fact1|
|L000010.J4.fact1|
    MUL a3,a2,a3
    ADD a2,a2,#1
    CMP a2,a1
    BLE |L000010.J4.fact1|
|L000020.J5.fact1|
    MOV a1,a3
    MOV pc,lr
fact2   MOVS a2,a1
    MOV a1,#1
    MOVEQ pc,lr
|L000034.J4.fact2|
    MUL a1,a2,a1
    SUBS a2,a2,#1
    BNE |L000034.J4.fact2|
    MOV pc,lr
```

Jak je vidět, drobná změna v `fact1` potřebná k vytvoření `fact2` způsobí náhradu instrukcí ADD a CMP za jedinou instrukci SUBS.
Navíc proměnná `n` díky její absenci není třeba ukládat, což vede na mnohem efektivnější kód.
Tato technika nadefinování si předem počtu operací a následná dekrementace je také aplikována na příkazy `while` a `do`.

4.6.2 Rozbalení smyčky

Pro dosažení větší výkonnosti algoritmu mohou být jednoduché smyčky *rozbaleny* na úkor velikoti programu. Čítač smyčky je tak několikrát méně častěji pozměňován a tím se vykoná i odpovídající počet skoků. ARM překladače zatím automaticky rozbalování smyček nedělají, a tak je vhodné použít rozbalení ve zdrojovém kódu.

Zjištění počtu nastavených bitů ve slově První rutina počítá nastavené bity jednotlivě, druhá pak čtyři v cyklu najednou.

```
int countbit1(uint n)
{ int bits = 0;
```

```
  while (n != 0)
  {
    if (n & 1) bits++;
    n >>= 1;
  }
  return bits;
}

int countbit2(uint n)
{ int bits = 0;
  while (n != 0)
  {
  if (n & 1) bits++;
  if (n & 2) bits++;
  if (n & 4) bits++;
  if (n & 8) bits++;
  n >>= 4;
  }
  return bits;
}
```

V prvním případě program zabírá 9 instrukcí, ve druhém 15. Přesto ale druhý program vykoná méně skokových instrukcí, což vede na časově nenáročnější výsledný kód.

4.7 Příkaz switch

Pokud má příkaz switch *hustý* charakter, překladač použije vyhledávací tabulku pro odskok na danou adresu požadované části kódu. Příkaz switch má hustý charakter pokud počet poloh tohoto přepínače je větší než polovina rozsahu, vyčleněného maximální a minimální hodnotou podmínky.
Pokud switch nemá hustý charakter, překladač přepínač rozvětví na jednotlivé části. Proto pro úsporu velikosti výsledného kódu je vhodné používat příkaz switch s co největší mírou hustoty.

4.7.1 Příkaz switch a vyhledávací taulky

Příkaz switch se obvykle používá z následujících důvodů:

- Volání jedné z několika funkcí
- Nastavení proměnné (např. zjednodušené matematické funkce)

- Vykonání jedné z několika částí programu

Pro je hustý počet případů ve výše uvedených možností použití příkazu switch je vhodná efektivnější imlpementace pomocí vuhledávací tabulky. Následující příklady demonstrují zpětný překlad podmínkového kódu do řetězce:

```
char * ConditionStr1(int condition)
{
    switch(condition)
    {
        case 0: return "EQ";
        case 1: return "NE";
        case 2: return "CS";
        case 3: return "CC";
        case 4: return "MI";
        case 5: return "PL";
        case 6: return "VS";
        case 7: return "VC";
        case 8: return "HI";
        case 9: return "LS";
        case 10: return "GE";
        case 11: return "LT";
        case 12: return "GT";
        case 13: return "LE";
        case 14: return "";
        default: return 0;
    }
}
char * ConditionStr2(int condition)
{
    if ((unsigned) condition >= 15) return 0;
    return
      "EQ\0NE\0CS\0CC\0MI\0PL\0VS\0VC\0HI\0LS\0GE\0LT\0GT\0LE\0\0" \
      + 3 * condition;
}
```

První funkce zabírá 240 bytů, druhá pouze 72.

4.8 Ukládání dat do registrů

Jedná se o důležitou metodu jak zefektivnit výsledný program. Jedná se o alokaci lokálních proměnných do registrů oproti standartní alokaci do paměti. Výhodou je pak výrazné zlepšení v rychlosti a velikosti výsledného kódu.

4.8.1 Typy proměnných ukládatelných do registru

Veškeré celočíselné typy, pointery a typy s plovoucí řádovou čárkou lze ukládat do registrů. Ukládat proměnné do registrů lze pokud:

- se jedná o lokální proměnnou a
- její adresa není není přiřazena jiné proměnné

Pole v struktuře může být alokováno do registru, pokud:

- je deklarováno lokálně nebo jako parametr funkce a
- struktura není přiřazena jako výsledek funkce a
- adresa struktury ani žádného jiné prvku uvnitř není nikdy přiřazena jiné proměnné.

Pro co možná největší použití alokace proměnných do registrů je potřeba se snažit:

- vyhnout se pointerům na lokální adresy
- omezit globální proměnné
- nepoužívat pointery na pointery (řetězení pointerů)

4.8.2 Pointery na lokální adresy

Jsou potřeba, například pokud je potřeba parametr funkce. V tomto případě nelze alokovat proměnnou do registru, řešením je však vytvoření kopie dané proměnné. Vše je ukázáno na příkladech `test1`, kde se jedná o tradiční způsoba `test2`, kde pomocí `dummy` proměnné umožníme alokaci proměnné `i` přímo do registru.

```
void f(int *a);
int g(int a);

int test1(int i)
{  f(&i);
   /* now use  i  extensively */
   i += g(i);
   i += g(i);
   return i;
}
```

```
int test2(int i)
{   int dummy = i;
    f(&dummy);
    i = dummy;
    /* now use  i  extensively */
    i += g(i);
    i += g(i);
    return i;
}
test1
    STMDB sp!,{a1,lr}
    MOV a1,sp BL f
    LDR a1,[sp,#0]
    BL g
    LDR a2,[sp,#0]
    ADD a1,a1,a2
    STR a1,[sp,#0]
    BL g
    LDR a2,[sp,#0]
    ADD a1,a1,a2
    ADD sp,sp,#4
    LDMIA sp!,{pc}
test2
    STMDB sp!,{v1,lr}
    STR a1,[sp,#-4]!
    MOV a1,sp
    BL f
    LDR v1,[sp,#0]
    MOV a1,v1
    BL g
    ADD v1,a1,v1
    MOV a1,v1
    BL g
    ADD a1,a1,v1
    ADD sp,sp,#4
    LDMIA sp!,{v1,pc}
```

První funcke alokuje `i` do stacku, přičemž potřebuje celkem 4 paměťové přístupy na `i`, druhá pak potřebuje 2 pam. přístupy pro `dummy` a žádný pro `i`.

4.8.3 Problematika globálních proměnných

Globální proměnné se nikdy nealokují do registrů (pokud není použit parametr `__global_reg`).Globální proměnné lze zaměmit, použitím pointerů, či voláním funkcí. Překladač tak nemůže mít prozatímě hodnotu globální proměnné v registru a používá instrukce pro práci s pamětí při každém použití globální proměnné. Proto je důležité nepoužívat globální proměnné uvnitř časově náročných smyček.
Pokud funkce přesto používá globální proměnnou, je výhodné dočasně zkopírovat globální proměnné do lokálních. Samozřejmě je to možné udělat pouze v případě, že kopírovaná globální proměnná není použita žádnou z volaných funkcí. Například:

```
int f(void);
int g(void);
int errs;
void test1(void)
{ errs += f();
  errs += g();
}
void test2(void)
{ int localerrs = errs;
  localerrs += f();
  localerrs += g();
  errs = localerrs;
}
test1
    STMDB sp!,{v1,lr}
    BL f LDR v1,[pc, #L00002c-.-8]
    LDR a2,[v1,#0]
    ADD a1,a1,a2
    STR a1,[v1,#0]
    BL g
    LDR a2,[v1,#0]
    ADD a1,a1,a2
    STR a1,[v1,#0]
    LDMIA sp!,{v1,pc} L00002c DCD |x$dataseg|
test2   STMDB sp!,{v1,v2,lr}
    LDR v1,[pc, #L00002c-.-8]
    LDR v2,[v1,#0]
    BL f
    ADD v2,a1,v2
```

```
    BL g
    ADD a1,a1,v2
    STR a1,[v1,#0]
    LDMIA sp!,{v1,v2,pc}
```

Funkce`test1` musí tak používat instrukce LDR a STR při každé inkrementaci globální proměnné `errs`, zatímco `test2` uchovává proměnnou `localerrs` v registru.

4.8.4 Pointery na pointery

Často se to používá k získání informace uvnitř struktur. Např. běžný kód pro 3 rozměrný prvek:

```
typedef struct { int x, y, z; } Point3;
typedef struct { Point3 *pos, *direction; } Object;
void InitPos1(Object *p)
{ p->pos->x = 0;
  p->pos->y = 0;
  p->pos->z = 0;
}
```

Pro každé přiřazení se zde musí `p->pos` znova získávat. Vylepšený kód má `p->pos` v lokální proměnné:

```
void InitPos2(Object *p)
{ Point3 *pos = p->pos;
  pos->x = 0;
  pos->y = 0;
  pos->z = 0;
}
```

Jinou možností je zařadit `Point3` strukturu do struktury `Object`. Takto se vyhneme pointerům úplně.

4.8.5 Délka života proměnných

V architektuře ARM můžeme použít až 14 registrů. V hardwarové FPU se nachází 8 registrů pro pohyblivou řádovou čárku (floating-point). V případě, že je použita softwarová emulace FPU, floating-point registry jsou uloženy v celočíselných ARM registrech.
Překladače umožňují tzv. *live-range splitting*, kde může být proměnná uložena do různých registrů, či paměti v různých částech funkce. *live-range* je

definován jako veškeré příkazy meziposledním přiřazením do proměnné a posledním použitím před další přiřazením nové hodnty. V *live-range* je hodnota proměnné vždy platná, mezi jednotlivými *live-ranges* ovšem ne, a tak je možné registr použít pro jinou proměnnou. Tak je překladači umožněno alokovat více proměnných do registrů.
Minimální počet registrů potřebných pro vyhovující proměnné je roven počtu překrývání *live-ranges* jednotlivých proměnných v každém bodě funkce. Pokud dojde k přesahu počtu registrů, je nutné dočasně proměnné ukládat do paměti. Teno proces se nazývá `spilling` - odlévání. Překladač tento proces nejdříve začíná používat u nejméně využitých proměnných, čímž minimalizije hardwarové nároky na spilling. Spillingu se dá vyhnout:

- Redukcí maximálního počtu lokálních proměnných. Nechá se toho docílit dělením funkcí na menší a jednodušší.
- Použitím třídy *register* pro často používané proměnné. Tím překladač dostává informaci, že proměnná bude často používána, a tak je alokována do registru s nejvyšší prioritou.

4.9 Typy proměnných

Překladač C podporuje základní typy `char`, `short` a `long` (znaménkové a neznaménkové), `float` a `double`. Pro zvýšení efektivity výsledného kódu záleží na výběru nejvhodnějšího typu.

4.9.1 Lokální proměnné

Je dobré se vyhybat proměnným typu `char` a `short` pokud to je možné. Pro tyto typy je totiž potřeba převodu na 16ti, resp. na 8mibitový formát. To se nazývá *znaménkové rozšíření (sign-extending)* pro znaménkové proměnné a *nulové rozšíření (zero-extending)* pro neznaménkové proměnné. Provádí se to logickým posuvem vlevo o 24, resp. 16 bitů a následný aritmetickým (logickým pro nulové rozžíření) posuvem doprava o stejný počet bitů.
Nevýhody použití typů `char` a `short` a způsob znaménkového, resp. nulového rozšíření ukazuje následující příklad:

```
int wordinc (int a)
{ return a + 1;
}
short shortinc (short a)
{ return a + 1;
```

```
}
char charinc (char a)
{ return a + 1;
}
wordinc
        ADD     a1,a1,#1
        MOV     pc,lr
shortinc
        ADD     a1,a1,#1
        MOV     a1,a1,LSL #16
        MOV     a1,a1,ASR #16
        MOV     pc,lr
charinc
        ADD     a1,a1,#1
        AND     a1,a1,#&ff
        MOV     pc,lr
```

4.10 Funkce

V základě je dobré pamatovat na jednoduché a kratší funkce, než na komplexní celky. Umožní to překladači efektivně realizovat různé optimalizace jako je např. alokace proměnné do registru.

4.10.1 Režie volání funkce

je díky rychlé sekvenci volání-návratu - `BL ... MOV pc, lr` (4 instr. cykly na StrongARM1 a 6 na ARM7) a multidatovým načítacím a ukládacím instrukcím velmi efektivní.
Ve standartní ARM proceduře volání funkcí je možné mít argumenty funkcí maximálně ve 4 registrech (typ `double` zabírá dva registry ostatní pak po jednom). Více argumentů je pak ukládáno do zásobníku a vyvoláváno uvnitř funkce.

```
int f1(int a, int b, int c, int d)
{ return a + b + c + d;
}
int g1(void)
{ return f1(1, 2, 3, 4);
}
int f2(int a, int b, int c, int d, int e, int f)
{ return a + b + c + d + e + f;
```

```
}
ing g2(void)
{ return f2(1, 2, 3, 4, 5, 6);
}
```

Ve funkci g2 je pátý a šestý parametr uložen ve stacku a obnoven v f2 se dvěmi paměťovými přístupy na parametr.

4.10.2 Minimalizace nákladů volání funkce

je možné dosáhnout, pokud:

- Se pokusíme zajistit 4 a méně parametrů funkcí. Není tak potřeba přístup do stacku.
- Potřebujeme-li přeci jenom více paramterů, ujistíme se, že 5tý a další argumenty nejsou používány častěji než 1.-4. argument.
- Použijeme jako argument pointer na strukturu namísto struktury samotné.
- Dáme přebytečné argumenty do struktury a její ukazatel předáme funkci.
- Minimalizujeme použití typů rozsahu dvou 32bitových slov. Týká se to typů `long long` a `double` (pokud je povolena softwarová emulace FPU).
- Vyhneme se instrukcím s proměnným počtem parametrů. Překladač pak všechny argumenty ukládá do stacku.

4.10.3 Použití __value_in_regs

Při potřebě návratu více hodnot z funkcí je vhodné použít struktury. Standartně je struktura navrácena ve stacku. Překladač poskytuje volbu `__value_in_regs`, která umožní strukturu do velikosti 4 slov navrátit v registrech. Docílí se tím snížení nároků na paměť a velikosti výsledného kódu.
Na následujícím příkladě se použitím `__value_in_regs` program zkrátí z 160 bytů na 52:

```
typedef struct { int hi; uint lo; } int64; // low word unsigned!

__value_in_regs int64 add64(int64 x, int64 y)
{       int64 res;
        res.lo = x.lo + y.lo;
```

```
        res.hi = x.hi + y.hi;
        if (res.lo < y.lo) res.hi++; // carry from low word
        return res;
}
void test(void)
{       int64 a, b, c, sum;
        a.hi = 0x00000000;
        a.lo = 0xF0000000;
        b.hi = 0x00000001;
        b.lo = 0x10000001;
        sum = add64(a, b);
        c.hi = 0x00000002;
        c.lo = 0xFFFFFFFF;
        sum = add64(sum, c);
}
add64
        ADDS    a2,a2,a4
        ADC     a1,a3,a1
        MOV     pc,lr
test
        STMDB   sp!,{lr}
        MOV     a1,#0
        MOV     a2,#&f0000000
        MOV     a3,#1
        MOV     a4,#&10000001
        BL      add64
        MOV     a3,#2
        MVN     a4,#0
        LDMIA   sp!,{lr}
        B       add64
```

Pokud je potřeba rutina v assembleru, která vrací více hodnot, volba `__value_in_regs` je efektivní způsob, jak tyto hodnoty předávat pouze pomocí registrů:

```
typedef struct
{
        int x, y, z;
} Point3;

/* this is an ARM assembler function which takes a coord in r0-r3
 * and returns a transformed coordinate in r0-r3
 */
```

```
__value_in_regs Point3 TransformCoord(Point3 a);
```

References

[1] : ARM overview product http://www.arm.com/miscPDFs/3823.pdf.

[2] Herout, P.: Učebnice jazyka C Nakladatelství KOPP České Budějovice 2001.

[3] Little, J., N., Franklin, G.,F., a kol.: Source codes of Signal Toolbox The MathWorks, Inc.1994.