Vyšší programovací jazyky pro mcu

Požadavky vpj na architekturu mcu, omezení a rozdíly vůči programování pro osobní počítače, optimalizace kompilátoru

Joseph Soukup – P4

Omezení a rozdíly vůči programování pro osobní počítače

Jazyk C pro mcu

- Je podporována norma jazyka C
- Omezené HW prostředky (velikost pamětí, hloubka a implementace zásobníku, ALU)
- Rozšíření díky speciálním instrukcím (bitové proměnné a instrukce) - u osobního PC musí být
    - SET, CLEAR, SET
    - Boolovské operace – nad některými bity je možné provádět logické operace
- implementovaná softwarově
- Absence OS

Datové typy

- Většina podporována
- Typ int je u 8/16bit MCU 16bitový a u 32bit MCU 32bitový
- Typ double u některých 8bit mcu není podporován
- Navíc typ bit pro bitové proměnné
- Klíčové slovo volatile pro SFR registry

Programové konstrukce

- Někdy nutná SW implementace zásobníku
- Omezené nebo zakázané rekurzivní volání funkcí

Absence OS

- Absence dynamického přidělování paměti (klíčové slovo New)
- Chybí kontrola přístupu do paměti
- Přímý přístup k HW (SFR)
- Vlastní řešení multitaskingu
- Vlastní řešení ukládání kontextu při volání přerušení

Požadavky VPJ na architekturu MCU

Požadavky hll na architekturu mcu

- Cílem je snížení kódové nadbytečnosti při používání vyšších programovacích jazyků
- Co nejefektivnější implementace typických konstrukcí jako jsou datové struktury, pole, matematické výpočty, větvení, volání podprogramů

Několik akumulátorů (pracovní registr)

- Aby nedocházelo ke zbytečnému zdlouhavému přelévání obsahu registrů pro aritmetické operátory

Krátký instrukční cyklus

- Cílem je vystačit s jedním cyklem na instrukci (Single Cycle Instructions) - Nejmenší počet taktů krystalu
    - Přečtení
    - Dekódování
    - Provedení instrukce s využitím architektury

Rozšířená podpora ukazatelů

- Přístupy k datům pomocí ukazatelům
- Speciální podpora 8, 16 a 24bitových ukazatelů
- Pre-dekrement a Post-inkrement při přístupech (X+, -X)

Dokumentace strana 395 (Instrukční sada)

- Poslední sloupec odkazuje na krátký instrukční cyklus
- První stránka udává 32 pracovních registrů

Indexování polí (array)

- Instrukce obsahuje jak adresu začátku pole, tak posun od začátku
- Přístup na prvky polí pomocí relativních adres (displacement – Y+q, Z+q)
    - Ukazatel ukazuje na začátek pole a offset slouží jako index (pro datové struktury a pole)

Paměťové ukazatele (memory pointer)

- Pro přístupy k datům by měly být k dispozici nejméně 4 datové ukazatele (pointery)
    - Data zdroj (source pointer)
    - Cíl (destination pointer)
    - Ukazatel na zásobník (stack pointer)
    - Datový segment

Šíření příznaku nuly (Zero-Flag Propagation)

- Nějaká instrukce zahrnuje v sobě výsledek předchozí operace
    - ADC – Add with Carry (Carry je výsledek přechozího sčítání)
    - SBC – Subtract with Carry
    - CPC – Compare with Carry
- Když porovnáváme 4bytové čísla na 1bytový ALU, ta stačí vědět, že se to v jednom bytu nerovnalo, a i kdyby se to ve zbylých rovnalo, tak výsledek bude false
- Podpora 16/32bitového odčítání/porovnání u 8bitových aritmeticko-logických jednotek (ALU)

Bitové proměnné

- Speciální bitové instrukce pro bitové proměnné, takže pro logické (booleovské) proměnné nejsou nutné náročné instrukce SET a CLEAR s použitím masek AND/OR
- Usnadňuje typy výpočtů (Stav jednotlivých pinů)
- Dokumentace 396 - BIT AND BIT-TEST INSTRUCTIONS

Kódy instrukcí delší než 8 bitů

- Instrukce včetně operandů je v paměti uložená na jednom místě (nemusíme přistupovat opakovaně)
- Tak je možno instrukci i operand přečíst v jednom hodinovém cyklu, není nutno je číst postupně

Registry v normální oblasti paměti

- Datová paměti u Atmega má na začátku blok 32bytů
    - Na ně jsou mapovány registry R0-R32
- Umístění registrů v normálním adresním prostoru paměti, takže na proměnné v registrech je možno obracet prostřednictvím ukazatelů jako v datové oblasti SRAM

Ukazatel na zásobník (stack pointer - HW)

- SW přístupný
- Tak je možno instrukci i operand přečíst v jednom hodinovém cyklu, není nutno je číst postupně

Podpora 16/32bitových dat (16/32 data support)

- Podporuje aritmetické nebo logické instrukce s vyšší, než je nativní bytová šířka
- U 8(16)bitového procesoru jestli ALU dokáže pracovat s 16(32)bitovými daty

Optimalizace kompilátoru

- Cílem optimalizací jsou minimální velikost programu a co největší rychlost
- Požadavky jdou proti sobě (Závislý na požadavcích nebo omezení např. paměti)
- Podle toho se vybírají určité optimalizace
- Dělí se na:
    - Závislé na HW
        - Musí být podpora ze strany architektury
    - Nezávislé na HW
        - Aplikované vždycky

Cíle optimalizace

- Optimalizace začíná u programátora návrhem programu a volbou reprezentace dat
    - Psát programy tak, abychom to kompilátoru co nejvíce usnadnili
- Zvýšení rychlosti programu
- Snížení velikosti programu a dat
- Způsob a stupeň optimalizace je možné volit v nastavení překladače

Optimalizace závislé na hardware

Registrované proměnné

- Automatické proměnné a parametry funkcí se, pokud možno co nejvíce umísťují do registrů
- Tím je přístup na ně efektivnější a nezabírají žádné paměťové místo v RAM

Optimalizace jednoduchým přístupem

- Přístupy na interní datové a bitové adresy jsou optimalizovány strojovými instrukcemi závislými na MCU, využívají se speciální posloupnosti strojových příkazů

Reorganizace kódu

- Je-li smyčka FOR efektivnější než programátorem použitá smyčka WHILE, kompilátor kód změní
- Nebo jestliže smyčka čítá nahoru a pak následuje dotaz <> 0, zkouší se, zda existuje příkaz procesoru pro ?:<>
- Pokud ne, smyčka se otočí a testuje se na=0

Optimalizace nezávislé na hardware

Zpracování konstant (constant folding)

- Výpočty, které obsahují konstanty, jsou v co největší míře prováděny již kompilátorem

Vyloučení opakujících se výpočtů nebo částí výrazů

- Vícenásobné (opakované) výpočty nebo stejné části výrazů uvnitř výrazu nebo funkce jsou pokud možno co nejvíce eliminovány a počítají se jen jednou, přičemž výsledek prvního výpočtu se ukládá do registru

Optimalizace skokových příkazů

- Meziskoky (skok na jiný skok) jsou odstraňovány tak, že se nahradí skokem na konečný cíl
- Zahrnuje to také volbu optimálního skokového příkazu v závislosti na délce skoku (16bitový nebo 32bitový příkaz)
- Vícenásobné skoky (IF) nahradí skokem přímo na cílovou adresu
- Kompilátor si může zvolit podle vzdálenosti, zda použije absolutní (úplná cílová adresa) nebo relativní (změna mezi současnou pozicí a cílovou pozicí) skok

Vyloučení mrtvého kódu

- Odstranění nepoužívaného pasivního zdrojového kódu z programu

Náhrada opakujících se úseků programu skoky

- Identické úseky kódu na různých místech v programu se vytvářejí jen jednou a zpracovávají se pomocí skokových příkazů

Negace skoků

- Testy pro podmíněné skoky se invertují, lze-li tím odstranit jiné skoky nebo nepoužívaný kód
- Někdy když zneguje podmínky, tak se můžeme zbavit jedné větvě nebo zjednodušit

Překrývání dat

- Datové segmenty funkcí jsou označeny jako staticky překrývané
- Spojovací program pak má možnost překrývat segmenty
- V některých knihovnách, či funkcí můžeme deklarovat lokální proměnné s klíčovým slovem static, tím je paměť přiřazena trvale, i když není dostupná ze všech částí programu
- Když kompilátor vyhodnotí, že může přidělenou paměť využívat vícenásobně aniž by to ovlivnilo běh programu, může paměťové místo sdílet mezi více části programu

Optimalizace plnění

- Redundantní příkazy plnění (zavádění) se odstraňují
- Vylučuje se tak nepotřebné zavádění dat a konstant z paměti
- Složitější operace se nahrazují jednoduššími, je-li tím možno ušetřit paměťové místo nebo čas běhu programu
- Do proměnné přidělíme hodnotu a potom se hodnota přepíše výsledkem jiné operace (na začátku vynulování), pokud zjistí, že její obsah není nikdy využíván, tak může počáteční přiřazení vynechat

Optimalizace jednoduchých smyček

- Jednoduché smyčky (kód, který není moc velký) se optimalizují z hlediska doby běhu tak, že se kód vysune ze smyčky
- Vezmeme kód ze smyčky a několikrát ho zkopírujeme za sebou
- Rychlost programu se zvýší, ale zároveň se zvětší paměťová náročnost

- Optimalizace obzvlášť >>horkých<< míst programu, většinou smyček (90% času v 10% kódu)

Rotace smyček

- V programových smyčkách se zamění uspořádání kódu, dosáhne-li se tím rychlejšího a efektivnějšího kódu
- Nesmí být na sobě závislé
- Pracování se stejnou proměnnou, která řídí opakování smyčky (for)

Optimalizace řídícího toku

- Výrazy typu switch – case (zkompilování kaskády if) se optimalizují a zjednodušují jako skokové tabulky nebo skokové řetězce
- Proměnné z různých oblastí paměti se zavádějí přímo do operace
- Pokud hodnoty v jednotlivých sekcí tvoří souvislý úsek, tak není nutné abychom testovaly jednotlivé hodnoty, ale můžeme skočit rovnou na správnou hodnotu