JPO - 2014/2015 - zimní semestr

Rozšíření zdrojového kodu mikroprogramovaného řadiče

Karel Rymeš

November 23, 2014

33. Programovatelný cyklický posuv vpravo

Posuňte cyklicky dvojité slovo uložené v paměti (způsobem "little-endian") na adrese uložené v registru S vpravo o počet pozic, který je dán hodnotou registru L. Výsledek uložte do paměti na adresu určenou v registru D. Nastavte příznaky.

1 Dokumentace

1.1 Upřesnění specifikace

V případě, že v registru L je na začátku 0, neprovede se žádné protočení a rovnou se nastaví flagy. Carry Flag a Overflow flag se pokaždé rovnají 0. Registr L se na začátku "vymaskuje" s 00011111, aby neprobíhalo zbytečně moc operací.

1.2 Dokumentace chování

Pro posuv doprava je v ALU jednotce vyhrazena operace s op. znakem 1. Její provedení spočívá v posunutí všech 16 bitů doprava o 1 pozici, tudíž vysunutí LSB (dá se odchytit v CF),a nasunutí stávájícího CF na MSB.

Tato operace se dá nádherně využit pro cyklický posuv vpravo, ale je třeba dbát na správné nastavování CF pro jednotlivé slova. Hodnota CF pro dané slovo a dané posunutí musí být vždy LSB druhého slova před posunutím. Tudíž v jednom cyklu je zapotřebí si nejdříve zjistit LSB např. horního slova, poté posunout dolní slovo s nasunutím zapamatovaného LSB z horního slova a zapamatovat si LSB dolního slova před posunutím, poté už jen stačí posunout horní slovo s nasunutím zapamatovaného LSB dolního slova. Tento cyklus se provádí, dokud není hodnota L == 0. Na konci se vyplatí stále držet v ALU dolní slovo, protože hned na začátku dalšího cyklu se budu ptát na jeho LSB. Tudíž ale před prvním provedení těla cyklu musím mít již v registru W uloženo dolní slovo. A též po provedení posledního cyklu je nutné z ALU ještě načíst horní slovo do registrů v datové části!

Ještě nastává jedna zrada, která je způsobená tím, že operace 1 je definována jako W shift right + CIN!. A pokud chceme správně nastavit flagy po operaci, musíme používat

cummulative zero, čímž pádem nám přijde ale carry flag na cin. A v případě, že by se zrovna z nějákého slova vysouvala do CF 1, tak by se nám objevila jako CIN a výsledek by zneplatnial. Tudíž flagy se musí nastavit až po celém cyklu. Průchod cyklem jsem zoptimalizoval tak, že nejprve se posouvá horní slovo a teprve poté až dolní slovo, abych po cyklu už musel nastavovat flagy jen za horní slovo, protože v cummulative zero modu využiji flagy nastavené na konci cyklu. Horní slovo musí být vyhodnoceno jako druhé, aby proběhlo správné nastavení sign flagu. To je jediné místo, kde se nám promítne způsob uložení v paměti "little-endian". Všude jinde na něm nesejde.

1.3 Popis instrukce

Kod instrukce: 10010001

1.3.1 Pseudokod

```
[S] -> U; S++
[S] -> T; S++

_____

MASK OUT L REGISTER WITH 00011111
CHECK L_ZERO
{
        true --> set_flags
        false --> next adress
}

_____

U -> W;
SETW0ASCARRYFLAG;

loop:
        W -> U;
        T -> W;
        WROR + SETW0ASCARRYFLAG;
        W -> T;
        U -> W;
        WROR + ECF;
        L--;
        check LZERO
        {
                true -->  write_pre; W -> U; + UCF
                false --> loop; stav: SETW0ASCARRYFLAG;
```

```
}
```

```
write_pre:
                    W -> U; UCF
                    T -> U; ECF
write:
        U -> [D]; D++
        T -> [D]; D++
```

1.4 Kvantitativní ukazatele

- dekodování - 3 mikroinstrukce
- natažení DWORDu z paměti - minimálně 12 mikroinstrukcí, ale záleží na rychlosti hlavní paměti
- vymaskování a kontrola L != 0 - 11 mikroinstrukcí
- v případě L != 0; "overhead" = příprava před prvním cyklem a flag setting - 3 mikroinstrukce
- v případě L = 0; overhead na flag setting - 2 mikroinstrukce
- samotné posouvání v cyklu - 9 mikroinstrukcí
- write DWORDu do paměti - minimálně 14 mikroinstrukcí

1.4.1 Celkové počty mikroinstrukcí

Pro k posunutí (obecně, bez vymaskování), kde $k > 0$, je potřeba $(3 + 12 + 11 + 3 + 14) + k * 9 = 43 + 9 * k$ mikroinstrukcí. V případě vymaskování je to $43 + (k \mod 32) * 9$ mikroinstrukcí. V případě, že je $k \mod 32 == 0$, je to ale pouze 42 mikroinstrukcí.

Pro k posunutí, kde $k == 0$, je potřeba $3 + 12 + 11 + 2 + 14 = 42$ mikroinstrukcí.

Počítejme, že uživatel bude používat všechny hodnoty v L od 0 do 255 se stejnou pravděpodobností. Tudíž střední hodnota provedených mikroinstrukcí (po vymaskování!) je:

$$8 * \sum_{i=1}^{31} (1/256) * (9 * i + 43) + \sum_{i=1}^{8} (1/256) * 42 = 182.47$$

mikroinstrukcí průměrně. Což je značné zlepšení oproti tomu, kdybych bral všechny nevymaskované hodnoty L, tak bych došel k diametrálně odlišné střední hodnotě:

$$\sum_{i=1}^{255} (1/256) * ((43 - 10) + i * 9) + (42 - 10) /256 = 1180.49$$

. Což je 6.5 krát více.
V nejhorší případě ($k \mod 32$) == 31 bude použito $43 + 31 * 9 = 322$ mikroinstrukcí.
V nejlepší případě ($k \mod 32$) == 0 bude použito 40 mikroinstrukcí.

2 Testování

2.1 Test 1

Registry před provedením instrukce
S = 0004
D = 0010
PC = 0000
L = 8
Obsah paměti před provedením instrukce
0000 = 91
0004 = FF
0005 = FF
0006 = 00
0007 = 00
Stav registrů po provedení
S = 0008
D = 0014
PC = 0001
SIGN FLAG: 1
AF: 1
OSTATNÍ: 0
Obsah paměti po provedení
0010 = FF
0011 = 00
0012 = 00
0013 = FF

2.2 Test 2

Registry před provedením instrukce
S = 0004
D = 0010
PC = 0000
L = FF
Obsah paměti před provedením instrukce
0000 = 91
0004 = 01
0005 = 00

0006 = 00
0007 = 00
Stav registrů po provedení
S = 0008
D = 0014
PC = 0001
Všechny flagy: 0
Obsah paměti po provedení
0010 = 02
0011 = 00
0012 = 00
0013 = 00

3 Závěr a poznatky

Úloha byla zajímává a pěkně zkonsolidovala mé znalosti z první poloviny semestru. Vzhledem k tomu, že jsem chtěl provést cyklus posuvů na co nejméně instrukcí, tak mně trochu pozlobil fakt, že operace 1 je W shift right + CIN a též, že CZM s sebou též nese CIN = CF. Další nápady na optimalizaci mě bohužel napadly až po dopsání práce. Přišel jsem na to, že v průměru ušetřil nějáké mikroinstrukce, když bych na začátku kontroloval, jestli je DWORD nula, nebo ne. Tudíž bych nemusel používat ani cummulative zero mode.