Principy počítačů

14. března 2017

Obsah

Varování

Tento dokument obsahuje spoustu chyb, jak v kódu, který jsem nedebugoval, tak v textu. Dále, veškerý kód by měl být spíše vnímán jako pseudokód. Má Cčkovou syntaxi, ale často, datové položky, kvůli kompatibilitě ze zadáním, používám jiné. Omluvte prosím občasné pravopisné chyby a překlepy.

22. 1. 2016

1

Předpokládejme sample rate $f = 44000Hz$, což je počet samplů za sekundu. Předpokládejme, že velikost samplu (=informace popisující "barvu"(výchylka u mechanického vlnění) zvuku) je 1B.
Do proměnné se nám vejde $\frac{1}{44000} \cdot 88199$ sekund zvuku (cca 2 sekundy), protože $f = \frac{1}{T}$, kde T je perioda.

2

TBD

3

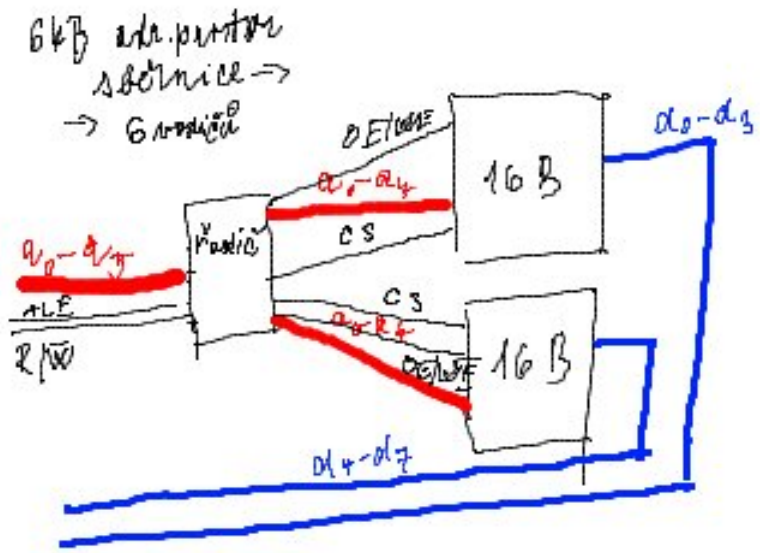
Instrukce INT *"offset"* je instrukce softwarového přerušení, jímž nepřímo voláme adresu na daném offsetu tabulky softwarové přerušení (interrupt vector table = IVT).
Instrukci CALL nemůžeme místo toho použít, stránky, na kterých IVT leží, jsou přístupné jen v privilegovaném režimu. Instrukce INT toto obchází tím, že volá pouze adresy na daných offsetech tabulky, a privilegovaný režim nastavuje implicitně.
Instrukce INT s parametrem 80h je systémové volání (syscall). Přepíná do supervisora, jdou v něm zavolat systémové funkce, které se starají o:

1. Vstup/výstup
2. Abstrakci nad diskem - ReadFile, WriteFile,...
3. Správu paměti - AllocMem, FreeMem
4. Program loader - Exec, Exit

4

TBD

5



Sběrnice naadresuje 64 bytů, tzn na úplné naadresování potřebujeme 6 adresových vodičů. Naše paměť budou 2 16-bytové moduly SRAM s velikostí slova 4 bity. Protože je velikost slova 4 bity, pak kdykoliv paměť pošle nějakou informaci, jsou to 4 bity (paměť o velikosti 16B se ukládá po 4 bitech) → potřebujeme 5 adresových vodičů na naadresování všeho, 6-tý budeme ignorovat.
O naadresování se bude starat řadič paměti. Do něj půjde všech 6 adresových vodičů sběrnice, z něj 5 do obou paměťových modulů.
Bude určeno, že v každém modulu budou uloženy 4-bity z každého bytu, tedy datové vodiče se budou skládat. Adresové vodiče povedou stejné do obou zařízení.
R/W signál určuje, jestli chce master do paměti zapisovat, nebo z ní číst.
R/W signál si přeloží řadič na OE/WE. Při čtení i zápisu vždy "chip-selectne"oba, a odpovídající paměť z tama přečte, či zapíše.
Pozn.: ALE signál v obrázku není potřeba, ten se používá jenom když jsou sdílené datové a adresové na rozlišení, co po nich právě jde.

6

Pojmenujme si globální proměnné g1 a g2, kde g1 bude ležet na adrese 00169138h a g2 na 0016913Ch. Funkce na všechno používá registr eax, nepoužívá žádné lokální proměnné.

```
longword DoSomeMagic() {
if (g1 + 5 < 7) {
if ((g2 << 1) >= g1) {
return g1 + 1;
}
}
return g2 + 1;
}
```

Pozn: Místo shiftu left 1 se dá pravděpodobně dát i "* 2". Překladač to často nahrazuje za shift, jedná-li se o mocniny dvojky.

7

1. Vstup z klávesnice - char ReadKey();... Tato funkce bude čekat na vstup z klávesnice. Kdykoliv uživatel zmáčkne nějakou klávesu, počítač na to zareaguje (například může to vypsat na obrazovku). Funkce vrátí znak, který uživatel napsal.
2. Výstup na obrazovku - void WriteToScreen(string message);... Tato funkce se bude starat o výpis na obrazovku. Parametrem je zpráva, která má být vypsána.
3. Číst sektory na disku - TBD

8

Můžu provést něco jako "pokusit se číst z adresy vaddr, a když to nepůjde, vyhodí to chybu, kterou chytím, a to bude vracet false, jinak true", ale asi se to takto nemá dělat.

```
typedef PLongword *longword;
bool CanRead(PLongword ptBase, longword vaddr) {
while (true) {

ptBase++;
}
}
```

TBD

10. 2. 2016

1

Nebudu číst :D...

2

TBD

3

1. Dělení nulou - není moc co popisovat, při dělení nulou v celočíselné aritmetice místo provedení příkazu procesor vyhodí softwarové přerušení (u x86 myslím INT 0x0)
2. Page fault - pokud se pokoušíme přistoupit do stránky, která je v guard modu
3. General protection fault - snažíme se přistoupit do stránky, kam nemáme právo, třeba něco zapsat do kernelu z user modu

Po vzniku kterékoliv chyby se místo příkazu, který es procesor chystal provést (třeba dělení nulou), zavolá obsluha přerušení na x86 příkazem INT *"offset přerušení v IVT"*. Dojde k automatickému nastavení supervisorského režimu při tomto volání, protože tato část paměti je v user modu nepřístupná. Příkazem IRET dojde k návratu.

4

Této situaci se říká "stack overflow". V moderních systémech je stránka, která následuje po zásobníku, rezervována příznakem "guard page". Kdykoliv zásobník přesáhne stránku, na které právě je, naalokuje tu následující v guard page modu, a bude chtít tu další zaalokovat do "guard modu". To se mu ale nepovede, protože tato stránka není ve stavu "free memory". Bude tedy vědět, že stránka, na které momentálně je, je jeho poslední, a dál už nemůže, a po případném přetečení této stránky vyvolá odpovídající chybu.

5

TBD

6

Vezměme z těchto dvou jazyků C#.
Jazyk se při kompilaci nepřeloží do strojového kódu žádného (známého) procesoru, místo toho se přeloží do "intermediate language"(tady CIL = common intermediate language).
Na spuštění zkompilovaného programu je potřeba "virtual machine"(tady .NET), která tyto příkazy bude interpretovat do strojového kódu procesoru, na kterým je program spuštěn. Bude to provádět tak, že vždy bude interpretovat ne po jednom příkazu, ale po daných úsecích takových, jakých potřebuje (=JITování). Tedy narazí-li na funkci, kterou se chystá zavolat, vyJITuje ji (převede do strojového kódu), a spustí.
Velkou výhodou "intermediate language"je přenositelnost. Protože se kód spouští na VM (=virtual machine), program je osvobozen od závislosti na procesorové instrukční sadě, a je tedy binárně přenositelný. Další výhodou je, že protože se samotný kód vykonává ve VM, je tam efekt "sandboxingu", tedy VM nás ochrání, pokud by se měla vykonat například nějaká nevhodná instrukce.

7

Určeme, že: Funkce začínající na 0x1036 bude Func1, 0x102A bude Func2 a 0x1042 bude Func3.

```
int Func2(word p1, word p2) {
return p1 + p2;
}
void Procedure1() {
Func3(Func2(Func1(), Func1()), 1);
}
```

Funkce Func1 nepřijímá parametry a má návratový typ "word". U funkce Func3 nelze rozhodnout, zda-li něco vrací, nebo ne, ale protože se to, co vrací nepoužívá, předpokládám void. Poznámka: Kdykoliv je celé číslo v jazyce, je automaticky 4 bytové.

8

Při spuštění debugování, ještě před spuštěním instrukcí, si debugger poznamená, na kterých instrukcích začínají breakpointy.

9

Za předpokladu, že kód není na případném OS read-only, můžeme vzít pointer na proceduru Print, a v inline assembleru potom například všechny operace související s voláním WriteLn nahradit operací nop, nebo lépe, můžeme místo napushování stringu 'Hello' do parametru hned na první adresu zapsat instrukci "jump"na instrukci "ret"(=end).

10

Všechno uděláme za pomocí shiftu a vhodných masek.
Nejprve extrahujeme bity ze vstupu:

```
qword Sign(dword flt32) {
if (flt32 & 2^31)
return 1;
return 0;
}
qword Mantissa(dword flt32) {
qword mask = 1;
for (int i = 0; i < 22; i++) {
mask = (mask << 1) + 1; // or ||
}
return (flt32 & mask);
}
qword Exponent(dword flt32) {
flt32 = flt32 << 1; // get rid of sign bit
flt32 = flt32 >> 24;
return (qword)flt32;
}
```

Máme funkce na extrahování jednotlivých složek, musíme je teď upravit na vhodný formát.

- Znaménkový bit zůstane stejný
- Exponent zůstane stejný, jen potřebujeme rozkopírovat nejvyšší bit toho původního (=znaménkové rozšíření)
- Mantissu musíme přepsat. Máme-li mantissu m původního 32-bitového floatu, pak mantissa 64-bitového floatu bude $m + (1023 - 127)$

Teď už stačí jen vhodnými ory a shifty všechno nastavit.

13. 7. 2016

1

Tří-stavová logika je metoda posílání informací taková, že v ní existují tři stavy: **logická 1, logická 0, floating state** (= float stav). První dva jsou jasné, třetí označuje stav, kdy po drátu neposílá žádná informace. Triky v elektrickém obvodu lze odlišit.

2

Je windowsácký swapovací soubor → když dojde systému fyzická paměť, začne používat virtuální → část toho, co má uloženého na fyzická paměť zkopíruje na harddisk, do swapovacího souboru si uloží informaci, kde to leželo, a kde to leží na hdd teď, a tím pádem se uvolní fyzická paměť.
Problémem tohoto postupu je, že když chci pracovat s informacemi, které jsem dočasně hodil na harddisk, bude to trvat odporně dlouho.
Řešení: kupte si větší RAMku nebo vypněte pár oken google chromu.

3

Doporučený postup:
Rozepište si při přepisování do Pascalu (nebo jinam) nejdřív kód assembleru tak, abyste akumulátor brali jako normální proměnnou, a přiřazovali do ní atd... Potom zkracujte.
Funkce Bar() je funkce na násobení a číslem n.

```
byte Bar(byte n, byte a) {
byte x = a;
while (n != 2) {
x += a;
n--;
}
return x;
}
```

Nechť g je globální proměnná na adrese $065b.

```
byte Foo(byte a, byte b) {
byte x = g;
x += Bar(a, 3);
x += Bar(b, 5);
return x;
}
```

4

Ve volací konvenci musí být specifikováno, jak se na zásobník ukládají parametry volané funkce, lokální parametry funkce a případně návratová hodnota.
V příkladu je použita nejspíš Céčková volací konvence, kdy nejprve jsou před zavoláním napushovány parametry funkce zprava doleva, potom je zavolána funkce, což uloží návratovou adresu na zásobník a zavolá ji, a potom jsou napushovány lokální parametry na zásobník zleva doprava.
Návratová hodnota se udržuje v akumulátoru.

5

Ano, změnilo.
Znaménková aritmetika, co se týká +, funguje stejně jako aritmetika neznaménková. Ovšem oproti kódu s novou hlavičkou by muselo dojít ke konverzi na bytu $0650, protože bychom z typu shortint

převáděli na typ byte. Na tomto místě by byla zavolána nějaká funkce RTL knihovny, která by se o ni postarala.

6

Při programování simulátoru by se mělo postupovat asi takhle:

1. Načtu soubor v prvním parametru, uložím do pole bytů začínajícího na adrese dané v druhém parametru
2. Vytvořím si proměnné pro všechny registry
3. Vytvořím funkce simulující jednotlivé instrukce.
4. Hlavní tělo programu bude nějaký cyklus, který bude běžet, dokud budou existovat ještě nějaké instrukce k přečtení.

Tady jsem to převedl nějak do kódu v C#. Problém je, že v zadání je, že do paměti si nahrajem jen daný kód co potřebujem, takže nemám obraz paměti, nemám registry, nemám stack. Nějak jsem si to dotvořil, ale celé to nefunguje, takže je to opět takový rozšířenější pseudokód, ve kterém ale některé věci chybí.
Odkaz: `http://pastebin.com/FUBiuitp`

7

Return hodnota word má hodnotu 2B, tzn musí se to předělat na sčítání s přenosem. To je jednoduché, protože operace ADC nastavuje carry příznak, a stejně tak ho implicitně využívá.

8

TBD

9

TBD

10

Na adrese 1F00 bude ležet a, na adrese 1F02 bude b, na 1F05 c, na adrese 1F08 bude d, na 1F0D e, od 1F0E by leželo pole charů = string, což by bylo 16 bytů + 1 koncový.
Big endianita se projeví jen na b a d, kde na nejnižší adrese by byl byte s nejvyšší relevancí.
Překladač by mohl zmenšit padding tím, že by přehodil pořadí položek, tedy nejprve tam naházet byty, potom word, potom longint, potom třeba string.