Komparcho paaiškinimai. Kodėl taip yra?

Pagal Jūsų klausimus sudarytas atsakymų rinkinukas bandant paaiškinti kodėl vieni ar kiti architektūriniai sprendimai yra tokie, kokie yra, o ne kitokie.

Turint patikslinimų kreiptis: benediktog@gmail.com

Updated on: 2016-01-10 14:19

Turinys

Skaičiavimo sistemos, baitai, žodžiai

Kam tie skaičiai su ženklu ir be ženklo? Kaip žinoti mano skaičius su ženklu ar be ženklo?

Užrašinėjant skaičius baituose ir žodžiuose *unsigned* (be ženklo – tik neneigiamų sk. pradedant nuliu) tapo per mažai.

Žinant, kad baituose galima atlikinėti veiksmus su nedideliais skaičiais 0..255 (žodžiuose

0..65535) kilo poreikis galėti dirbti ir su nedidelėmis neigiamomis reikšmėmis – tarkim kaip tiesiog iš turimo skaičiaus atimti vienetą? Skaičiavimai, kur galima tik pridėti nėra tokie lankstūs, kaip būtų tokie, kur galima naudoti ir atimtį.

Pastebėta, kad tai visai paprasta pasiekti, atsisakant dalies teigiamų reikšmių.

Esminis aritmetinių veiksmų poreikis yra, kad aritmetikoje visuomet gautume teisingus rezultatus. Baitams ir žodžiams turint ribotą reikšmių kiekį pastebėta, kad baite: 255 + 1 = …. ne kas kito, kaip 0.

Tada pagalvota – galbūt galima sakyti, kad (-1) + 1 = 0, tą 255 pasižymint, kaip -1 ?

Pasirodo galima.

Tuomet apsiribota intervalų [0;255], [0;65535] perskėlimu per pusę, taip, kad 0 ir teigiami skaičiai tiek su ženklu, tiek be ženklo būtų užrašomi vienodai, o kita intervalo pusė (baituose 80-FF, žodžiuose 8000-FFFF) būtų interpretuojama arba kaip skaičius su ženklu arba kaip skaičius be ženklo pagal poreikį ir turėtų skirtingas reikšmes:

Baitai: 00, 01, 02, 03 … 7E, 7F, ***80***, 81 … FD, FE, FF

Be ženklo: 0, 1, 2, 3 …. 126, 127, ***128***, 129 …. 253, 254, 255

Su ženklu: 0, 1, 2, 3 …. 126, 127, ***-128***, -127 … -3, -2, -1

Panašiai ir žodžiuose, tik juose maksimali teigiama dešimtaine reikšmė su ženklu yra ne 127, o 32767 (tiesiog žodžiuose įtalpiname didesnes reikšmes, bet taisyklės taikomos tos pačios).

Jei teko programuoti C ar C++ ar panašiomis kalbomis, tai teko šiuos tipus sutikti žymimais *signed* ir *unsigned* keyword'ais.

Kompiuteriui tai yra tiesiog dvejetainės baitų sekos, kurias interpretuojame pagal poreikį kaip skaičius su ženklu ar be ženklo. Iš tiesų visi duomenys kompiuteriui yra tik dvejetainės bitų sekos, kurias interpretuoti kaip skaičius be ženklo, tekstą, paveiksliuko dalį, muzikos natą ar pan. priklauso nuo naudojimo konteksto.

Tarkim skaičiuodami atstumą nesirūpiname neigiamais skaičiais, o skaičiuojant temperatūrą tokie jie yra reikalingi.

Adresavimas

Ką duoda plėtimo pagal ženklą taisyklė?

Jei skaičiai suprantami kaip skaičiai su ženklu, tai reikšmes, kurios telpa ir viename baite [-128;127] galima sutalpinti į baitą ir nenaudojant tam pilno žodžio. Neneigiamoms reikšmėms žodis vis tiek visada prasidės 00, o neigiamoms FF. T.y.:

0 → 00 → 0000

1 → 01 → 0001

127 → 7F (0 vyriausias bitas) → 007F

-128 → 80 (1 vyriausias bitas) → 0080

-129 → 81 (1 vyriausias bitas) → 0081

-2 → FE → FFFE.

Dėl ko pirmiau eina jaunesnysis, o po to vyresnysis baitas?

Taip buvo paprasčiau realizuoti techniškai. Tai dar vadinama “Big Endian” formatu.

Lengviau techniškai turėti:

- Nuskaityk jauniausią baitą adresu EA (*a.k.a efektyvus adresas*)
- Nuskaityk dar ir vyresnį baitą adresu EA+1, jei tai žodis

  T.y. prireikia vos vieno patikrinimo papildomam veiksmui.

  Tai paprasčiau negu:
- Nuskaityk jauniausią baitą adresu EA, jei baitas
- Nuskaityk jauniausią baitą adresu EA+1, jei žodis
- Nuskaityk vyresnį baitą adresu EA, jei žodis

  Kas techninę realizaciją labai apsunkintų.

SF registras

ZF flago prasmė

Kadangi dažniausiai į SF registrą atsižvelgiama po CMP komandos, o ši atlieka dviejų reikšmių atimtį, tai ZF prasmė gaunama tokia:

Jei atimties rezultatas nulis (lyginti du vienodi skaičiai) – ZF = 1, kitu atveju ZF = 0. Vadinasi iš ZF sužinome ar lyginti skaičiai yra vienodi ar ne.

Taip pat jei po SUB komandos (atimties) ar gautas rezultatas yra nulis.

CF ir OF flagų prasmė

Dirbant su baitais/žodžiais kaip skaičiais su ženklu arba be ženklo prireikia skaičiuoti ar rezultatas neišlindo iš atitinkamo intervalo.

Sudėties atveju tai reikštų, kad buvo gautas didesnis nei telpa rezultatas.

Atimties atveju, kad atimtas didesnis skaičius iš mažesnio.

Kadangi kompiuteris tik atlieka veiksmus ir nežino ar skaičiai yra be ženklo ar su ženklu, jis paskaičiuoja abu dalykus – tiek perpildymą skaičiuje be ženklo (CF), tiek su ženklu (OF).

PF flago prasmė

Lyginumo bitas sutinkamas teorijoje apie kodavimą: http://www.mif.vu.lt/~skersys/15r/dm/konsp5.pdf (žr. kontrolinio simbolio kodą).

http://ccm.net/contents/134-introduction-to-encryption-with-des žr. “Principle of DES” pirmą pastraipą.

Perduodant informaciją dažnai naudojami klaidas taisantys kodai – pridedama papildomų simbolių, kurie padėtų nustatyti ar informacija yra sugadinta ir jei taip – pamėginti ją atstatyti, tame pasitarnauja ir bitų parity (lyginumo) žinojimas, kuris ~50% atvejų pataiko ar informacija iškraipyta ar ne. Jei nieko daugiau nenaudojame, tai yra nepatikimas rodiklis, bet naudojant kartu su kitais gali pasitarnauti išvengiant dar sudėtingesnių skaičiavimų, kai jie nebūtini.

Šifruojant duomenis greitaveika irgi svarbi (negalime šifruoti teksto metus, jei reikia jį perduoti per kelias sekundes), todėl jei yra kompiuteryje architektūrinėmis priemonėmis užtikrinamas efektyvus parity skaičiavimas tai apsaugo nuo sudėtingų/resursus ryjančių procedūrų rašymo, kiekvienas sutaupytas skaičiavimas yra sutaupytas ir laikas bei pajėgumas gali būti panaudoti kitais tikslais.

Pertraukimai

Kokia tų pertraukimo procedūrų nauda?

Leidžia aprašyti vienodą reakciją į programose nuolat pasikartojančius poreikius („simbolio spausdinimas ekrane“, „programos darbo pabaiga“ ir kt.)

Leidžia sureaguoti į apratūros įvykius: dingo elektra (power-fail), pajudinta pelė.. ir kt. Susiradus Ralf-Brown interrupt list galima atrasti daug įvairių paskirčių.

Bene pagrindinis operacinių sistemų skirtumas ir yra tas, kad jos net naudodamas tą patį procesorių, tą pačią atmintį yra nesuderinamos, nes naudoja skirtingas pertraukimų procedūras.

DOS INT 21h pertraukimas iš dalies suderinamas su INT 0x80 (80h) Linux sistemose, bet juos susieti būtų ne taip jau paprasta, nekalbant jau apie kitus specifinius sprendimus. Linux sistemose visus išvedimus (nesvarbu ekraną, failą ar kur kitur) programiškai suprasti kaip baitų sekų rašymą į failą, tik priklausomai nuo įrenginių skiriasi kokios baitų sekos kaip yra suprantamos.

Kaip gali užtekti 256 pertraukimų procedūrų?

Užtenka, nes galima padavinėti papildomus argumentus registruose, tarkim int 21h pertraukimas savo funkciją, kurią reikės vykdyti sužino iš ah registro, o funkcijų parametrus iš kitų prie pačios konkrečios funkcijos nurodytų registrų. Pavyzdžiui int 21h, ah=02h spausdintino simbolio reikšmę pasiima iš registro DL.

Slankaus kablelio skaičiai

Kas ta normalizuota forma ir kam ji reikalinga?

Dešimtainėje sistemoje populiari yra standartinė išraiška, t.y.:

$123 = 1.23 * 10^{2}$

$0.000123 = 1.23 * 10^{-4}$

Kur skaičius dauginamas iš 10 laipsnio visada yra ne mažesnis už 1 ir griežtai mažesnis už 10.

Dvejetainėje sistemoje tiek skaitmenų nėra, čia standartinė išraiška (normalizuota forma) visada prasideda vienetu:

$5.7510 = 101.11 = 1.0111 * 2^2$

$0.7510 = 0.11 = 1.1 * 2^{-1}$

Normalizuotos formos naudojimas leidžia sutaupyti vietos (papildomą bitą) slankaus kablelio formatuose, kur ir taip žinoma, kad prieš kablelį visada bus vienetas.

Kodėl realiesiems skaičiams pasirinktas būtent toks slankių kablelių formatų sprendimas?

Poreikiai buvo tokie:

- Reikia taupyti atmintį (4,8,10 baitų nėra daug)
- Reikia aprašyti tiek labai dideles reikšmes (kosminius skaičius), tiek mažytes (bakterijos dydį ir pan.)
- Nebūtinas ypatingas tikslumas
- Turi greitai veikti aritmetinės operacijos

To leidžia pasiekti trumpas, ilgas, vidinis realūs formatai.

Leidžia aprašyti tiek didelius, tiek mažus skaičius, susitaikant su tuo, kad tam tikra nežymi reikšmės atžvilgiu paklaida vis tiek bus, žinant, kad rašant programas ir naudojant *float* arba *double* tipus, teks pasirinkti paklaidą (apie 1tūkstantąją pačios reikšmės ar net mažesnę) kuriai esant bus laikoma, kad skaičiai yra lygūs.

Aritmetinės operacijos atrodo šitaip:

Sudėtis ir atimtis:

- Pashiftiname mantises taip, kad suvienodintume charakteristikas
- Atliekame veiksmus
- Normalizuojame reikšmes, nustatomą naują charakteristiką ir mantisę.

Daugyba:

- Sudedame charakteristikų eiles
- Sudauginame mantises
- Normalizuojame reikšmes, nustatome naują charakteristiką ir mantisę

Su dalyba kiek kebliau, bet esmė panaši kaip daugyboje, tik ten jau charakteristikų eilės

atimamos. Bet esmė tokia.

Kodėl dešimtyje baitų neprivaloma normalizuota forma?

- Užtenka vietos, 80 bitų ir taip gana daug
- Formatas naudojamas naudojantis koprocesoriumi, leidžiant patiems iškart pateikti koprocesoriui reikšmes, kuriose jau yra vienodos charakteristikos ar atlikti kitokie optimizavimai, kurie kartais natūraliai išplaukia iš situacijos, kai tarkim ir taip turimi skaičiai su vienodomis charakteristikomis.

Supakuoti – išpakuoti skaičiai

Kam jie išvis reikalingi, jei jau turime realiuosius skaičius (float)?

Realieji skaičiai turi labai realų trūkumą – jie leidžia sau prarasti tikslumą, nes realiųjų skaičiųjų aibė yra tolydi (nėra trūkių), o natūraliųjų – diskreti. T.y. tarp dviejų realiųjų skaičių visada bus dar kitas realus skaičius, bet tarp natūralių 2 ir 3 nebėra jokio tarpinio kito natūralaus.

Saugant banko sąskaitų likučius netikslumai negalimi, taip pat naudojama dešimtainė sistema, kur žinoma, kiek pozicijų bus po kablelio ir kiek sveikajam skaičiui – todėl supakuotų ir išpakuotų skaičių komandos leidžia turėti decimal tipo operacijas palengvintas architektūriškai. Taip jos ir veikia greičiau. (google for: Decimal type dėl daugiau info)

Valdymo perdavimas

Ką palengvina tai, kad IP rodo į sekančią komandą?

Nors valdymo perdavimo metu ir dažniau einama vykdyti ne tos komandos, kuri eina paeiliui po dabartinės, bet... gi dauguma komandų visai nėra valdymo perdavimo. Didžiąją laiko dalį kodas vyksta nuosekliai, todėl jei procesorius komandos vykdymo metu jau gali žinot apie sekančios komandos buvimo vietą, jau gali būti imtąsi veiksmų analizuoti sekančią komandą, nuskaityti ir įvertinti jos tipą, pavadinimą, naudojamus parametrus ir taip pagreitinti procesoriaus darbą.

Kodėl reikėjo netiesioginio valdymo perdavimo?

Rašant sudėtingas programas pastebėta, kad kartais yra labai gerai atmintyje pasidėti procedūros adresą:

mov *bx*, *offset magiška_procedūra*

ir panaudoti:

jmp word ptr[bx] ← valdymas perduodamas adresu cs:offset magiška_procedūra (pereinama į ją)

Kodėl išorinis valdymo perdavimas ima naują IP ir neskaičiuoja jokių poslinkių?

Nes jei jau keičiamas netgi CS registras, tai akivaizdu, kad sėdima nebe tame pačiame segmente ir tiksliai žinoma, kur bus norima nušokti (su kokiu poslinkiu jame). Mažoms programos (iki 10000_{10} eilučių) dažniausiai nėra poreikio turėti išorinį valdymo perdavimą.

Kam RET komandai tas steko išlyginimas?

C kalbos atžvilgiu kai viena funkcija kviečia kitą: *sudeti(1, atimti(5, 3));* tai prieš kviečiant call komandą yra sudedami parametrai į steką taip, kad vykdant tokią C funkciją prieš kviečiant CALL komandą sudėjus tos funkcijos argumentus vėliausiai būtų supushintas grįžimo adresas. Grįžtant iš procedūros pasiimamas adresas ir „pravalomas stekas“ (iš tikrųjų jokios reikšmės nevalomos) taip, kad SP pastatomas ten, kur stovėjo prieš kviečiant funkciją (paimta iš kito mano konspekto “KA_KursoApzvalga”):

Tokia C kalba parašyta programėlė...:

```
 1  #include <stdio.h>
 2
 3  int main(){
 4      char masyvas[20];
 5      masyvas[0]=3;
 6      masyvas[1]=011;
 7      masyvas[2]=0x11;
 8      masyvas[3]='1';
 9      putchar(masyvas[3]);
10      return 75;
11  }
12
```

Viename iš kompiliavimo etapų – vertime į asemblerinį kodą virsta labai panašia programėle į šitą:

```
1   .model small
2   .stack 100h
3   .data
4       masyvas db 20 dup (?)
5   .code
6   main PROC
7       mov ax, @data
8       mov ds, ax
9
10      mov masyvas[0], 3 ;irasyk desimtaine reiksme 3
11      mov masyvas[1], 11o ;irasyk astuntaine reiksme 11
12      mov masyvas[2], 11h ;irasyk sesioliktaine reiksme 11
13      mov masyvas[3], '1' ;irasyk '1' ASCII koda
14      push word ptr masyvas[3] ;padedam parametra procedurai
15      call putchar ;kvieciame procedura
16
17      mov ah, 4Ch
18      mov al, 75 ;pasakome programos baigimo grazinama baita (exit code)
19      int 21h ;uzbaigiame darba, nes pertraukimas INT 21h gauna ah=4Ch
20  main ENDP
21
22  putchar PROC ;proceduros apibrezimas
23      mov bp, sp ;bp - base pointer (adresavimui steke, susigrąžinam)
24          push ax
25          push dx
26              mov ah, 2
27              mov dl, [bp+2] ;imame parametro jaunesniji baita
28              int 21h ;ah=2, tai bus simbolio spausdinimas
29          pop dx
30          pop ax
31      mov sp, bp ;atstatom SP reiksme
32      RET 2 ;steko islyginimas parametru praleidimui (2*parametru kiekis)
33  putchar ENDP
34
35  END
```

Žr. 32 eilutę.

Kaip gali užtekti sąlyginių JMPų su vieno baito poslinkiu?

Neužtenka, todėl programuojant įterpiami tarpiniai JMP be sąlygų, kurie jau gali nušokti bet kur. Programuojant EMU tai nesijaučia, nes EMU tai padaro už mus pačius, o Tasm'e gauname kompiliavimo klaidas ir turime pasitaisyti.

Eilutinės komandos

Ką gero duoda šios komandos?

Leidžia patogiai dirbti su simbolių eilutėmis. Poveikiui pastebėti užtenka pabandyti keliskart jomis pasinaudoti „ant popieriaus“:

MOVS – leidžia vieną simbolių eilutę iš DS:SI nukopijuoti į ES:DI. Iš kurio galo skaitoma nusako DF flago reikšmė, kiek baitų/žodžių → CX registro reikšmė.

LODS – leidžia po vieną pasiimti į AL (arba AX) vis sekantį eilutės iš DS:SI simbolį/žodį. Neprasminga turėti REP LODSB, bet prasminga šią komandą kviestis cikle vis sekančiam simboliui/žodžiui pasiimti. O simbolis ar žodis čia dar ir priklauso nuo situacijos ir koduotės.

STOS – leidžia vieną ir tą pačią baito/žodžio reikšmę įrašyti į CX vietų paeiliui. Į kurią pusę judama vėl parodo DF flagas.

CMPS – leidžia judėti per simbolių eilutes tol, kol jos sutampa arba nesutampa.

SCAS – leidžia judėti per simbolių eilutes ieškant konkretaus simbolio sutapimų/nesutapimų.

Visais atvejais už kartų kiekį atsako CX (jei yra pakartojimo prefiksas), už kryptį DF flagas, o už dydį žodžiais ar baitais prie komandos prikabinta B (byte) arba W (word) raidė.

MPL

Dėl ko tai sugalvota? Ar neužtenka asemblerio?

Asembleris nors iš pažiūros paprasta kalba, bet reikalautų didelių mikroschemų sąnaudų, jei norėtume įgyvendinti visas situacijas, todėl apsiribojama kitokiu principu – sugalvota paprasta schema, kurioje vyksta tik reikšmių persiuntimai ir sumavimo veiksmas, tada išmokta, kad atimtį baituose/žodžiuose galima susimuliuoti per sudėtį pakeitus atimamo skaičiaus ženklą, daugybą atliekant daug sudėčių su shift'ais (pastūmimais) ir dalybą – su daug atimčių. O visa kita jau gauname iš šių veiksmų.

Taigi paprasčiau buvo sukurti tokias mikroschemas, ir procesoriuje parašyti savotišką „disasemblerį“ (komandų analizatorių perprantantį kokia čia komanda) ir išmokyti jį atitinkamai reaguoti ir vykdyti komandas taip, kad reikalinguose registruose atsirastų reikalingos reikšmės, ir nueitų į atmintį ten kur reikia, tokia reikšmė, kokios reikia.

Taip išvengiama to, kad programuoti MPLu būtų labai skausminga patirtis, o procesorių sukurti iškart suprantantį asemblerį, be jokio MPL lygmens reikštų, kad turėsime sudėtingą procesorių, o kuo sudėtingesnius dalykus kuriame, tuo labiau tikėtina, kad jie dar ir turės klaidų, bei prireikus ką nors pakeisti bus sugaištama neprotingai daug laiko.