FKEF.02.094

Arvuti arhitektuur

Computer Architecture and Organization

dotsent
Toomas Plank

FKEF.02.094

Masinkood (2)

4. loeng,
14. märts 2008

Jutujuht

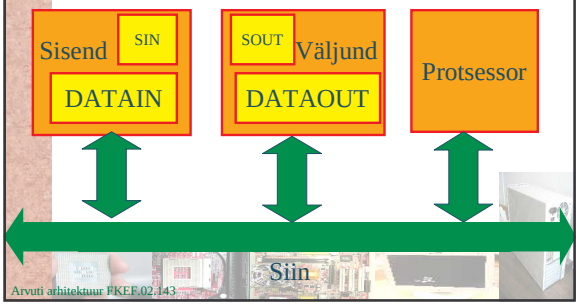
Selles loengus tutvume:

- Sisend/väljund operatsioonid
- Pinu
- Järjekord
- Andmetabel
- Nihe, pööre
- Masinkoodi käskude struktuur

Sisend/väljund operatsioonid

- Sisend/väljund operatsioonide tegemisel on mureks, et sisend/väljund seadmed ei suuda töötada sama kiiresti kui protsessor
- Seega on vaja andmete saatmine nende seadmete vahel sünkroniseerida
  - Protsessor saadab väljundseadmele info ja ootab seadmelt vastust
  - Sisendseade tõstab lipu üles ja ütleb sellega protsessorile: mul on sinu jaoks andmeid

Sisend/väljund operatsioonid



Näide

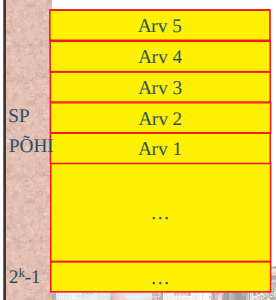
```
        Move #LOC,R0
READ    TestBit #2,INSTATUS
        Branch=0 READ
        MoveByte DATAIN,(R0)
ECHO    TestBit #2,OUTSTATUS
        Branch=0 ECHO
        MoveByte (R0),DATAOUT
        Compare #CR,(R0)+
        Branch≠0 READ
```

- Sellise programmi korral toimetab protsessor suurema osa ajast ootamistsüklites
- Tohutu ressursi raiskamine L
- Mõistlik kasutada katkestusi J

Pinu (*stack*)

- Andmete kogum (tavaliselt baidid või sõnad), mille puhul saab elemente lugeda/kirjutada ainult ühest andmekogumi otsast (ülemisest)
- Analoog: söökla kandikute virn
- LIFO (*last-in-first-out*)
- Spetsiaalne register (*stack pointer*) peab meeles, millisel aadressil asub pinu kõige värskem element
- Pinu kasvab tavaliselt väiksemate aadresside suunas

Pinu graafiliselt



- Pinusse panemine
  Subtract #4,SP
  Move UUSASI,(SP)
- Pinust võtmine
  Move (SP),LUGEM
  Add #4,SP
- Lühemalt
  - Move UUSASI,-(SP)
  - Move (SP)+,LUGEM

Pinu (*stack*)

- Reaalses elus on pinul ka omad piirid – teatud fikseeritud mälupiirkond, s t.
  - ei saa panna pinusse arvu, kui ülemine piir on kätte jõudnud
  - ei saa võtta pinust arvu, kui põhi on juba käes

Näide: olgu pinu põhi kohal 3000
ja lagi kohal 1000

- Compare #3000,SP
  Branch>0 PINU_PÕHI_KÄES
  Move (SP)+,LUGEM
- Compare #1000,SP
  Branch≤0 PINU_TÄIS
  Move UUSASI,-(SP)

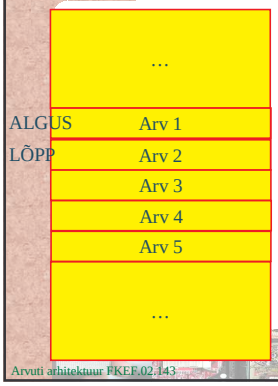
Järjekord *(Queue)*

- Üsna sarnane struktuur võrreldes pinuga

Eripärad:

- FIFO (*first-in-first-out*)
- järjekord kasvab tavaliselt suuremate aadresside suunas
- Elemente lisatakse alumisest otsast ja loetakse ülemisest otsast
- Järje pidamiseks vajame kahte pointerit

Järjekord graafiliselt



- Näeme, et järjekord liigub pidevalt kõrgemate aadresside suunas
- Kasutatakse ka ringpuhvrit
- Silmas tuleb ka pidada seda, et reserveeritud mälupiirkonnast välja ei satutaks

Alamprogrammid

- Sageli on tarvis mingit kindlat ülesannet teha palju kordi
- Üks võimalus on kirjutada alamprogramm
- See kutsutakse välja (*call*) iga kord, kui seda ülesannet teha vaja on
- Peale alamprogrammi täitmist on vaja endisesse kohta tagasi jõuda (*return*)
  - kuna alamprogrammi kutsutakse välja erinevatest kohtadest, siis on ka potentsiaalseid naasmiskohti palju
  - teine keerukus on selles, et peale naasmist peab programmiloenduri sisu vastama uuele/vanale/ olukorrale (ehk siis call käsk peab selle info kuskil talletama)

Subrotine linkage method

- Link-registris salvestatakse tagasipöördumise aadress (sisuliselt PC sisu)
- Tagasipöördumisel viidatakse kaudselt sellele registrile

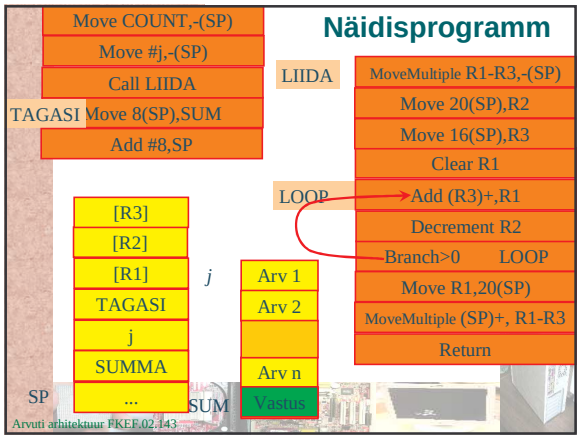
Probleem

- sageli juhtub nii, et alamprogrammi sees on tarvis kasutada teist alamprogrammi
- ka selle alamprogrammi jaoks on vaja link-registris salvestada tagasipöördumise aadress
- Lahendus: kasutatakse pinu

Parameetrite edastamine

- Alamprogrammi väljakutsumisel tuleb saata sinna ka teatav hulk parameetreid
- Alamprogrammist naastes on vaja põhiprogrammi tuua rehkenduse tulemus(ed)
  - Üks võimalus on kasutada registreid või mälu
  - Teine võimalus on panna parameetrid/andmed pinusse
- Väikese arvu parameetrite korral on esimene variant lihtsam
- Pinu kasutamine on paindlikum lahendus – saab edastada palju parameetreid/andmeid

Näidisprogramm

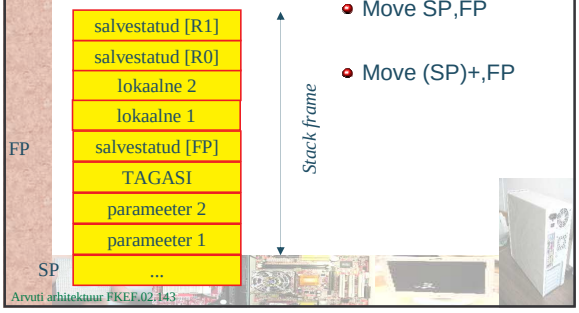


Stack pointer ja *Frame pointer*

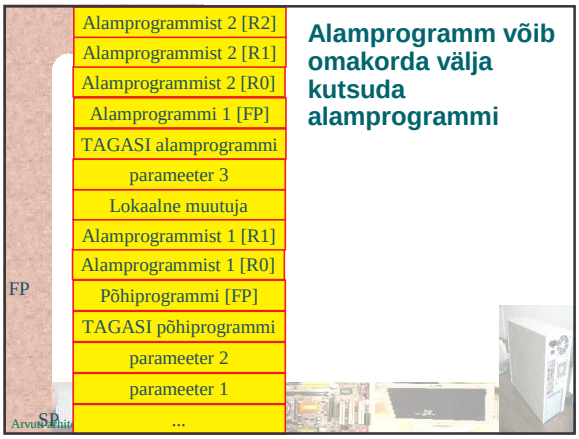
- Lisaks *Stack pointer*'ile on kasulik sisse tuua veel teine pointer – *frame pointer*
- *Frame pointer* võimaldab mugavalt viidata muutujatele pinus
- *Stack pointeriga* seda teha ei saa, kuna SP ei seisa ühe koha peal (muutuja lisamisel/kustutamisel SP väärtus muutub)
- Tavaks on, et FP viitab vahetult tagasipöördumise aadressi peal olevale väljale
- Sellel väljal salvestatakse FP senine väärtus

Piltlikult:

- Move FP,-(SP)
- Move SP,FP
- Move (SP)+,FP



Alamprogramm võib omakorda välja kutsuda alamprogrammi



Numbrite kirjapanek

- Kahendkoodi arvud
  - **#%00011011**
  - Add #%00011011,R0
- Kümnendkoodi arvud
  - **#27**
  - Add #27,R0
- Kuueteistkümnendkoodi arvud
  - **#$1B**
  - Add #$1B,R0

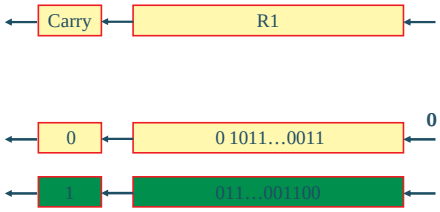
Loogikakäsud masinkoodis

- Loogikaahelate ülesehitamisel on ehituskivideks NING (AND), VÕI (OR) ja EI (NOT) ahelad
- Mõnikord on kasulik suuta teha neid operatsioone ka tarkvaraliselt
- Näide 1: Positiivsest täisarvust negatiivse saamine. Olgu arvuks **+5 (0101)** registris R1
  - Not R1 annab meile **-6 (1010)**
  - Add #1,R1 annab meile **-5 (1011)**
- Näide 2: Olgu meil tarvis kindlaks teha, kas registris R1 oleva 32-bitise sõna vasakpoolseim täht on X. ASCII koodis on täht X: 01011000 ehk $58
  - And #FF000000,R1
  - Compare #$58000000,R1
  - Branch=0 …

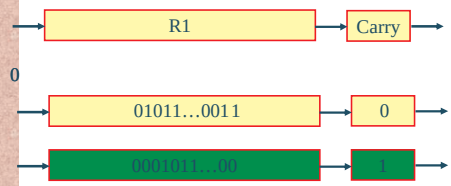
Nihe masinkoodis

- Paljudes rakendustes on tarvis nihutada operandi bitte paremale või vasakule etteantud arvu kohtade võrra
- Loogiline nihe
  - LShiftL arv,asukoht
  - LShiftR arv,asukoht
- Aritmeetiline nihe
  - AShiftR arv,asukoht

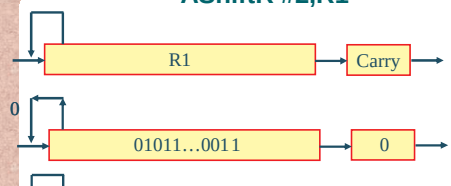
Nihe vasakule
LshiftL #2,R1



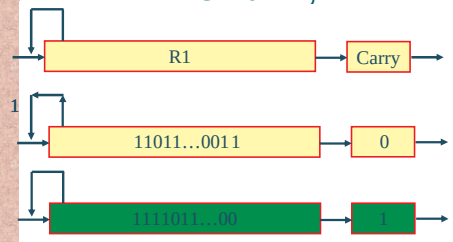
Nihe paremale
LShiftR #2,R1



Aritmeetiline nihe paremale
AShiftR #2,R1



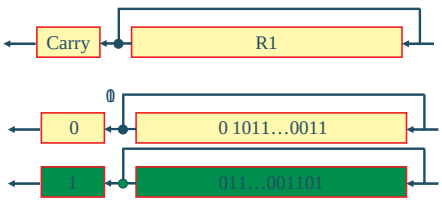
Aritmeetiline nihe paremale
AShiftR #2,R1



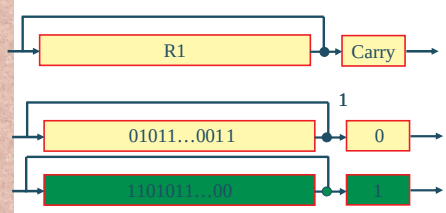
Pööre masinkoodis

- Nihke puhul lähevad osad bitid kaotsi
- Pöörde korral saame kogu info siiski säilitada
- Pööre ilma ülekannet (*Carry*) kasutamata
  - RotateL arv,asukoht
  - RotateR arv,asukoht
- Pööre ülekannet (*Carry*) kasutades
  - RotateLC arv,asukoht
  - RotateRC arv,asukoht

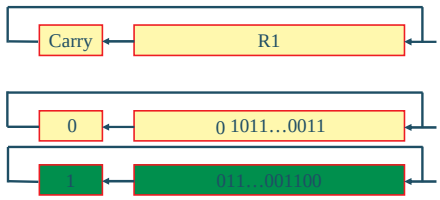
Pööre vasakule
RotateL #2,R1



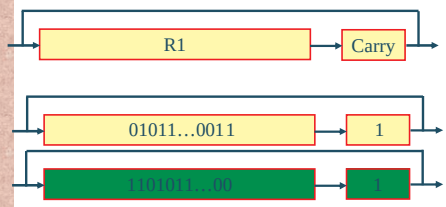
Pööre paremale
RotateR #2,R1



Ülekandega pööre vasakule
RotateLC #2,R1



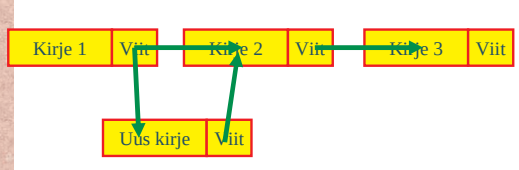
Ülekandega pööre paremale
RotateRC #2,R1



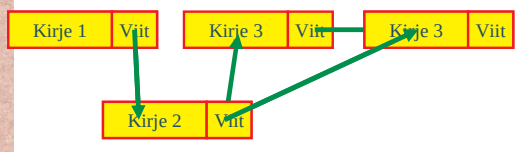
Korrutamine ja jagamine

- Enamuses käsustikke võimaldavad teha korrutamistehet: Rj←[Ri]×[Rj]
- Multiply Ri,Rj
  - kui sisendsuurused on n-bitised, siis vastus võib olla kuni 2n-bitine
  - seega võib tulemuse salvestamiseks vaja minna kahte registrit
- Mõned käsustikud võimaldavad teha ka jagamistehet: Rj←[Ri]/[Rj]
- Divide Ri,Rj
  - siin on vaja salvestada nii täisosa kui ka jääk
  - seega võib tulemuse salvestamiseks vaja minna kahte registrit
  - mõnikord jäetakse jääk ka arvestamata

Seotud jada
Elemendi lisamine



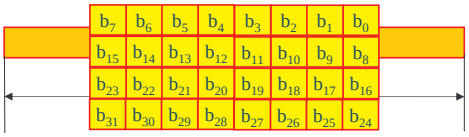
Seotud jada
Elemendi kustutamine



Masinkood

- Protsessoris käivitamiseks tuleb kõik käsud kodeerida
  - see ei ole enam *Assembly language*
  - vaid binaarkood
  - seda binaarkoodi kutsutakse masinkoodiks

Add R0,R1



- Operatsioonikood
  - Näiteks 8 bitti
  - 256 võimalust
- Registri aadressid
  - 16 registri puhul tuleks 4 bitti
- Registri adresseerimise viis
  - Näiteks 3 bitti
- Nüüd jääb veel 10 bitti üle

Move R0,27(R1)

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}
b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}

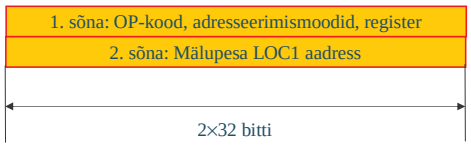
- Operatsioonikood
  - Näiteks 8 bitti
  - 256 võimalust
- Registri aadressid
  - 16 registri puhul tuleks 4 bitti
- Registri adresseerimise viis
  - Näiteks 3 bitti
- Indeksi väärtuse 27 salvestamiseks on nüüd 10 bitti
  - 27 à 1 1011 à vähemalt viis bitti + märgibitt
  - **Move R0,511(R1)** à 1023à 0111 1111 1111 à üheksa bitti + märgibitt
  - **Move R0,1023(R1)** à 1023à 0 1111 1111 1111 à kümme bitti + märgibitt

Move R0,LOC1

- Operatsioonikood
  - Näiteks 8 bitti
  - 256 võimalust
- Registri aadressid
  - 16 registri puhul tuleks 4 bitti
- Registri adresseerimise viis
  - Näiteks 3 bitti
- Mälu adresseerimiseks on nüüd 14 bitti
  - Seda on ilmselgelt liiga vähe L

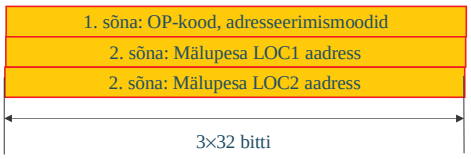
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}
b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}

Move R0,LOC1



- Võtame kasutusele kahesõnalise käsu

Move LOC1,LOC2



- Võtame kasutusele kolmesõnalise käsu

CISC

- Eelmistes näidetes jõudsime käsustikuni, mis lubab kasutada erineva pikkusega käske
- Sellised käsud on reeglina juba küllaltki keerulised
- *Complex Instruction Set Computer*

RISC

- Alternatiiv on nõuda ainult ühesõnaliste käskude kasutamist
- Käsk Add R0,LOC1 pole siin lubatud
- Käsk Add R0,(R1) on lubatud
  - R1 peab siis viitama mälupesale LOC1
- *Reduced Instruction Set Computer*
  - Kasutatakse lihtsaid ja lühikesi käske
  - Meie näide võtaks kuju
  - Move (R1),R3
  - Add R0,R3
- Kuidas 32-bitine aadress registrisse lugeda?
  - ühe sõnaga ei saagi
  - üks toimiv võimalus on kasutada nihutust ja loogilisi tehteid, ladudes selle aadressi väiksematest tükkidest kokku

Kasutatud kirjandus

- Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Computer organization 5th edition (2002) 805 p.