E-kursuse "Mikroprotsessorid"
materjalid

Aine maht 6 EAP

Artur Abels (Tartu Ülikool), 2012

Sissejuhatus

Enne mikrokontrolleri ehitusest ja tööpõhimõttest rääkimist tuleb vaadata, mis see on ja milleks seda vaja on.

Mikroprotsessoritele on erinevad kasutusvaldkonnad. Näiteks arvutit on tavaliselt vaja selleks, et teha suure arvutusmahuga tehteid, kus on vaja sisendit inimeselt. Arvutis on väga paindlik operatsioonisüsteem, kus saab käivitada paralleelselt mitu programmi, luua faile jne. Aga tihti meil pole vaja nii võimast ja paindliku tööriista nagu arvuti.

Teine äärmus on see, kui meile piisaks mingist triviaalsest asjast, näiteks lülitist või lihtsast loogikaskeemist. Näiteks ühel aastal „Robotex“ võistlusel oli tehtud robot mille juhtimisskeemis polnud ühtegi elektroonikakomponenti – ainult lülitid ja releed.

Tihti on meil vaja midagi lülititest targemat, samas arvutist palju lihtsamat Sageli on meil vaja käivitada ainult üks programm, mis jookseks aastaid ja teeks oma tööd - näiteks automaatikasüsteemides. Samuti väga tihti pole meil vaja suurt arvutusvõimsust ega suurt mälu mahtu.

Oletame, et me tahame teha kurkide kastmise süsteemi, mis mõõdaks kasvuhoones temperatuuri ja niiskust ning kontrolliks kastmiseks vee klappe. Selleks pole vaja teha võimsaid arvutusi ega saada kasutaja käest mingit sisendit., Lihtsalt on vaja mitme kuu jooksul kord minutis mõõta midagi ja teha otsus, kas on nüüd õige aeg kurke kasta või mitte.

Teine, keerulisem näide oleks see, et meil on vaja teha UAV autopiloot. Selle juures oleks meil vaja mitukümmend korda sekundis mõõta andurite näite, arvutada välja UAV orientatsioon ruumis ning kontrollida mootoreid, et hoida lennuk tasakaalus. Samas on vaja liikuda mingi etteantud lennuplaani järgi. Sellisel juhul kaalupiirangu tõttu me enamasti ei saaks panna lennuki peale 10kg arvutit. Samuti me ei tahaks autopiloodile suurel hulgal arvutis olevaid lisavõmalusi nagu bluetooth printeri jagamine jne. Meil oleks vaja midagi väiksemat kui arvuti, tihti mitte nii paindlikku ja mitte nii palju inimesele orienteeritud. Siin tulevadki mängu mikrokontrollerid. Mikrokontroller on sisuliselt väike arvuti ühe kiibi sees, kus on püsimälu programmi hoidmiseks, muutmälu, CPU ja palju muud.

Mikrokontrollereid on erinevaid – alustades väga lihtsatest 8 bitistest ja lõpetades päris võimsate 32 bitiste mikrokontrolleritega, mida kasutatakse näiteks mobiiltelefonides. Sõltuvalt vajadustest saab valida mikrokontrolleri vajalike parameetritega. Käesolevas aines me vaatleme lihtsamaid 8bitilisi mikrokontrollereid AVR ja PIC perekondadest.

Assembler

Te olete programmeerinud varem mingis programmeerimiskeeles, olgu see C, Java, Python või midagi muud. Kõrgema taseme keeles programmeerides te ei puutu väga palju kokku sellega kuidas programmi tegelikult madalamal tasemel täidetakse. Kui te Pythonis kasutate listi, siis enamasti te ei mõtle andmete paigutusele mälus või masinkäskude käivitamisele andmete lisamisel listi. Selleks aga, et normaalselt

programmeerida mikrokontrollereid, on vaja head arusaamist mikrokontrollerite tööst. Selleks peame süvenema madalamal tasemel toimuvasse.

Käesoleva aine raames me hakkame programmeerima Assembler keeles. See on madala taseme keel, mille käsud vastavad üks-üheselt antud platvormi masinkeele käskudele. Kuna igal protsessori tüübil on masinkeel erinev, on erinevad ka masinate assemblerid.

Assembleri programm koosneb käskudest mida täidetakse järjest. Näiteks üks programm, mida me aines uurime näeb välja nii:

```
LDI R16, 0xFF
OUT 0x04, R16
OUT 0x05, R16
INC R16
RJMP 2
```

Assemblerist endast pole võimalik rääkida ilma aru saamata, mis toimub mikrokontrolleri sees. Sisuliselt annab assembler nimed masinkeele instruktsioonidele. Olemasolevate instruktsioonide vaatamiseks saab andmelehe peatükist „Instruction Set Summary“. Teine hea koht instruktsioonide nimekirja vaatamiseks on Atmel Studio help.

Kõige tähtsam instruktsioon, mis on olemas igas arhitektuuris, on „NOP“ (No OPeration), mis ei tee midagi.

NB! Kui te ei saa aru, mis on 0xFF, siis uurige kuusteistkümmend süsteemi kohta.

Taktsagedus

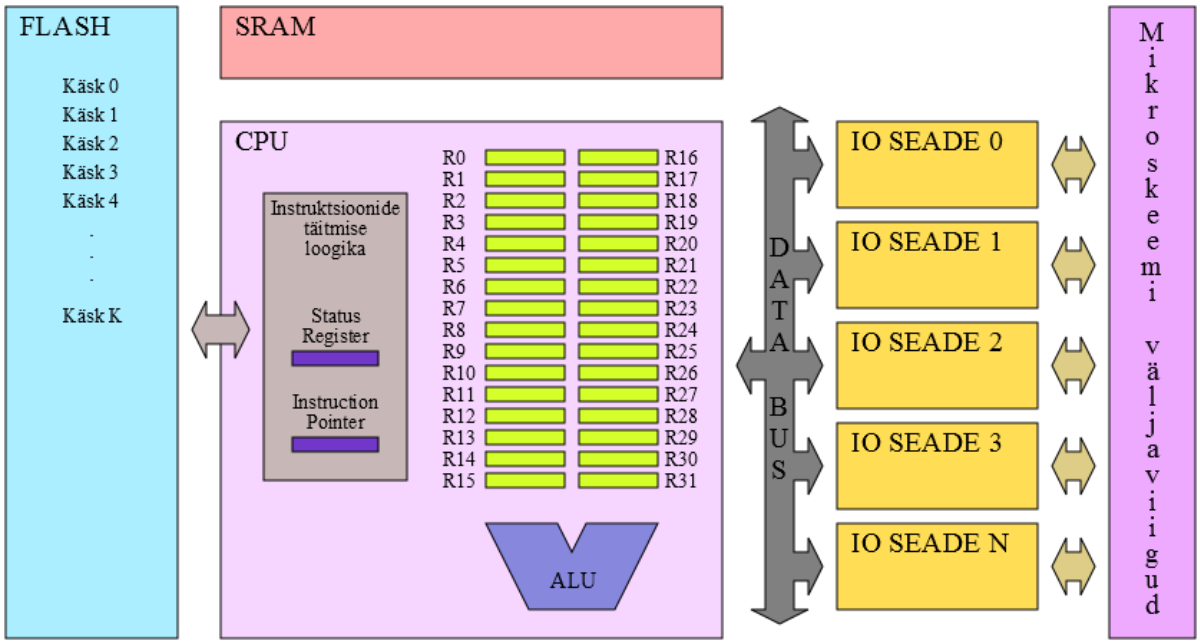
CPU käivitab instruktsioone, e. masinkeele käske. Mis kiirusega seda tehakse? Olete kuulnud, et mingi protsessori kiirus on nt. 2GHz. Mida see tähendab? Kas see tähendab et CPU täidab 2 miljardit instruktsiooni sekundis? Tegelikult mitte. Asi on selles, et instruktsioone ei käivitata pidevalt vaid käivitatakse taktsignaali järgi taktide kaupa. Seda protsessi ei saa peatuda näiteks poole takti pealt.

Mõned käsud võtavad 1 takti, mõned võtavad 2, 3 või rohkem takte. See, mitu takti mingi käsk võtab, sõltub arhitektuurist. Näiteks AVR arhitektuuris on enamus käske 1 taktilised, kuid mõned võtavad 2 või 3 takti. Teise näitena võib tuua Intel ATOM 1.6GHz - see on umbes sama kiire nagu Celeron 900MHz. See näitab,et kuigi ATOMi taktsagedus on suurem, võtab käskude käivitamine rohkem takte.

Iga takti kestvuseks on 1/Taktsagedus sekundit. Näiteks kui taktsagedus on 2MHz, siis iga takti kestvuseks on 0.5 mikrosekundit. Kui me räägime, et mingi protsess võtab näiteks 4 takti tähendab see, et selleks kulub 2 mikrosekundit 2MHz taktsignaali juures. Räägime tihti sagedusest 2MHz, kuna see on AVRButterfly plaadil oleva mikrokontrolleri taktsagedus.

Millest koosneb mikrokontroller

AVR mikrokontrollerite üldine ülesehitus on näidatud järgmisel skeemil.



Mikrokontroller koosneb järgmistest loogilistest osadest:

- FLASH – programmimälu. Seal hoitakse käske, mida tuleb täita ja sealt loetakse ja täidetakse neid ühe kaupa.
- SRAM – muutmälu. Muutujate hoidmiseks mõeldud mälu. See on kiire, aga sellest kaovad andmed ära kui te lülitate toite välja.
- CPU – plokk, mis käivitab instruktsioone. Selle sees on kõige tähtsamad järgmised alamplokid:
    - Üldkasutuseks mõeldud registrid (general purpose registers) – R0…R31. Need on sellised mälukohad millega CPU saab operatsioone teha. Iga register on 8 bitiline.
    - ALU – Aritmeetika ja loogika plokk. See plokk võimaldab näiteks liita kahe registri väärtust ja teha muid aritmeetilisi ja bitikaupa tehteid.
    - Status Register (SR) – See on väga tähtis osa, millest sõltub kuidas käivitatakse mõned käsud. SRst räägime detailsemalt pärast.
    - Instruction Pointer (IP) – instruktsiooni viide. Selles registris hoitakse käivitatava instruktsiooni aadressi. Selle aadressi järgi saab teada, kus FLASH mälus asub see instruktsioon, mida hakatakse täitma. Peale iga instruktsiooni täitmist see suureneb ühe võrra, põhjustades sellega järgmise instruktsiooni käivitamist. Sellest räägime detailsemalt hiljem.
- Data Bus – andmesiin. Selle kaudu info liigub CPU ja IO seadmete vahel. Selleks, et mingi info satuks andmesiinile ja läbi selle kuhugi kindla IO seadme sisse, on olemas spetsiaalsed masinkäsud. Sellest detailsemalt ka hiljem.
- IO Seadmed – nende kaudu mikrokontroller suhtleb välismaailmaga. Mõned nendest on ühendatud mikroskeemi väljaviikudega. Millised IO seadmed maailmas olemas on ja mida nad teevad saame teada hiljem.

Registrid

Register on üks kõige tähtsamaid mõisteid. Üldiselt sõna register tähendab sellist riistvaralist komponenti, kus hoitakse andmeid ja kust neid saab kätte. Mikrokontrollerit puhul on tavaliselt olemas mitut tüüpi registreid:

- üldkasutuseks mõeldud registrid (neid nimetame edaspidi lihtsalt „registriteks“)
- IO registrid (asuvad IO seadmete sees)
- erinevad spetsiaalsed registrid (nagu SR või IP).

Üldkasutuseks mõeldud registriteks (edaspidi lihtsalt registrid) nimetatakse spetsiaalseid riistvaralisi mälukohtasid protsessori sees, mis on ühendatud otse ALUga. Selle ühenduse olemasolu tõttu saab nendega teha aritmeetilisi ja loogilisi tehteid. Registritega saab teha mitmeid operatsioone:

- registritesse saab laadida väärtust
- reistreid saab omavahel liita, lahutada ning nende peal rakendada muid aritmeetilisi instruktsioone
- registritega saab teha bititehteid
- registreid saab võrrelda nii omavahel kui ka konstandiga
- registritesse saab laadida väärtust RAMist või salvestada registri väärtust RAMi
- registreid saab kasutata IO seadmetega suhtlemiseks (sellest täpsemalt hiljem)

AVR arhitektuuris on 32 üldkasutuseks mõeldud registrit ja nad kõik on 8 bitilised. Neid nimetatakse R0…R31. Vaatame üle mõned assembleri käsud, mille abil saab registritega tehteid teha.

Registrisse väärtuse kirjutamine AVR arhitektuuris käib käsuga LDI (LoaD Immediate – laadida registrisse konstant), näiteks käsk

```
LDI R16, 10
```

laeb registrisse R16 väärtuse 10

Ühest registrist väärtuse kopeerimine teisse käib käsuga MOV, näiteks käsk

```
MOV R20, R16
```

laeb registrisse R20 väärtuse, mis oli registris R16, selle juures R16 väärtus ei muutu.

Liitmine käib käsuga ADD

```
ADD R16, R20
```

liidab registris R16 olevale väärtusele registris R20 oleva väärtuse ning tulemuse paneb registrisse R16

Lahutamine käib käsuga SUB (SUBstract)

```
SUB R18, R17
```

lahutab registris R18 olevast väärtusest registris R17 oleva väärtuse ning tulemuse paneb registrisse R18

Ühe võrra suurendamist saaks teha küll ADD käsuga, aga selleks on olemas spetsiaalne suurendamise käsk INC (INCrement)

```
INC R16
```

suurendab R16 väärtust ühe võrra

Käsk DEC (DECrement) vastupidi vähendab ühe võrra registri väärtust

```
DEC R2
```

Vähendab registris R2 oleva väärtuse ühe võrra ja kirjutab tulemuse tagasi R2 sisse

Kui te olete programmeerinud mingis kõrgema taseme keeles (näiteks C), siis teate sellisest kontseptsioonist nagu ***muutuja***. See on mingi informatsiooni tükk, millega saab teha operatsioone. Näiteks 2 täisarvulist muutujat A ja B saab liita. C keeles saaksite kirjutada selle jaoks koodi:

```
int A, B;
A = 10;
B = 20;
A = A+B;
```

Kui mõtlete, kuidas saaks seda realiseerida Assembleris, siis kõige lihtsam moodus olekski asetada muutujad registritesse. Selle koodi analoog Assembleris näeks välja nii (semikooloniga tähistatakse kommentaari):

```
LDI R16, 10  ; hoiame R16 sees muutuja A väärtust
LDI R17, 20  ; hoiame R17 sees muutuja B väärtust
ADD R16, R17 ; A = A+B
```

Seega register on lihtsalt mälukoht, millega saab teha igasuguseid operatsioone. Sellest, et on olemas teistsugused mälukohad, millega EI saa teha aritmeetilise operatsioone,saame teada hiljem (kui räägime RAMist).

Programmi mälu (FLASH)

Arvutis hoitakse tavaliselt programme kõvaketa peal. Kuid programmi käivitamisel loetakse seda kõvaketta pealt muutmällu ning protsessor käivitab käske juba mälust. Erinevalt arvutitest AVR mikrokontrollerites on programmi jaoks eraldi püsimälu, kus programmi hoitakse ja kust programmi ka käivitatakse. See tähendab, et CPU osa mikrokontrollerist loeb programmimälust käsud ükshaaval ja kohe täidab loetud käsku. Kohe, kui mikrokontrollerile antakse toidet, loeb see käsku, mis asub aadressil „0“ ja hakkab seda täitma, siis võtab järgmise käsu. See on väga erinev arvutitest, kus ei hakata täitma kohe toite andmisel kõike, mis asub kõvakettal.

Tavaliselt on programmimälu FLASH tüüpi. Selle maht varieerub mikrokontrollerist mikrokontrollerini ja tavaliselt asub vahemikus 1kB…1MB.

Instruction Pointer

Mikrokontrolleri CPU osa sees on olemas spetsiaalne register (mitte general purpose register) nimega „Instruction Pointer“. See register hoiab endas käivitatava instruktsiooni aadressi. Kohe peale reseti on selles registris väärtus „0“ ja selle tõttu kohe esimese asjana hakatakse käivitama instruktsiooni, mis asub programmimälus aadressil 0. Peale iga instruktsiooni käivitust suureneb IP ühe võrra, sellega viidates järgmisele instruktsioonile, mida koheselt hakatakse täitma.

Üldiselt on selge, et alguses käivitatakse instruktsioon, mis asub FLASHis aadressil 0, seejärel instruktsioon aadressil 1, siis 2 jne… Aga mõnikord pole meil vaja käivitada instruktsiooni, mis asub järgmiseö aadressil, vaid pigem on vaja hüpata mõne teise instruktsiooni peale - näiteks tsükli tegemiseks. Selle jaoks on olemas instruktsioonid, mis muudavad IP väärtust, et jätkata programmi käivitamist kohast, mis ei asu programmimälus järgmisel aadressil. Selliseid instruktsioone on mitu ja andmelehest neid leiab peatüki „Instruction Set Summary“ osast „Branch Instructions“. Kõige lihtsam selline instruktsioon on RJMP (Relative JuMP). See instruktsioon hüppab etteantud aadressile. Näiteks programmis:

```
LDI R16, 0xFF ; see käsk asub aadressil „0“
OUT 0x04, R16 ; me veel ei tea mis see on aga see asub aadressil „1“
OUT 0x05, R16 ; „2“
INC R16
RJMP 2
```

hüppab instruktsioon `RJMP 2` käsule, mis asub aadressil 2, ehk käsule `OUT 0x05, R16,` mida hakataksegi järgmisena käivitama. Tuleb välja, et selles programmis on tehtud lõpmata tsükkel, kus peale 1. instruktsiooni käivitatakse 2., siis 3., siis 4., 5. ja siis jälle 3., 4., 5., 3., 4., 5. jne…

Veel üks näide lõpmata tsüklist oleks:

```
RJMP 0
```

see programm täidab lõpmatult ainsat instruktsiooni, mis hüppab aadressile „0“, kus see instruktsioon ise asubki.

Mida teeb järgmine kood?

```
RJMP 2
NOP
RJMP 0
```

Vastus on olemas siin valge tekstigaKui saate aru, mida see kood teeb, siis veenduge, et saite aru õigesti. Kui saite aru valesti, siis lugege antud peatükk uuesti. VASTUS:

On olemas ka teised instruktsioonid, mis hüppavad kuhugi, aga ***KÕIK sellised instruktsioonid, kaasa arvatud RJMP, teevad seda registri IP („Instruction Pointer“) muutmise abil***. Seega ***IP register on üks kõige tähtsaimad kontseptsioone*** ja see on olemas igas arhitektuuris. Näiteks teie lauaarvutite protsessorites on see ka olemas, aga selle nimeks on PC („Program Counter“).

IO

Tavaliselt kui räägite programmist, siis te kujutate ette mingeid aknaid, nuppe ja muid GUI komponente. Inimeste mõttes tavaliselt assotsieerub programm sellega, kuidas see näeb välja, mitte sellega, mis käske jooksutab protsessor. See illustreerib seda, et tavaliselt programmi mõte ei seisne selles, et lihtsalt käivitada käske vaid programmi töö kasulikuks tulemuseks on mingi mõju välismaailmale. Olgu see ekraanile pildi näitamine, faili salvestamine, üle USB pordi mingi seadmega suhtlemine, autopiloodi poolt lennuki mootorite juhtimine, suhtlemine mingite anduritega või midagi muud. Programm, olles

jooksutatud protsessori poolt, peab kuidagi „tulema protsessorist välja“ ja suhtlema välismaailmaga, muidu sellest programmist pole kasu. Selleks ongi vaja IO seadmeid.

Erinevad IO seadmed täidavad erinevaid funktsioone – mõned on järjestikliidesed, mõned on aja mõõtmiseks, pinge mõõtmiseks, paralleelliidesed, pulsilaiusmodulatsiooni kontrollerid jne. Arvutist rääkides on IO seadmeteks klaviatuurid, hiired, USB kontrollerid, kõvaketa kontrollerid, videokaardid, LAN liidesed jne. Aga kõigil IO seadmetel on üks ühine joon – nad on loogiliselt eraldatud CPUst ja nendega CPU suhtleb läbi andmesiini. Sellega on saavutatud paindlikus , et mikrokontrollerite tootja võib teha mitu mikrokontrollerit sama arhitektuuriga, aga erineva IO seadmete komplektiga. Tegelikult on sellise ülesehitusega saavutatud palju tähtsamaid asju kui lihtsalt paindlikus, aga sellest me ei räägi selle aine raames ☺.

Väljaviikude seisundid

Enne kui jätkame, tuleks aru saada, mis seisundites võib olla mikroskeemi väljaviik. Mõnikord on meil vaja näiteks mingi väljaviigu kaudu juhtida mootorit, mõnel teisel juhul on meil vaja mõõta väljaviigu peal mingist andurist tulevat pinget, kolmandal juhul on meil vaja teada, kas väljaviigu külge ühendatud nupp on all või mitte. Mis erinevused on nende kolme juhu vahel?

Esimesel juhul me tahame mootorit juhtida ehk ***me tahame, et välismaailma (nt. mootori) käitumine sõltuks väljaviigu seisundist***. Seega peab väljaviik mõjutama välismaailma ja selle väärtus peab sõltuma ainult mikrokontrolleri programmist.

Teisel juhul tahame mõõta andurist tulevat pinget - me tahame vastupidi, et väljaviik ei mõjutaks seda andurit ja sõltuks ainult sellest andurist, mitte mikrokontrolleri programmist. Tõesti, kui meil on näiteks andur, mille väljundpinge on võrdeline temperatuuriga, siis me tahaks teada puhast temperatuuri, mitte mikrokontrolleri programmi poolt mingil teadmata moel moonutatud pinget, mis ei pruugi näidata temperatuuri. ***Ehk me tahame, et välismaailm mõjutaks väljaviiku, mitte mikrokontroller***.

Samamoodi on ka kolmandal juhul kui me räägime nupust.

Seega on mõtet rääkida kahest põhilisest väljaviigu seisundist – ***väljund*** ja ***sisend***.

Väljund

Kui väljaviik on ideaalseks väljundiks, siis selle peal olev pinge sõltub ainult mikrokontrollerist. Lisaks mootori kontrollile võib näiteks tuua mikrokontrolleri jala, mille kaudu mikrokontroller saadab infot arvutisse. Me tahame, et arvutisse jõuaks saadetud info, mitte näiteks müra tõttu tekkinud signaal. Me ei taha, et seda mõjutaksid sellega seostamata müraallikad.

Väljundil on omakorda 2 võimaliku seisundit – „0“ ja „1“. „0“ on seisund kui väljundi jalg on mikrokontrolleri sees ühendatud toitepinge negatiivse poolusega (ehk maaga). „1“ tähendab vastupidi ühendust positiivse toitepinge poolusega. See tähendab, et seisund „1“ tegelikult sõltub toitepingest. Näiteks kui me toidame mikrokontrollerit 5V toitega, siis „1“ tähendab, et väljaviigu ja maa vahel on 5V pinge. Kui me toidaks mikrokontrollerit 3.3V toitega, siis „1“ tähendab 3.3V maa suhtes. „0“ on aga

ühendus maaga sõltumata toitepingest, seega selles seisundis väljaviigu peal olev pinge maa suhtes on alati 0V.

Seisundit „0“ nimetatakse ka „madalaks“ ja „1“ – „kõrgeks“.

Sisend

Kui väljaviik on sisendiks, siis vastupidi määrab selle seisundit välismaailm mitte mikrokontroller. Näiteks me tahame, et andurist tulev pinge ei muutuks selle tõttu, et me hakkame seda mõõtma. Teine näide on jalg, mille kaudu arvuti saadab infot mikrokontrollerisse- me tahame, et see info ei muutuks selle tõttu, et me hakkasime seda vastu võtma.

Sisendil on ka mitu võimaliku alamvõimalust – pull-up, pull-down, high-Z, aga sellesse me praegu ei hakka süvenema.

Seega ideaalne sisend sõltub ainult välismaailmast ega mõjuta seda ning väljund vastupidi sõltubki ainult mikrokontrollerist mitte välismaailmast.

Proovige vastata järgmistele küsimustele:

Kas meil on vaja kasutada sidendit või väljundit selleks, et … ?

1. kontrollida valgusdioodi
2. lugeda peegeldusanduri näitu
3. mõõta valgustustaset
4. kontrollida mootorit
5. lugeda mootori keerlemisandurite näitu

Vastused:

Mis pinge on väljaviigu ja maa vahel kui:

1. Mikrokontrolleri toitepinge on 5V, väljaviik on väljundiks ja „0“?
2. Mikrokontrolleri toitepinge on 5V, väljaviik on väljundiks ja „1“?
3. Mikrokontrolleri toitepinge on 5V, väljaviik on sisendiks ja sellega on ühendatud nupp, mis ühendab selle jala +5Vga?
4. Mikrokontrolleri toitepinge on 5V, väljaviik on sisendiks ja sellega on ühendatud nupp, mis ühendab selle jala maaga?
5. Mikrokontrolleri toitepinge on 5V, väljaviik on väljundiks ja „0“ ning sellega on ühendatud nupp, mis ühendab selle jala +5Vga?

Vastused:

Kui teie vastused ei klapi, siis LUGEGE SEE PEATÜKK UUESTI LÄBI – iteratiivselt arusaamiseni.

Kõige lihtsam IO seade

Kui me teame, mis seisundites võivad mikroskeemi jalad olla, siis on kõige lihtsam IO seade selline, mis võimaldab jala seisundit määrata. Sellise IO seadme nimeks on „PORT“. Port võimaldab:

- seadistada jalga väljundiks ja konfigureerida, kas selle väärtuseks on „0“ või „1“
- seadistada jalga sisendiks ja lugeda, kas väljast antakse sinna „0“ või „1“

AVR arhitektuur on 8bitiline ja selle tõttu iga port kontrollib kuni 8 jalga. Porte nimetatakse tähtedega - Port A, Port B, Port C jne.

NB! Port on väga ülekoormatud mõiste. Me kasutame seda tähistades vastavat IO seadet.

IO registrid

Olgu meil olemas mingi seade nimega „Port“, mis on võimeline manipuleerima jalgade seisundeid. Kuidas siis CPU saab öelda sellele pordile, et mingid jalad tuleb seadistada väljundiks ja mõned teised sisendiks? Peab olema mingi moodus teavitada IO seadet sellest, mida me tahame, et see teeks. Samuti peab olema mingi moodus IO seadme käest informatsiooni saamiseks.

Igal IO seadmel on olemas omad nn. „IO registrid“ (mitte sassi ajada CPU üldkasutamiseks mõeldud registritega). Mõned IO registritest määravad IO seadme tööd, teised vastupidi – nende lugemisel saab teada seadme staatust. Kui me vaatame arhitektuuri pilti, siis me näeme, et CPU ja IO seadmed on ühendatud andmesiini kaudu. CPUl on olemas käsud selleks, et andmesiini kaudu lugeda ja kirjutada IO registritesse. Igal IO registril on olemas oma aadress andmesiinil.

Portidel (edaspidi kasutame näidete jaoks Port B) on 3 IO registrit: DDRB, PORTB (mitte sassi ajada PORTB IO registrit Port B IO seadmega) ja PINB.

- DDRB (Data Direction Register B) register määrab, kas jalad on seadistatud sisenditeks või väljunditeks. See register on 8 bitine (nagu kõik registrid ikka 8 bitilises arhitektuuris) ja selle iga bitt vastab ühele mikrokontrolleri jalale. Kui jalale vastavas bitis on „1“, siis antud jalg on seadistatud väljundiks, kui „0“, siis sisendiks.
- praegu lihtsustame PORTB (PORT data register B) registri funktsionaalsust, ja ütleme ainult, et kui mingi jalg on konfigureeritud väljundiks DDRB registri poolt, siis PORTB register määrab, kas see on „0“ või „1“. Jällegi igale jalale vastab üks bitt selles registris (kui tekib küsimusi, siis pöörduge tagasi peatükile, mis kirjeldab väljaviigu võimalikke seisundeid),
- PINB (input PIN value B) register näitab, kas jala peale on väljast antud „0“ või „1“. Juhul, kui mingi jala peal on tekkinud „1“, siis „1“ tekib ka PINB registri antud jalale vastavas bitis.

Selleks, et kirjutada mingisse IO registrisse teatud väärtust, on vaja teada selle IO registri aadressi andmesiinil. Seda saab teada andmelehest iga IO seadme iga registri kohta. Näiteks PORTB registri aadress on 0x05, DDRB aadress on 0x04 ja PINB aadressiks on 0x03.

Port A IO seadmel on ka 3 IO registrit vastavalt nimedega DDRA, PORTA ja PINA. Nende aadressideks on 0x01 (DDRA), 0x00 (PINA), 0x02 (PORTA).

IO registrite poole pöördumise käsud

Me vaatleme siin kahte põhilist käsku millega saab pöörduda IO registrite poole. IO registri lugemiseks on käsk IN ja kirjutamiseks on olemas käsk OUT. Näiteks

```
IN R16, 0x03
```

loeb IO registrist aadressiga 0x03 (ehk PINB) väärtuse CPU registrisse R16. Käsk

```
OUT 0x04, R17
```

kirjutab IO registrisse aadressiga 0x04 (ehk DDRB) väärtuse CPU registrist R17. Programm, mis seadistab Port B kõik jalad väljundiks, võiks näha välja näiteks nii:

```
LDI R19, 0xFF ; alguses laeme väärtuse, kus kõik bitid on „1“, R19
sisse
OUT 0x04, R19 ; kirjutame selle IO registrisse aadressiga 0x04 (DDRB)
```

NB!! AVR arhitektuuris IO registrisse saab kirjutada ainult väärtust CPU registrist. OUT käsu teine operand PEAB olema register, mitte konstant. Seega OUT käsku ei saa kasutada näiteks nii:

```
OUT 0x04, 0xFF ; ei saa kirjutada väärtust (nt. 0xFF) otse IO
registrisse
```

Poril B on 8 jalga – PB0 … PB7. Igale jalale vastab üks bitt registrites. Järgmine programm paneb ainult jala PB0 väljundiks ja teised jalad jätab sisenditeks.

```
LDI R16, 1 ; binaaris on see 0b00000001 ehk ainult alumine bitt on „1“
OUT 0x04, R16 ; seadistame PB0 jala väljundiks, PB1..PB7 sisendiks
```

Proovige vastata järgmisele küsimusele: Mis Port B väljaviike seadistavad järgmised programmid väljundiks?

1)

```
LDI R16, 2
OUT 0x04, R16
```

2)

```
LDI R16, 0b10101010
OUT 0x04, R16
```

3)

```
LDI R16, 0b10101010
OUT 0x05, R16
```

Vastused:

Lugege andmelehe peatüki portidest „I/O Ports” ja veenduge, et saate aru. Siis vaadake peatüki „Register Summary“ ja otsige sealt välja, mis on näiteks registri PINC aadress. Vastus:

Maagiline programm viiest reast

Nüüd me peaksime olema võimelised aru saama, mida teeb see programm, mis käis selles dokumendis juba kaks korda läbi.

```
LDI R16, 0xFF
OUT 0x04, R16
OUT 0x05, R16
INC R16
```

```
RJMP 2
```

- Esimene rida laeb registrisse R16 väärtuse 0xFF, ehk binaarsüsteemis 0b11111111
- Teine rida kirjutab selle väärtuse IO registrisse DDRB, millega põhjustab jalgade PB0…PB7 seadistamist väljundiks.
- Read 3-5 täidetakse tsükliliselt, kuna viies rida hüppab tagasi kolmandale (aadressile 2)
    - Kolmas rida kirjutab registri R16 väärtuse IO registrisse PORTB. Esimesel iteratsioonil see põhjustab kõikide jalgade minekut kõrgesse olekusse. Järgmistel iteratsioonidel tulemus sõltub sellest, mis väärus on R16 sees.
    - Neljas rida suurendab R16 väärtust, et järgmisel tsükli iteratsioonil PORTB läheks registrisse järgmine väärtus. Kui teate binaarsüsteemi, siis saate aru, et see põhjustab jalgade PB0..PB7 peal „kastsignaali“, kusjuures jala PB0 peal on signaali sagedus kõige suurem ja iga järgmise jala peal on see sagedus eelmisest 2 korda väiksem. Kui meil oleksid nende jalgade külge ühendatud valgusdioodid, siis see põhjustaks nende vilkumist erinevate sagedustega.

Selle programmi töö kohta on olemas video – vaadake seda. Videos on olemas FLASHi kõrval väike nool, mis näitab millist käsku praegu täidetakse, ehk registri IP väärtust.

IO peatüki kokkuvõte

Siin on kokkuvõte antud peatükist. Kui midagi jääb ebaselgeks, siis lugege see peatükk uuesti läbi, sest see on baas kogu edaspidise töö jaoks.

- ***Mikrokontroller suhtleb välismaailmaga IO seadmete kaudu***
- ***IO seadmed on ühendatud CPUga andmesiini kaudu***
- ***Oleks liiga keeruline ja ebapaindlik tekitada CPUs eraldi masinkäsk iga IO seadme iga võimaliku funktsionaalsuse kohta. Selle asemel on tehtud vaid paar käsku, millega saab kirjutada/lugeda andmesiinilt, ja sedasi suhelda ükskõik millise IO seadmega.***
- ***IO seadmetel on omad IO registrid. Konkreetse IO registri tähendus sõltub seadmest. Nende registrite kohta saab infot andmelehest.***
- ***Kogu suhtlus CPU ja IO seadmete vahel käib IO registrite kirjutamise/lugemise teel.***
- ***Üheks IO seadme tüübiks, mida me teame, on Port***
    - ***See võimaldab kontrolli mikroskeemi jalgade üle***
    - ***Sellel on olemas 3 IO registrit DDRx, PORTx ja PINx, mille kaudu saab manipuleerida väljaviikude seisundit***

Label (Märgis)

Me vaatlesime juba mõnda näidet, kus kasutame RJMP instruktsiooni. Sellele instruktsioonile anname ette aadressi, kuhu hüpata. Probleem seisneb selles, et programmi algusse ühe rea lisamisega, kõikide järgmiste instruktsioonide aadressid muutuvad. Näiteks kui me lisame programmi

```
LDI R16, 0xFF
OUT 0x04, R16
OUT 0x05, R16
INC R16
RJMP 2
```

ette ühe rea, teeme koheselt selle programmi katki

```
NOP
LDI R16, 0xFF
OUT 0x04, R16
OUT 0x05, R16 ; selle käsu aadressiks on nüüd 3, mitte 2
INC R16
RJMP 2        ; nüüd on see aadress vigane
```

Selleks, et ei peaks selle pärast muretsema ja igat aadressi nihutama uue rea lisamisel, on olemas selline mõiste nagu „Label“ – see on sisuliselt nimetus mingile programmi kohale. Seda kasutades saaksime muuta meie programmi järgmiseks:

```
LDI R16, 0xFF
OUT 0x04, R16
TSYKLI_ALGUS:
      OUT 0x05, R16     ; seda koha programmis nimetasime TSYKLI_ALGUS
      INC R16
RJMP TSYKLI_ALGUS       ; aadressi asemel kasutame labelit
```

Selle programmi kompileerimisel assemblerist masinkoodiks kompilaator ise asendab käsus `RJMP TSYKLI_ALGUS` oleva labeli õige aadressiga. Nii et me ei pea muretsema selle pärast, kuidas meil see aadress muutub kui me muudame koodi, sest alati asendatakse see korrektse aadressiga.

Teine muudatus selles koodis on treppimine. Koodi treppimine ei ole kohustuslik, kuid võimaldab paremini näha koodi struktuuri. See on kasutamiseks väga soovitatud.

Tingimuslik täitmine ja Status Register

Me oskame nüüd kasutada käsku RJMP, millega saab teha näiteks lõpmatut tsüklit. See käsk hüppab alati etteantud kohale. Aga tihti oleks meil vaja hüpata kusagile ainult juhul, kui mingi tingimus on rahuldatud. Näiteks hüppame tsükli algusse ainult esimesed 10 korda, et saada lõplikku tsüklit. Kuidas seda teha? Vaatame järgmist koodi:

```
LDI R17, 10
TSYKKEL:
      NOP
      DEC R17
BRNE TSYKKEL ; kui vähendamise tulemus ei ole null, siis hüppame NOPile
NOP          ; siia me jõuame peale 10 tsükli iteratsiooni
```

See kood laeb alguses registrisse R17 väärtuse 10 ja siis läheb tsüklisse. Tsükli sees R17t vähendatakse. Käsk `BRNE TSYKKEL` (BRanch if Not Equal) hüppab tähise TSYKKEL peale juhul, kui eelmise tehte tulemus ei olnud võrdne 0ga. See tähendab, et peale esimest tsükli iteratsiooni R17 sees on väärtus 9,

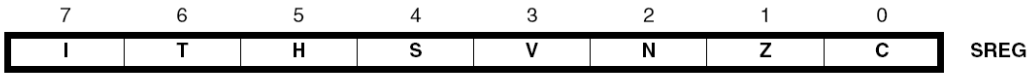
mis ei võrdu nulliga, seega BRanch if Not Equal hüppab tsükli algusse, põhjustades uue iteratsiooni. Teise iteratsiooni lõpus on R17 sees väärtus 8, mis ka ei võrdu nulliga. Selle tõttu BRNE hüppab jälle ette. Ja nii edasi kuni lõpuks 10nda iteratsiooni lõpus `DEC R17` ei anna tulemuseks nulli. Kuna eelmise tehte tulemus oli null, siis BRNE ei hüppa enam algusse ning käivitatakse kuuendal real olev käsk `NOP`. See näitab, kuidas me saame kasutada tingliku käivitamist lõplike tsüklite tegemiseks.

Peale käsku BRNE on olemas palju muid tingimusliku käivitamise käske. Mõned nendest on loetletud siin:

- BREQ – BRanch if EQual, hüppame kui eelmise tehte tulemus oli võrdne nulliga
- SBRS – Skip if Bit in Register is Set, jätab järgmise käsu täitmata ja hüppab otse ülejärgmise peale kui ette antud registris ette antud biti väärtus on „1“.
- SBRC – Skip if Bit in Register is Cleared, jätab järgmise käsu täitmata ja hüppab otse ülejärgmise peale kui ette antud registris ette antud bitti väärtus on „0“.
- BRGE – BRanch if Greater or Equal, hüppame kui eelmise võrdluse tulemusena esimene võrreldav oli suurem või võrdne teisega.
- BRCS – BRanch if Carry Set, hüppame kui eelmise tehte tulemusena tekkis numbriline ülekanne (number, mis peaks olema suurem kui 255 ja sellega ei mahtunud 8 biti sisse)

Nüüd kui olete lugenud tähelepanelikult, siis olete märganud arusaamatuid sõnastusi. Mõne käsu kirjelduses figureerib fraas „kui eelmise instruktsiooni“ või „kui eelmise võrdluse“. Nagu näiteks ka meie lõpliku tsükli koodis `DEC R17` vähendab väärtust, aga `BRNE TSYKKEL` kuidagi teab, kas eelmise tehte tulemus oli null või mitte. Pöörake tähelepanu sellele, et BRNEle ei anta ette registrit, mida tuleks võrrelda nulliga ja see ikka kuidagi teab, mis on „eelmise käsu tulemus“. Samamoodi käsk BRCS teab, kas tehte tulemusena oli tekkinud ületäitmine (biti ülekanne). Kuidas on see võimalik?

Vastus on lihtne – järelikult eelmine käsk pidi kusagile jätma selle info, mille järgi saab teada, kas vähendamise tulemus oli null või mitte, kas tulemus oli positiivne, kas tehte käigus tekkis ülekanne jne. Kohta, kuhu seda infot salvestatakse , nimetatakse Status Register (SREG). SREG koosneb järgmistest bittidest (neid nimetatakse ka lippudeks, kuna nad on mõeldud signaliseerimaks mingi sündmuse tekkimist. Öeldaks, et lipp on „püsti“, kui biti väärtus on „1“ ning lipp on „maas“, kui selle väärtus on „0“):



Bit 7 – I: Global Interrupt Enable
Bit 6 – T: Bit Copy Storage
Bit 5 – H: Half Carry Flag
Bit 4 – S: Sign Bit, S = N ⊕ V
Bit 3 – V: Two's Complement Overflow Flag
Bit 2 – N: Negative Flag
Bit 1 – Z: Zero Flag
Bit 0 – C: Carry Flag

Käsk DEC paneb SREG lipu „Zero“ püsti, kui selle tulemus oli 0, ja tingimusliku käivitamise käsud vaatavad just SREGi väärtust, et teada, kas on vaja hüpata või mitte. Nii käsk BRNE vaatab SREG lippu „Zero“, käsk BRCS vaatab lippu „Carry“ jne. See mehhanism on väga võimas ja on kasutusel KÕIKIDES arvutusmasinates. Teie lauaarvuti protsessoril on ka staatuse register olemas, kuid selle lipud võivad erineda. Lippud Zero ja Carry on olemas igal masinal.

Mõned koodi näited

Eelmises näites lugesime 10st nullini. Mõnikord on aga vaja lugeda numbri suurenemise suunas:

```
LDI R16, 0
MINU_SUPER_TSYKKEL:
      ; teeme vahepeal midagi tarka
      ; kogu tarkus on tehtud, tarkust oli null rida
      INC R16
      CPI R16, 150            ; võrdleme R16 väärtusega 150
BRNE MINU_SUPER_TSÜKKEL       ; kui pole võrdne hüppame üles
```

See kood loeb R16 nullist 150ni, seega teeb midagi tarka (mida praegu koodi sees pole) 150 korda. Selles programmis on kasutatud sellist käsku nagu CPI (ComPare with Immediate), mis võrdleb registri sisu ette antud konstandiga (lahutamise teel) ja muudab mitu lippu SREGis:

- Paneb püsti lipu Zero kui väärtused on võrdsed, võtab lipu maha kui pole võrdsed
- Paneb püsti lipu Negative kui lahutamise tulemus on negatiivne
- Ja muudab veel mitut lippu, mis meie jaoks praegu nii tähtsad pole

Üks tüüpiline viga on see, et LDI käsk pannaks tsükli sisse:

```
MINU_SUPER_TSYKKEL:
      LDI R16, 0              ; see on viga
      ; teeme vahepeal midagi tarka
      ; kogu tarkus on tehtud, tarkust oli null rida
      INC R16
      CPI R16, 150            ; võrdleme R16 väärtusega 150
BRNE MINU_SUPER_TSÜKKEL       ; kui R16 != 150 siis hüppame üles,
                              ; AGA R16 on alati võrdne 1, kuna üleval
                              ; on sellele omistatud 0
```

See põhjustab olukorda, kus tsükkel ei lõppe kunagi ja programm „hangub“. Sellise vea tõttu R16 väärtust võrreldaks 150ga kuid seda nullitakse igal iteratsioonil. Seega iga iteratsiooni lõpus on R16 sees väärtus 1 ja BRNE hüppab ALATI üles.

Nüüd vaatame koodi, mis oli mõeldud valgusdioodide vilgutamiseks:

```
LDI R16, 0xFF
OUT 0x04, R16
TSYKLI_ALGUS:
      OUT 0x05, R16
      INC R16
RJMP TSYKLI_ALGUS
```

Selle programmi tsükli üks iteratsioon koosneb kolme käsu täitmisest, mis kokku võtavad 4 takti aega. See tähendab, et näiteks 2MHz taktsagedusega mikrokontrolleris on kõige kiiremini vilkuva valgusdioodi vilkumise sagedus 2Mhz/4/2 = 250kHz, ning kõige aeglasemalt vilkuva valgusdioodi sagedus on veel 128

korda aeglasem, ehk ~2kHz. Seega seda vilkumist ei ole silmaga näha. Me tahame ikkagi vilkumist näha ja selleks on meil vaja teha igal iteratsioonil viivis. Viivise tekitamiseks teeme lõpliku tsükli, mis ei tee midagi kasuliku, vaid raiskab aega:

```
LDI R16, 0xFF
OUT 0x04, R16
TSYKLI_ALGUS:
      OUT 0x05, R16
      INC R16

      LDI R17, 255
      VIIVIS:
            NOP
            DEC R17
      BRNE VIIVIS

RJMP TSYKLI_ALGUS
```

Viivise tsükli igal iteratsioonil täidetakse ka 3 käsku, mis kokku võtavad 4 takti. Seda tehakse 255 korda, seega kokku peatsükli igal iteratsioonil kulutatakse 4+255*4=1024 takti. Seega kõige kiirema valgusdioodi sageduseks 2MHz taktsageduse puhul on 2MHz/1024/2 ~= 1kHz ja kõige aeglasema valgusdioodi vilkumiskiiruseks on ~8Hz ehk me juba näeme vilkumist silmaga. See näitab, kuidas me saame kasutada lõplikke tsükleid ja kuidas me saame kasutada tingliku täitmise käske.

Kokkuvõte

- ***CPU sees on olemas staatuse register, mis koosneb erinevatest lippudest***
- ***Mõned käsud panevad lippe püsti või võtavad maha sõltuvalt tulemusest***
- ***Mõne teise käsu täitmine sõltub sellest, kas mingi kindel lipp on püsti või maas***
- ***Kogu see süsteem võimaldab teha koodi käivitust sõltuvaks andmetest***
- ***SREG kontseptsioon on ÜKS KÕIGE TÄHTSAMAID BAASASJU, mida on kasutatud igas arvutis juba mitukümmend aastat ja seda kasutatakse veel mitukümmend aastat***
- ***Tingliku täitmise käske saab kasutada lõplike tsüklite tegemiseks, if lause realiseerimiseks assembleris ja paljuks muuks***

Millest koosneb mikrokontroller

Käesolevast dokumendist saite teada, millest koosneb mikrokontroller. Tegelikult see ei käi ainult mikrokontrolleri kohta, vaid käib ka iga arvuti kohta. Antud dokumendis kirjeldatud põhimõtted on kasutusel väga laialt ja jäävad kasutusele veel väga pikaks ajaks. Sellest dokumendist aru saamine on baas kogu aine edukale sooritamisele rääkimata juba sellest, et iga endast lugu pidav programmeerija peab seda teadma. Siin on loetelu asjadest, millest peate aru saama:

- Registrid
- Programmimälu
    - Instruction Pointer
    - Hüppe instruktsioonid (nagu RJMP)
- IO seadmed
    - IO registrid

- Status register
    - Tinglik täitmine ja selleks mõeldud käsud
    - Zero lipp, CMPI instruktsioon, BRNE ja BREQ instruktsioonid
- RAM, millest räägime järgmisel korral

Muutmälu (RAM)

Igas arvutis on olemas muutmälu, mille kogust tänapäeval mõõdetakse gigabaitides. AVR mikrokontrollerites on ka olemas RAM muutmälu. Selle hulk varieerub mõnest sajast baidist mõne kilobaidini. Esmapilgul tundub selline hulk tühine, aga tegelikult saab sellega teha väga palju.

RAMi lugemiseks on olemas käsud LD (LoaD indirect), LDD (LoaD indirect with Displacement) ja LDS (LoaD direct from Sram). Vastavalt nendele käskudele on olemas ka 3 variatsiooni salvestamiskäsust – ST, STD ja STS. Esialgu uurime LDS/STS käskude paari:

```
LDS Register, Aadress
STS Aadress, Register
```

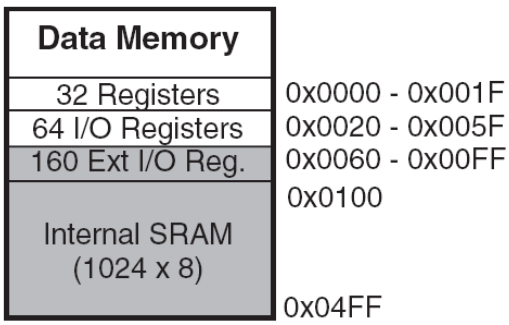
Aadressid ja Aadressiruumid

Kui te pole varem kokku puutunud aadressidega, siis praeguseks hetkeks on neid kogunenud juba kuidagi liiga palju. Me rääkisime aadressidest siis, kui oli jutu FLASHist, teist korda kui oli jutu andmesiini kohta ja nüüd kui räägime RAMist, siis räägime aadressidest veel kord. Kas need on samad aadressid? Mille poolest need erinevad? Mis see aadress üldse on?

Aadress on moodus iga mälukoha nummerdamiseks. Igal FLASHi kohal on oma aadress nullist kuni FLASHi koguseni. Igal IO registril on oma aadress 0st kuni mingi arvuni. Öeldakse, et FLASH ja IO registrid asuvad eraldi aadressiruumides. Aadress ühe aadressiruumi sees määrab mälukohta üks-üheselt, kuid täpselt samasugune aadress võib eksisteerida ka teistes aadressiruumides. Näiteks FLASHi nullinda aadressi eksisteerimine ei välista, et on olemas ka IO register nullinda aadressiga. Seda sellepärast, et FLASH ja IO registrid asuvad erinevates aadressiruumides.

Aadressiruume võib väga lihtsalt „ühendada“. Kui ühes aadressiruumis on N aadressi nullist kuni N-1 ja teises on K aadressi 0 kuni K-1, siis saab teha sellest ühise aadressiruumi N+K aadressiga nullist kuni N+K-1.

Kui vaatame mikrokontrolleri ülesehituse diagrammil RAMi asukohta, siis näeme, et see on CPUst täiesti eraldatud plokk. Aga aadresside mõttes on LDx ja STx instruktsioonide jaoks tehtud eraldi suurem aadressiruum, kus paiknevad nii RAM kui ka IO registrid ja CPU registrid. Seda on näha kõrval oleval pildil. Nii et näiteks:




```
LDS R16, 10
```

laeb registrisse R16 registri R10 väärtuse, sellepärast et aadressil 10 selles aadressiruumis asub register R10. Samamoodi

```
LDS R16, 35
```

laeb registrisse R16 IO registri PINB väärtuse, kuna aadressist 32 algavad IO registrid ja aadressil 35 asub kolmas IO register ehk PINB (mida me eelnevalt oleme lugenud `IN Rx, 0x03` käsuga).

Edasi, kui tahame adresseerida näiteks RAMi esimest (nullindat) baiti, siis kasutame käsku

```
LDS R16, 0x100  ; loeme RAMist väärtuse
LDI R17, 10
ADD R16, R17    ; liidame R16le 10
STS 0x100, R16  ; salvestame tagasi RAMi
```

Muutujad

Siiani oleme hoidnud andmeid registrites. AVR arhitektuuris on olemas tervelt 32 registrit, mille tõttu sinna mahub suhteliselt palju muutujaid. Aga mingi hetk jääb sellest ikka väheks ja siis jõuame RAMi ja registrite kasutamiseni.

Üldiselt hoitakse muutujaid RAMis, mitte registrites. Kuna RAMiga ei saa teha aritmeetilisi tehteid või kirjutada RAMist baite otse andmesiinile, siis iga tehte tegemiseks tuleb laadida muutuja väärtus RAMist registrisse ja peale muutmist salvestada tulemus tagasi RAMi.

Oletame, et meil on programmi alguses vaja initsialiseerida mingid muutujad ja tükk aega hiljem on vaja neid kasutada. Siis üldiselt me ei saa lubada endale hoida need muutujad kogu aeg registrites, sest meil on registreid vaja teiste asjade tegemiseks:

```
unsigned char a, b, c;
a = 10;
b = 20;
c = 0;

// siin teeme midagi väga
// tarka, mille tegemiseks
// meil on vaja kõike
// registreid

// ja lõpus olgu meil vaja
// teha liitmise
c = a + b + c;
```

```
LDI R16, 10
STS 0x100, R16  ; muutuja a

LDI R17, 20
STS 0x101, R16  ; muutuja b

LDI R18, 0
STS 0x102, R16  ; muutuja c

; teeme midagi väga tarka,
; mille tegemiseks meil läheb
; vaja kõiki registreid

; loeme RAMist muutuja a
LDS R16, 0x100

; loeme RAMist muutuja b
LDS R17, 0x101

; loeme RAMist muutuja c
LDS R18, 0x102

; liidame registrites kokku
ADD R16, R17
ADD R16, r18

; salvestame tulemuse
; muutuja c sisse
STS 0x102, R16
```

Esiteks pöörame tähelepanu sellele, selles programmis määrasime käsitsi igale muutujale aadressi. Mõtlesime, et oleks loogiline paigutada muutujad RAMis järjest, alustades RAMi esimesest baidist.

Seega kuna muutujad on meil ühebaidilised, siis nende aadressideks me määrasime 0x100 muutuja `a` jaoks, 0x101 muutuja `b` jaoks ja 0x102 muutuja `c` jaoks.

Teiseks tähtsaks momendiks on see, et programmi alguses (muutujate initsialiseerimise käigus) me salvestame muutujate väärtusi RAMi, mitte ei hoia neid registrites. Selle on mitu põhjendust:

- Meie programmi keskel tehakse midagi tarka ja seal võib neid muutujaid vaja minna. Kui nii, siis see „tark" kood arvatavasti oletab, et RAMis on muutuja väärtus up-to-date . Oletame, et seda tarka koodi ei kirjuta teie, vaid kirjutab mõni teie sõber. Siis ta ei saa mitte kuidagi mõistlikult oletada, et õiget muutuja `a` väärtust (ehk 10) peaks otsima mitte sealt, kus see peaks asuma (RAMist aadressilt 0x100), vaid hoopis mingist registrist.
- Registreid on vähe - ainult 32 (ja tegelikult seegi on väga palju, sest tavaliselt on neid veel vähem - näiteks arvutis on 4 põhilist registrit – EAX, EBX, ECX ja EDX). Kuna registreid on vähe, siis on need on väga väärtuslik ressurss. Seega pole tõenäoliselt mõttekas hõivata kaks-kolm registrit mingite muutujatega, mida meil on vaja ainult kunagi tükk aega hiljem.

Kolmandaks - enne liitmist peame laadima muutujate praegused väärtused RAMist registritesse. Sellel on ka mitu põhjendust:

- Registrite väärtused võisid muutuda „targa" koodi käivitamisega
- Muutujate väärtused võisid muutuda „targa" koodi käivitamisega

Seega, kui me ei saa eeldada, et registrites R16 ja R17 on muutujate `a` ja `b` väärtused õiged, siis peame laadima õiged väärtused RAMist.

Nii et üldiselt kui me räägime muutujatest, siis nad paiknevad RAMis ja just RAMis hoitakse nende viimast (up-to-date) väärtust. Selleks, et teha muutujatega mingeid operatsioone, peame laadima neid RAMist registritesse ning peale operatsioonide tegemist salvestama tulemuse tagasi RAMi.

Muutujate deklareerimine assembleris

Eelmises näites määrasime ise käsitsi muutujate aadressid. Selline lähenemine pole mugav ja on programmivigade allikas. Ideaaljuhul me tahame teha nii, et me lihtsalt ütleme kompilaatorile „reserveeri meile nii mitu baiti RAMis ja nimeta seda muutujaks *muutuja_nimi*". Ja pärast tahame, et *muutuja_nimi* saaks kompileerimise käigus asendatud selle muutuja aadressiga. See säästaks meile palju tööd ja vigu. Siis meie eelmine kood näeks välja nii:

```
.DSEG           ; data segment – siit algab RAMi sisu ehk andmed
a: .BYTE 1      ; defineerime muutuja a öeldes:
b: .BYTE 1      ; „nimeta see b'ks ja tee see palun baitidest, ühest"
c: .BYTE 1      ; ja muutuja c

.CSEG           ; code segment – siit edasi on FLASHi sisu ehk kood
LDI R16, 10
STS a, R16      ; siin asendatakse muutuja a selle aadressiga RAMis

LDI R16, 20
STS b, R16
```

```
LDI R16, 0
STS c, R16

; teeme midagi väga tarka, mille tegemiseks meil läheb vaja kõiki
; registreid

LDS R16, a      ; loeme RAMist muutuja a
LDS R17, b      ; loeme RAMist muutuja b
LDS R18, c      ; loeme RAMist muutuja c

ADD R16, R17    ; liidame registrites kokku R16=a+b
ADD R16, r18    ; R16=(a+b)+c

STS c, R16      ; salvestame tulemuse muutuja c sisse
```

Täpselt nagu programmi sees label annab nime mingile programmi kohale ja kompileerimisel asendatakse see järgmise käsu aadressiga, nii on ka muutujate nimetused sisuliselt labelid RAMi kohtadele –kompileerimise ajal asendatakse need muutujate aadressidega.

Kui vaatame, kuidas on kompilaator paigutanud muutujad mälus, siis näeme, et nad asuvad järjest alustades esimesest RAMi baidist. Eelmise koodi muutujate aadressid on 0x100 (a), 0x101 (b) ja 0x102(c).

Nüüd teame, mis on muutuja. Selle asemel, et öelda: „RAMi koht suurusega X baiti, kus me hoiame neid ja neid andmeid“, me lihtsat ütleme: „muutuja bla“.

Massiivid

Oletame, et meil on vaja teha mitu sama põhimõttega muutujat – näiteks meil on 10 samasugust andurit ja meil on iga anduri jaoks üks muutuja. Või meil on 10 samasugust mootorit ja iga mootori jaoks on vaja mitu muutujat, nt. koordinaat, kiirus, takistusjõud jne. Sel juhul saame teha 10 samasugust muutujat ühe kaupa iga mootori jaoks ja nimetaks neid näiteks mootor_1_koordinaat, mootor_2_koordinaat jne.

Tihti on selline lähenemine ebamugav ja lihtsalt inetu. Selle vältimiseks tehakse massiivid. ***Massiiv on hulk samasuguseid muutujaid, mis on paigutatud RAMis järjest***. Näiteks kümne mootori kiiruse muutuja asemel saame teha ühe massiivi 10st mootori kiirusest.

Teine näide oleks arvutilt info vastuvõtmine - kui võtate infot baithaaval vastu, siis ei saa seda tihti kohe töödelda, vaid tule panna kuhusagile puhvrisse, et töödelda hiljem. Sellise puhvri jaoks saaks kasutada massiivi, kuna pole mõistlik tekitada eraldi 100 muutujat nimedega vastuvõetud_bait_1, vastuvõetud_bait_2, …, vastuvõetud_bait_100.

Vaatame kuidas deklareerida massiivi assembleris ja kuidas see asub mälus:

```
.DSEG
a: .BYTE 1    ; tavaline 1 baidiline muutuja nimega a
b: .BYTE 1
c: .BYTE 1
d: .BYTE 10   ; massiiv 10st ühebaidilisest elemendist
e: .BYTE 3    ; massiiv 3st ühebaidilisest elemendist
f: .BYTE 1    ; jälle üksik muutuja
```

0x100	a
0x101	b
0x102	c
0x103	d[0]
0x104	d[1]
0x105	d[2]
0x106	d[3]
0x107	d[4]
0x108	d[5]
0x109	d[6]
0x10A	d[7]
0x10B	d[8]
0x10C	d[9]
0x10D	e[0]
0x10E	e[1]
0x10F	e[2]
0x110	f
0x111	jne

Kõik need muutujad ja massiivid paigutatakse RAMi järjest alustades esimesest RAMi kohast.

Viit (pointer)

Viidaks nimetatakse muutujat, mille väärtuseks on mingi teise muutuja AADRESS. Selle aadressi järgi saab pöörduda teise muutuja poole. Kuna AVR arhitektuur on 8bittine ja mälu on suurem kui 256 baiti, tuleb adresseerimiseks kasutada 2 kaheksabitist registrit. AVR arhitektuuris on 3 registripaari, mida saab kasutada viitadeks - X(R27:R26), Y(R29:R28), Z(R31:R30). See tähendab, et kui on olemas masinkäsud, mis oskavad kasutada X, Y ja Z registrite väärtusi aadressina, saab selle aadressi järgi teha mingit operatsiooni.

Näiteks selline kood:

```
LDI R16, 25
LDI R26, low(0x100)   ; alumine aadressi bait R26 sisse
LDI R27, high(0x100)  ; ülemine – R27 sisse
ST X, R16
```

salvestab R16 sees oleva väärtuse (25) mälukohta, mille aadress on X registripaari sees (antud juhul panime sinna aadressi 0x100). Register X on viit muutujale, mis asub aadressil 0x100.

See on väga tore, aga milleks seda vaja on? Me saame ju kirjutada selle asemel `STS 0x100, R16`. Siis kood on lühem ja kiirem ka. Milleks siis on vaja viiteid? Asi on selles, et ***mõnikord kompileerimise ajal pole teada, kuhu andmed tuleb salvestada, vaid see selgub ainult programmi käivitamise jooksul***.

Näide: kujutage ette, et võtate baithaaval arvutilt järjest vastu infot. Te peate neid baite salvestama puhvrisse, et pärast töödelda. See, kuhu me kirjutame vastuvõetud baidi, sõltub praeguseks hetkeks vastuvõetudbaitide hulgast. Kui meil pole ühtegi baiti vastu võetud, peame kirjutama puhvri algusse. Kui meil on juba 10 baiti, peame kirjutama 11nda baidi puhvrisse.

Seega aadress, kuhu tahame andmed salvestada, võib selguda programmi käivitamise käigus. Selleks kasutataksegi massiive ja viiteid. Andmete salvestamiseks aadressi saamiseks tuleb võtta massiivi esimese elemendi aadress ja liita sellele indeks massiivi sees.

Vaatame massiivi näite juurde käivat mälu paigutust. Näeme, et massiiv `d` algab aadressilt 0x103. Kui tahame kirjutada selle massiivi viiendat elementi (elementi indeksiga 4), peame võtma massiivi esimese elemendi aadressi ja liitma sellele 4, et saada aadress, kus paikneb element `d[4]`. See aadress võrdub 0x103+4 = 0x107.

0x100	a
0x101	b
0x102	c
0x103	d[0]
0x104	d[1]
0x105	d[2]
0x106	d[3]
0x107	d[4]
0x108	d[5]
0x109	d[6]
0x10A	d[7]
0x10B	d[8]
0x10C	d[9]
0x10D	e[0]
0x10E	e[1]
0x10F	e[2]
0x110	f
0x111	jne

Vaatame, kuidas saaks seda teha assembleris. Olgu meil kirjutatava elemendi indeks ette antud registrite paaris R17:R16 ning registris R18 olgu antud väärtus, mida tuleb sinna kirjutada:

```
.DSEG
massiiv: .BYTE 200   ; massiiv 200 baiti

.CSEG
; teeme midagi tarka
blabla

; selleks ajaks on eelnev programmi osa kuidagi tekitanud R17:R16
; sisse indeksi ja R18 sisse väärtuse, mida tuleb kirjutada
LDI R26, low(massiiv)
```

```
LDI R27, high(massiiv)
ADD R26, R16          ; liidame indeksi aadressi alumisele osale
ADC R27, R17          ; liidame aadressi ülemised osad ülekande lipuga
                      ; nüüd meil on X registrite paaris viide õigele
                      ; elemendile
ST X, R18             ; salvestame väärtuse viite järgi
```

Kui teil on küsimusi 16 bitilise liitmise kohta, siis vaadake eelneva praktikumi seletust.

Pöörame tähelepanu väljendile `low(massiiv)`. Kompileerimise käigus `massiiv` asendatakse selle aadressiga (antud juhul 0x100) ning `low(0x100)` asendatakse arvu 0x100 alumise baidiga ehk 0x00ga. Samuti ka `high(massiiv)` asendatakse arvu 0x100 ülemise baidiga ehk 0x01ga. Kõik need tehted tehakse kompileerimise käigus, neid ei arvuta mikrokontroller mingite masinkäskude abil. Nii tekib registrite paaris R27:R26 massiivi esimese elemendi aadress.

Teise näitena massiividest ja viidetest vaatame massiivide kopeerimist:

```
.DSEG
array1: .BYTE 10
array2: .BYTE 10


.CSEG

; midagi tarka, mis muudab array1 massiivi sisu
; oletame, et selleks ajaks on selles massiivis väärtused
; 20, 21, 22, 23, 24, 25, 26, 27, 28, 29

LDI R16, 0          ; indeks = 0

; paneme X registrite paari sisse massiivi array1 aadressi
LDI R26, low(array1)
LDI R27, high(array1)

; paneme Y registrite paari sisse massiivi array2 aadressi
LDI R28, low(array2)
LDI R29, high(array2)

loop_start:
      ld R17, X+    ; laeme ühest massiivist registrisse
      st Y+, R17    ; salvestame teisse massiivi
      inc R16       ; suurendame indeksit
      cpi R16, 10   ; võrdleme, kas me oleme juba massiivist väljas
BRLO loop_start     ; kui pole, siis läheme järgmisele ringile
```

See kood on põhimõtteliselt väga lihtne. Andmeted kopeerimiseks ühest massiivist teise on meil vaja mõlema kohta viidet. Üks viide (X) viitab kopeeritavale elemendile, teine (Y) viitab sellele elemendile, kuhu andmeid kopeeritakse.

Masinkäsk `ld R17, X+` teeb kaks asja ühe käsuga - laeb Xis oleva aadressi järgi registrisse R17, ja seejärel suurendab Xi. See masinkäsk on mõeldud just selleks, et massiivi elemendid järjest läbi käia . Samuti `ST Y+, R17` salvestab viidatud kohta R17st väärtuse ja seejärel suurendab Y registritepaari väärtust.

Proovige vastata, mida teeb järgmine kood?

```
.DSEG
data: .BYTE 10

.CSEG
LDI R16, 0
LDI R26, low(data)
LDI R27, high(data)

loop_start:
      st X+, R16    ; kirjutame viite järgi, X järelsuurendamisega
      inc R16
      cpi R16, 10
BRLO loop_start
```

Vastus valge tekstiga on siin:

See on väga suur ja keeruline teema, eriti neile, kes pole programmeerimisega kokku puutunud (kahjuks on selliseid tudengeid aines väga palju). Seega kui te ei saa aru, mida see programm teeb, siis ärge kiirustage ja lugege see dokument uuesti läbi. Võib olla tasub isegi eelmiste loengute materjal üle vaadata.

Pinu (stack)

Pinu on veel üks täiesti fundamentaalne arvutustehnika kontseptsioon, mille tähtsust ei saa hinnata enne kui teate, mida sellega tehakse. Kui poleks välja mõeldud pinu, oleks raske isegi ette kujutada, kuidas saaks programmeerida. Kõik, kes on kõrgema taseme keeles programmeerinud, kasutavad pinu igas programmis tihti isegi aru saamata, et asjad käivad pinu kaudu.

Inimkeeles öeldes on pinu koht, kuhu saab andmeid sisse panna ja kust saab neid tagurpidises järjekorras välja võtta. Pinu on nagu mitu taldrikut üks teise peal - peale saab panna veel ühe taldriku ja veel ühe, aga ära võtta saab ainult ülemise taldriku – keskelt kätte ei saa. Seega kui panete virna alguses kollase, siis rohelise ja viimasena punase taldriku, siis saate nad kätte vastupidises järjekorras: alguses võtate ülevalt punase, siis rohelise ja alles lõpus saate kõige alumise – kollase.

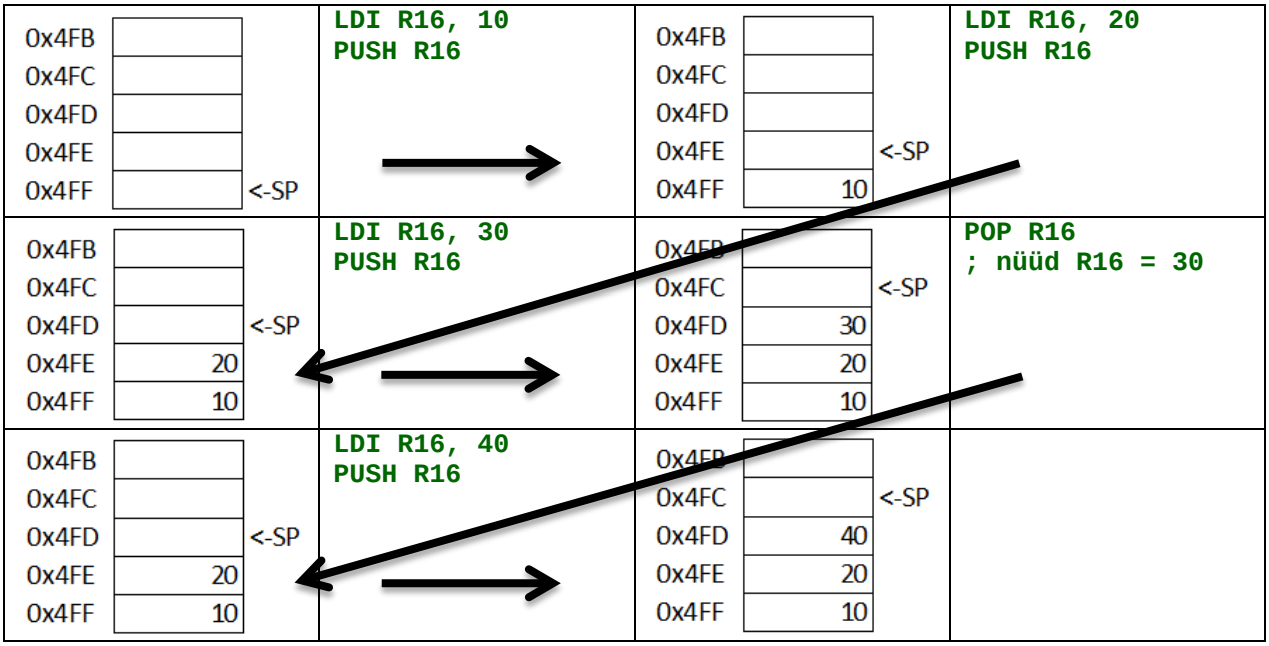
Kui nüüd rääkida riistvaraliselt, siis stacki kontseptsioonis on kolm komponenti:

- RAM, kus pinu hoitakse
- Spetsiaalne 16bitiline register CPU sees nimega Stack Pointer (SP). Selles registris hoitakse viidet kohale, kuhu andmeid hakatakse salvestama.
- Instruktsioonid PUSH (salvestab andmed stackile) ja POP (võtab andmed välja)
    - ***Instuktsioon PUSH salvestab andmed sinna, kuhu viitab SP,seejärel vähendab SP. Nii et järgmine kord PUSH instruktsiooni kasutades salvestatakse andmed teise kohta.***
    - ***Instruktsioon POP teeb vastupidiseid asju vastupidises järjekorras – alguses suurendab SP ja seejärel võtab andmed kohast, kuhu SP viitab.***

RAMi piirkonda, kuhu andmeid sellisel moel salvestatakse, kutsutaksegi „pinu“ või „stack“ või „magasin“.

Kui joonistame RAMi välja nii nagu me tegime RAMi dokumendis, kus väiksemad aadressid on üleval ja suuremad on all, siis näeme, et stack kasvab ülespoole (täpselt nagu taldrikute virn). Sellepärast initsialiseeritakse pinu tavaliselt nii, et alguses SP viitab viimasele RAMi kohale ja andmete lisamisel kasvab see ülespoole. Kui andmeid POPitakse, siis stack väheneb ning SP suureneb.

Vaatame, mis toimub stackiga, kui teeme sellega mingeid operatsioone.



SP näitab noolega, kuhu viitab Stack Pointer. Suured mustad nooled näitavad stacki seisude muutmist. Alguses viitab SP RAMi viimasele kohale (SP sees on väärtus 0x4FF). Esimese PUSHiga salvestatakse pushitav väärtus sinna, kuhu viitab SP ning SP vähendatakse. Seega enne järgmist pushi on SP = 0x4FE. Järgmine PUSH paneb RAMi aadressil e0x4FE väärtuse 20 ning jällegi SP vähendatakse, ja nii edasi.

Väga tähtis on see, et SPt tuleb initsialiseerida enne stacki kasutamist. Kui meil on 1kB RAMi, tuleb ise SP sisse kirjutada väärtus 0x4FF. Kui SP on initsialiseerimata, ei saa ennustada, kuhu stacki salvestatakse. AVR arhitektuuris saab SPle ligi IO registrite näol. SP initsialiseerimine käib nii:

```
LDI R16, 0x04
OUT 0x3E, R16   ; kirjutame SP ülemisse baiti väärtuse 0x04
LDI R16, 0xFF
OUT 0x3D, R16   ; kirjutame SP alumise baiti väärtuse 0xFF
```

Iga programm, mis kasutab pinu, PEAB initsialiseerima SP.

Proovige vastata järgmistele küsimustele:

Mida teeb järgmine programm?

```
PUSH R16
PUSH R17
POP R16
POP R17
```

Vastus valge tekstiga:

Mis väärtus asub RAMis aadressil 0x4FE ja mis on SP väärtus pärast sellise koodi käivitamist?

```
LDI R16, 0x04
OUT 0x3E, R16   ; kirjutame SP ülemisse baiti väärtuse 0x04
LDI R16, 0xFF
OUT 0x3D, R16   ; kirjutame SP alumisse baiti väärtuse 0xFF
LDI R16, 20
PUSH R16
LDI R17, 30
PUSH R17
LDI R18, 40
PUSH R18
```

Vastus valge tekstiga:

Mis väärtus on registris R16 pärast selle programmi käivitamist?

```
LDI R16, 0x04
OUT 0x3E, R16   ; kirjutame SP ülemisse baiti väärtuse 0x04
LDI R16, 0xFF
OUT 0x3D, R16   ; kirjutame SP alumisse baiti väärtuse 0xFF
LDI R16, 20
PUSH R16
LDI R17, 30
PUSH R17
LDI R18, 40
PUSH R18
PUSH R17
POP R16
POP R16
POP R17
```

Vastus valge tekstiga:

Pinu iseenesest ei ole keeruline, aga on väga fundamentaalne teema. Veenduge, et saate sellest aru enne jätkamist.

Milleks pinu kasutatakse?

Nüüd me teame, et on olemas mälu koht, mida nimetatakse pinuks ja millega saab teha PUSH ja POP operatsioone. Milleks see hea on? Kõige tähtsamaks pinu kasutuseks on funktsioonide (protseduuride, alamprogrammide - edaspidi need terminid loetakse sünonüümideks) kutsed. Enne funktsioonide kutsete vaatamist selgitame välja, mis on protseduur. Need, kes on varem programmeerimisega tegelenud, teavad kindlasti, mis on protseduur.

Protseduurid

Enamasti ei kirjutata koodi lihtsalt ühe suure käskude hunnikuna, vaid proovitakse jagada loogilisteks plokkideks. Sellel on mitu põhjust:

- Kui kood on kirjutatud ühe suure hunnikuna, siis on sellest võimatu aru saada. Kujutage ette 20000 rida assembleri koodi, mis lihtsalt tulevad järjest ja pole kuidagi loogiliselt üles ehitatud. Sellist koodi on võimatu lugeda ja parandada. Arusaadavuse parandamiseks jagatakse kood loogilisteks plokkideks.
- Tihti on nii, et tahame mõnda koodijuppi kasutada mitu korda. Üks võimalus on seda kopeerida igasse kohta, kus me seda kasutame, aga see on väga halb lähenemine. Esiteks ühe koodijupi 20 koopiat võtavad palju programmimälu. Teiseks, kui selles kopeeritud koodis oli viga, siis me peame otsima välja kõik kohad, kuhu see kopeeritud oli ja parandama. Palju mõistlikum on eraldada taaskasutatav kood eraldi protseduuriks ja kutsuda seda protseduuri mitmest kohast välja.
- Tihti ei kirjutata programmi üksi, vaid koodi kirjutavad mitu programmeerijat koos. Sel juhul ainus võimalus ülesandeid jagada on jaotada programm loogilisteks tükkideks ja anda erinevatele inimestele tegemiseks võimalikult vähe seotud tükke. Näiteks kui programmis on vaja juhtida mootorit ja lugeda mingi anduri näitu, siis saab jagada seda tööd nii, et üks inimene teeb näiteks anduri lugemise protseduuri, mis paneb andurist loetud väärtuse kuhugi muutujasse. Teine inimene ei pea mõtlema selle peale, kuidas see tehtud on, vaid lihtsalt kutsub seda välja, et anduri näitu teada saada.

Tegelikult on väga raske seletada inimesele, kes pole varem programmeerinud, et ei saa lihtsalt kirjutada programmi, vaid peab tegema veel mingisuguseid protseduure. Need, kes veel pole ise tundnud , et kood kasvab suureks ja koledaks, peavad lihtsalt uskuma, et koodi võiks jaotada loogilisteks juppideks.

Proovime tekitada sellist juppi, mida saab kutsuda välja mitmest kohast. Üks näide koodist, mida me juba oleme teinud ja tahame korduvalt kasutada, on viivitus. Oletame, et meil on selline programm:

```
; teeme midagi tarka

; siis teeme viivituse
LDI R16, 0xFF
viivituse_tsükkel:
      NOP
      DEC R16
BRNE viivituse_tsükkel

; siin teeme midagi veel targemat

; ja siis jälle tahame viivitust
LDI R16, 0xFF
viivituse_tsükkel2:
      NOP
      DEC R16
BRNE viivituse_tsükkel2
```

Me näeme, et meil on koodis 2 samasugust kohta, mida tahame eraldada protseduuriks ja siis seda koodi välja kutsuda. Tulemus peaks olema, et meil on 2 programmijuppi – peaprogramm ja viivituse protseduur. Peaprogramm muutub palju lihtsamaks, kuna asendame iga viivituse protseduuri kutsega:

```
; teeme midagi tarka

KÄIVITAME VIIVISE PROTSEDUURI
```

```
; siis teeme midagi veel targemat

KÄIVITAME VIIVISE PROTSEDUURI
```

Ja viivituse protseduur näeb välja selline:

```
viivituse_protseduur:
      LDI R16, 0xFF
      viivituse_tsükkel:
            NOP
            DEC R16
      BRNE viivituse_tsükkel

      HÜPpAME TAGASI SINNA, KUST SEDA PROTSEDUURI VÄLJA KUTSUTI
```

Ainus küsimus seisneb selles, kuidas käivitada seda protseduuri ja kuidas hüpata protseduurist tagasi. Selleks, et paremini probleemist aru saada, proovime teha hüppamist RJMP käsuga. Siis peaprogramm muutub selliseks:

```
; teeme midagi tarka

RJMP viivituse_protseduur

; siis teeme midagi veel targemat

RJMP viivituse_protseduur
```

Protseduuris aga tekib probleem:

```
viivituse_protseduur:
      LDI R16, 0xFF
      viivituse_tsükkel:
            NOP
            DEC R16
      BRNE viivituse_tsükkel

RJMP ????? ; kuhu me peame siit hüppama ?
```

Probleem seisneb selles, et protseduuris oleks meil vaja hüpata tagasi, aga selleks on vaja teada, kust protseduuri välja kutsuti välja. Seega oleks meil vaja salvestada seda infot kusagile. Selle jaoks on olemas spetsiaalsed instruktsioonid – CALL ja RET (RETurn).

CALL hüppab ette antud aadressile (nagu RJMP), aga enne seda salvestab PINUSSE aadressi, kuhu tuleb tagasi hüpata ehk PC+2. Instruktsioon RET võtab pinust tagasihüppe aadressi, ning hüppab sinna. Kuna PC on 2 baidiline, siis pinu iga CALL operatsiooniga kasvab 2 baidi võrra, ning iga RET instruktsiooniga kahaneb 2 baidi võrra. Seega näeb meie programm välja nii:

```
; initsialiseerime pinu, kuna CALL ja RET kasutavad seda
; teeme midagi tarka

CALL viivituse_protseduur

; siia hüppame esimest korda protseduurist väljudes
; siis teeme midagi veel targemat

CALL viivituse_protseduur
; siia hüppame teist korda protseduurist väljudes
```

```
; programmi lõpp

viivituse_protseduur:
      LDI R16, 0xFF
      viivituse_tsükkel:
            NOP
            DEC R16
      BRNE viivituse_tsükkel

RET
```

Esimese CALL käsuga pannakse pinusse tagasihüppamise aadress ning minnakse viivise protseduuri sisse. Kui viivise protseduur saab täidetud, siis RET käsk võtab pinust selle aadressi ja hüppab sinna tagasi. Sedasi toimuvad kõik protseduurikutsed kõikides arhitektuurides ning pinu ja selle kasutus protseduuride kutsumiseks on üks fundamentaalsetest asjadest.

Pinu kasutamine võimaldab kutsuda funktsiooni seest teisi funktsioone või veel kord sama funktsiooni (seda kutsutakse rekursiooniks). Selle näide on toodud järgmises koodis, kus peaprogrammist kutsutakse välja pika viivise protseduur, mis omakorda kutsub välja lühema viivise protseduuri.

```
; initsialiseerime pinu, kuna CALL ja RET kasutavad seda
; teeme midagi tarka

CALL pika_viivituse_protseduur
NOP ; siia me hüppame esimest korda pika_viivituse_protseduurist
    ; väljudes


CALL pika_viivituse_protseduur
NOP ; siia me hüppame teist korda pika_viivituse_protseduurist
    ; väljudes

; programmi lõpp

pika_viivise_protseduur:
      LDI R17, 0xFF
      pika_viivise_tsükkel:
            CALL lühema_viivituse_protseduur
            DEC R17 ; selle käsu aadress pannakse pinusse ja siia
                    ; hüpatakse lühema_viivituse_protseduurist väljudes
      BRNE pika_viivise_tsükkel


lühema_viivituse_protseduur:
      LDI R16, 0xFF
      lühema_viivituse_tsükkel:
            NOP
            DEC R16
      BRNE lühema_viivituse_tsükkel
RET
```

Sellise programmi käivitamisel toimub järgmine: alguseks kutsutakse välja pika viivise protseduur, mille välja kutsumine lisab pinusse tagasihüppe aadressi. Selle protseduuri sees toimub lühema viivise protseduuri kutse, mille puhul lisatakse pinusse veel üks tagasihüppe aadress (käsu **DEC R17** aadress).

Siis on 2 pinus tagasihüpe aadressi, millest esimene võetakse välja kui hüpatakse lühema viivise protseduurist välja ning teine võetakse välja kui hüpatakse pikema viivise protseduurist välja.

1. Praktikum

Sissejuhatus, bititehted

Sissejuhatus:

Selle praktikumi eesmärgiks on edaspidi kasutatava riistvaraga tutvumine. Lisaks sellele tutvutakse praktikumi käigus arendusvaheniga Atmel AVR Studio 5 (Edaspidi “AVR Studio”) ja kirjutatakse mõned lihtsamad programmid. Väga oluline on kohe esimest praktikumist alates teha tutvust ka dokumentatsiooniga, mille lugemine võib esialgu natuke raske tunduda aga, hiljem aitab hea dokumentatsiooni tundmine ülesandeid oluliselt kiiremini lahendada.

Ülesanded on mõeldud iseseisvaks lahendamiseks. Probleemide tekkimisel pöörduge kindlasti praktikumi juhendaja poole.

Töövahendid:

1x AVR Butterfly plaat koos patareiga
1x RS-232 kaabel (COM)
1x USB-A -> RS-232 üleminek (Kui arvutil puudub RS-232 järjestikport.)

Atmel Atmega 169p andmeleht
Atmel AVR Butterfly andmeleht (ing. k. *datasheet*)
AVR Studio

Tähistused:

Juhendis esineb mitmesuguseid arve, mis erinevad nii väärtuselt, kui ka arvusüsteemilt. Kõik ilma erimärgistuseta arvud on kümnendsüsteemis. Kuueteistkümnendsüsteemis olevad arvud on kujul 0hxx, kus xx on arv kuueteistkümnendkujul. Kõik arvud, mis on kujul b’01010101’ on kahendsüsteemis.

Ülesanded:

Lõpmatu tsükkel :)

Eesmärgiks on katsetada AVR Studio simuleerimist ja õppida uue projekti tegemist ning kompileerimist.

a. Kui selle ülesande tegemisel tekib raskusi, siis vaata loengu slaide ja loengus kasutatud videot.
   i. Loo AVR Studios uus projekt.
   ii. Kirjutada lõpmatu tsükkel, mille sees on vähemalt üks „nop“ käsk. Miks on see käsk vajalik? [1]
   iii. Simuleerida programmi täitmist AVR Studio simulaatori abil.

Valgusdiood ja lõpmata tsükkel

Eesmärgiks on õppida programmi mikrokontrollerisse laadimist ning mikrokontrolleri jalgade seisundi muutmist. Lisaks alustame andmelehe kasutamisega.

a. Selle programmi tulemusena peab kolmas valgusdiood põlema.
   i. Loe hoolikalt läbi andmelehe peatükk portide kohta.
   ii. Teha programm, mis konfigureerib PORT B kolmanda jala väljundiks ning läheb lõpmatusse tsüklisse. Milleks on vajalik lõpmata tsükkel programmi lõpus? [2]
   iii. Lae see programm mikrokontrollerisse. Kuna AVR Studio 5 ei toeta AVR butterfly sisse programmi laadimist, erinevalt AVR Studio 4st, siis programmi sisselaskmiseks tuleb kasutada WinAVR paketist “avrdude” programmi. Selleks tuleb:
      1. Tõmmata ja Installida WinAVR
      2. Leida mis kausta tekitatakse teie projekti .hex faili
      3. Käivitada command prompt (Start->Run->cmd) kus minna selle kataloogi sisse
      4. Käivitada käsku avrdude -c butterfly -p m169 -P COM7 -e -v -U f:w:loeng1_a.hex:i kusjuures COM7 asendada õige COM pordi numbriga ja programminimi.hex õige hex faili nimega.
      5. Oodata kuni avrdude proovib võtta ühendust plaadiga, ja hoides joysticku vajutatud tsenterasendis teha mikrokontrollerile reset.

Lõplik tsükkel

Eesmärgiks on aru saada lõplikute tsüklite ülesehitusest.

a. Loengus näidatud tsükkel
   i. Simuleerida loengus antud viivisega programm AVR Studio abil.
   ii. Laadida see mikrokontrollerisse ja vaadata mis toimub. Kas toimub see mis peaks? Miks?
b. Loengus oli selline lõplik tsükkel, mis vähendas registri väärtust, ja lõpetas itereerimise kui väärtuseks oli null. On olemas ka teised võimalused.

i. Teha tsükkel, kus registri väärtust suurendatakse 0st 0xFFni ja itereerimist lõpetatakse kui see on 0xFF.
ii. Simuleerige programmi läbi.

Pikk lõplik tsükkel

Eesmärgiks on teha selline viivis, mis kestaks 1000000 takti.

a. Mis on ülesandes 2.a ja 2.b tehtud tsüklite maksimaalne pikus kui me piirame nopide arvu 7ga? [3]
b. Mida saaks teha et seda suurendada? [4]
c. Mõtle see välja ja implementeeri ära.
    i. Tee valgusdioodi vilgutamiste vahel viivis 1000000 takti.
    ii. Mis sagedusega nüüd vigub kõige kiiremini vilkuv valgusdiood?

Aritmeetilised tehted

Eesmärgiks on välja selgitada aritmeetilistele tehetele vastavad instruktsioonid ning õppida neid kasutama. Lisaks sellele uurime 8-bitilise aritmeetika piiranguid.

a. Laadida kaks arvu vabalt valitud registritesse ja sooritage nendega aritmeetilisi tehteid (ADD, SUB, MUL, ...). Peale tehte sooritamist peab program sisenema lõpmatusse tsüklisse. Kasutage iga tehte toimimise uurimiseks simulaatorit. Korrake erinevate arvudega.
    i. Mis juhtub, kui liidate 100 ja 200?
    ii. Mis juhtub, kui lahutate 100-st 200?
    iii. Kas on oluline milliseid registreid te tehete sooritamiseks kasutate? [5]
    iv. Kust saate infot käskude tööpiirkondade kohta? [6]

Piezo kõlar

Uurida välja mis mikrokontrolleri jala külge on ühendatud piezo kõlar ja väljastada sinna selline samasugune ruutsignaal nagu ka valgusdioodile, ainult et helisagedusega. Proovida erinevad sagedused läbi. Teha see nii et samal ajal vilguks ainult see valgusdiood mis on kõlariga sama mikrokontrolleri jala küljes. Ehk siis ainult üks PORT B jalg peab olema konfigureeritud väljundiks, mitte kõik.

Vastused juhendis toodud küsimustele:

1. Et programmi täitmine oleks simulaatorist selgemalt näha.
2. Asi on selles et meie programm koosneb ainult mõnest käsust. Kui seda kirjutatakse FLASH mällu siis kirjutatakse üle ainult mõned esimesed FLASH mõned baidid (ntx 10 vms). Ülejäänud FLASHi baidid jäävad muutmata, ehk seal on mingi varasema programmi tükk. Selle programmi tüki käivitamise tulemust ei ole võimalik ette teada, juba selle pärast et me ei tea mis programm seal on. Seega me peame tagama et meie programmi käivitamise lõpus ei hakata käivitama teadmata koodi. Selleks ongi vajalik programmi lõpus lõpmata tsükkel panna.
3. 2550
4. Seda saaks suurendada kasutades seitsme nopi asmel omakorda veel tsüklit.
5. Võite kasutada erinevaid registreid, mitte ainult r16 ja r17. NB! mõned assembleri käsud töötavad ainult osade registritega. Näiteks käsk LDI on võimeline opereerima ainult registritega r16 kuni r31.
6. Selleks, et olla kindel millises aadressivahemikus Assembleri käsk töötab, tuleks vaadata selle käsu kirjeldust AVR Studio kasutusjuhendist.

1 Kahendsüsteemi tahvlipraktikumis osalemine

a. Enne praktikumi vaadata läbi http://www.youtube.com/watch?v=qdFmSlFojIw

2 16-bitiline aritmeetika

a. Laadida LDI käsu abil registritesse kaks 16-bitilist arvu. Kuna mõlemad arvud on 16-bitised, siis igaüks nendest võtab kaks registrit. Olgu meil arv A, mille ülemised 8 bitti (AH) on registris r17 ja alumised 8 bitti (AL) registrisr16. Levinud tavaks on suurema numbriga registris hoida ülemist arvu osa. Arv B asub registrite paaris r19:r18(BH:BL). Liita need kokku (kasutada instruktsioone ADD ja ADC).
   i. Esiteks tuleb liita omavahel arvude alumised osad, kasutades selleks instruktsiooni ADD. Juhul, kui summa on suurem kui 0xFF, sätitakse *statuse registri "carry" lipp* (ülekanne) 1ks.
   ii. See bitt oleks nagu 9s bitt, mida tuleks liita arvude ülemiste osade summale. Just seda teeb instruktsioon ADC, mille abil tuleks liita arvude ülemised osad.
   iii. Millistes registrites on tulemus?
   iv. Kontrollige erinevate arvudega, et liitmine käib õigesti - kaasa arvatud sellistega, kus alumise osa liitmise tulemus on > 255 (ehk tekib carry).

b. Selleks, et eelmise praktikumi 4ndat ülesannet lahendada ilma kolme üksteise sees olevat tsüklit kasutamata, tuleb kasutada ühe 16-bitilise arvu suurendamist (*nop*'e võib olla kuni 20).
   i. Suurendatavat arvu hoiame näiteks registrite paaris R17:R16 ja sellele liidetavat arvu (arvu 1) hoiame registrite paaris R19:R18. Iga viivise iteratsiooniga liidame registritele R17:R16 registrid R19:R18 nii, et tulemus satuks tagasi registritesse R17:R16 ja R19:R18 ei muutuks.
   ii. Oletame, et tahame teha 50000 iteratsiooni, sest iga iteratsioon võtab 20 takti. Selleks, et tsükli lõpetamise vajadust tuvastada, tuleb võrrelda registreid R17:R16 50000ga kasutades CPI käske. Kui registrite paaris R17:R16 olev väärtus == 50000, tuleb iteratsioon lõpetada. See tähendab, et R17 peab olema võrdne 195ga ja r16 80ga (kui ei saa aru miks, siis olete halvasti osalenud esimeses ülesandes ⇒ Küsige juhendaja käest).
   iii. Realiseerige sellise viivitusega valgusdioodi vilgutamise programm ja kontrollige, kas vilkumise sagedus on enam-vähem 1 Hz.

3 Bititehted

a. Lugeda läbi dokument bititehete kohta. Teil pole mõtet seda lugeda kui teate väga hästi, mis on mask ja milleks kasutatakse OR, AND ja XOR tehteid (mitte mida nad teevad, vaid milleks kasutatakse).

b. Teha programm, mis joysticku “vasakule” vajutamise peale paneb 0nda valgusdioodi põlema. Selleks on vaja
   i. Lugeda AVRButterfly andmelehest, millise pordi millise jala külge on ühendatud joysticku “vasakule” vajutamise nupp.
   ii. Konfigureerida valgusdiood 0 jala väljundiks (ainult 0. valgusdioodmitte kogu port!)
   iii. Konfigureerida joysticku soovitud suunale vastav jalg pull-up režiimi. Kes ei saa aru miks, küsib praktikumi juhendaja käest. See nõuab Ohmi seaduse tundmist.
   iv. Lugeda joysticku vasakule suunale vastava pordi PINx registri väärtust
   v. Nullida kõik selle väärtuse bitid peale seda, mis vastab meie tahetud vasaku suuna jalale (AND tehe abil)
   vi. Kui AND tulemuse väärtus on 0, siis järelikult PINx registri nupu jalale vastavas bitis oli null (sest me nullime ülejäänud bitte AND operatsiooniga) ja kui AND tulemus pole null, siis järelikult tolles bitis oli 1.
   vii. See, kas alla vajutatud nupule vastab 0 või 1, tuleb vaadata AVRButterfly andmelehest (elektriskeemist ja selleks on vaja teada Ohmi seadust), või leida teadusliku nuputamise meetodiga, kuna võimalusi on ainult 2.
   viii. Alla vajutatud nupule vastava AND tehe tulemuse korral panna 0. valgusdiood põlema
   ix. Demonstreerida, et see töötab.

c. Programm, mis iga nupu vajutuse peale paneb põlema järgmise valgusdioodi. Mõned joysticku suunad on ühendatud PORTBga, aga teised on ühendatud PORTEga. Asi on selles, et me ei saa korraga kasutada PORTB jalga nii väljundiks (valgusdioodi jaoks) kui ka sisendiks (joysticku jaoks). Seega, kui tahame kasutada kõiki 8 valgusdioodi (ehk kogu PORTB peab olema väljundiks), siis saame kasutada ainult neid joysticku suundi, mis on PORTE külge ühendatud.
   i. Konfigureerida kogu PORTB väljundiks
   ii. Konfigureerime PORTE joystiku soovitud suunale vastava jala pull-up rezhiimi.
   iii. Hoiame LEDide põlemismaski näiteks registris R16. Teeme nii, et alguses ei põle ükski LED (ehk R16 algväärtustame nulliks).
   iv. Teeme lõpmata tsükli, mille sees tehakse järgnevat:
      1. sisseehitatud lõpmata tsüklis oodatakse joysticku mingis suunas vajutamist (analoogiliselt ülesandega 3.b ehk loetakse PINE registrit ja ANDitakse õige maskiga, et nullida kõik bitid

peale vajaliku. Kui tulemus on null, siis väljutakse vajutuse ootamise tsüklist).

2. teises sisseehitatud lõpmata tsüklis oodatakse nupu lahti laskmist (analoogselt vajutusele)
3. kui programmi käivitus on jõudnud siia, siis järelikult on nupp korra alla vajutatud ja vabastatud. Järelikult siin tuleb panna põlema järgmine LED. Kui R16 väärtus (punkt 3.c.ii) on null, laeme sinna ühe, kui pole null, siis käsuga LSL nihutame vasakule. Kirjutame PORTB registrisse R16 väärtuse.

Praktikum nr 3

Massivid

1) Teha programm, mis:
    a. Tekitab mällu 10 baidilise massiivi,
    b. Initsialiseerib selle – massiivi i-nda elemendi väärtuseks peab olema 10*i+15
    c. Käib teist korda massiivi läbi, et arvutada kõikide elementide summa

    Simuleerida programmi käivitamist ja jälgida. mis juhtub registriga X (Y, Z kasutate just seda pointeriks). Jälgida, kuidas muutub mälu sisu programmi käivitamisel.

    Mis muutub kui enne massiivi tekitamist on RAMis lisaks allokeeritud teised muutujad, näiteks:
    a: .BYTE 1
    b: .BYTE 1
    c: .BYTE 1

2) Teha programm, mis:
    a. Tekitab mällu kaks 10 baidilist massiivi – m1 ja m2
    b. Initsialiseerib massiivi m1 – massiivi i-nda elemendi väärtuseks peab olema 10*i+15
    c. Paneb m2[i] = sum(m1[0] ... m1[i]) ehk i-nda elemendi sisse läheb summa massiivi m1 elementidest 0 kuni i.

    Simuleerida.

Protseduurid

1) Lugeda läbi dokument Stacki, Stack pointeri ja protseduuride kohta.
2) Teha programm, mis:
    a. initsialiseerib Stack Pointeri nii, et see viitaks RAM'i lõppu (edaspidi alati iga programm, mis kasutab stacki, algab SP initsialiseerimisega)
    b. laeb registrisse r16 väärtuse 10
    c. läheb tsükli sisse, mille iga iteratsiooniga teeb mitu PUSH r16 ja POP r16. Uurida, mis juhtub kui PUSHe on rohkem kui POPe ja vastupidi. Kas peale igat iteratsiooni on SP sama või erinev?
3) Simuleerida järgmine programm:

```
;siia teha SP initsialiseerimine
Blaaabla

LDI r17, 10
LDI r18, 20
PUSH r17
PUSH r18
POP r17
POP r18
```

Mida see programm teeb?

4) Teha kõige lihtsam protseduur nimega StupidProcedure, mis koosneb ühest NOPist ja RET käsust. Teha programm, mis kutsub seda protseduuri mitu korda järjest välja (sisuliselt sama, mis on teie poolt loetud dokumendis). Simuleerida ja uurida eriti hoolikalt, kuidas muutub StackPointer CALL ja RET instruktsioonide käivitamisel. Samuti uurida eriti hoolikalt, kuidas muutub RAMi sisu CALL instruktsiooni käivitamisel.
5) Kui on selge, mis toimub, siis lisada veel üks protseduur nimega SmarterProcedure, mis kutsub kaks korda järjest välja StupidProcedure. Peaprogrammis kutsuda ühe korra StupidProcedure ja ühe korra SmarterProcedure. Simuleerida kuni arusaamiseni stacki tööst (vaadata eriti hoolikalt SP ja RAMi muutmisi).
6) Teha LEDi vilgutamise programm nii, et:
    a. Viivis oleks eraldi protseduurina.
    b. Protseduuri kutsumiseks peab kasutama instruktsiooni CALL.
    c. Protseduuri lõpus peab olema instruktsioon RET.
    d. Viivise protseduuri käivitamine peab kulutama enam-vähem 0.5 sekundit.
    e. Peatsüklis tehakse iteratiivselt järgmist:

```
Kolm korda
    ;LED põlema
    CALL delay
    ;LED kustu
    CALL delay

Kolm korda
    ;LED põlema
    CALL delay
    CALL delay
    CALL delay
    ;LED kustu
    CALL delay

Kolm korda
    ;LED põlema
    CALL delay
    ;LED kustu
    CALL delay

10 korda
    CALL delay
```

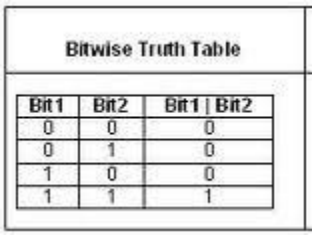
Kui näete SOSi, siis võite koju minna ☺

Bititehted

See juhend on põhiliselt neile, kes EI OLE kunagi kokku puutunud bititehete ja loogiliste operaatoritega. Kes on väga kindel oma bititehete teadmistes võib selle vahele jätta ja hakata pihta otse nupu ülesandega. Teil pole mõtet seda lugeda kui teate väga hästi, mis on mask ja milleks kasutatakse OR, AND ja XOR tehteid.

Millest see dokument räägib?

Kui hakkate otsima internetist bititehete kohta, siis leiate kohe, et on olemas hunnik tehteid: AND, OR, NOT, XOR, nihutamine paremale/vasakule, rotate paremale/vasakule. Te leiate isegi tõeväärtustabelid, mis näevad välja midagi sellist:



Bitwise Truth Table

Bit1	Bit2	Bit1 | Bit2
0	0	0
0	1	0
1	0	0
1	1	1

(Väike kõrvalepõige: see tabel on võetud google pildiotsingu esimeselt lehelt, AGA see on vale. Väärtused vastavad AND tehtele, kuigi ülemises reas on OR tehte tähistus. Seega ei tohi üleliigselt usaldada internetti!)
Aga mida see tabel tegelikult tähendaks (kui see oleks õige) ja miks on nendel tehetel sellised nimetused - AND, OR, NOT?

Inimkeeles

Alustame sellest, et kõik andmed arvutis on esindatud bittidena. Pärast esimest loengut ja praktikumi (loodetavasti enamus teadis seda ka varem) me teame, et ühes baidis on 8 bitti. Iga biti väärtus saab olla kas 0 või 1. Neid olekuid nimetatakse ka “loogiline 0” ja “loogiline 1”. Sellised nimetused on tulnud formaalsest loogikast, kus igal “lausel” on olemas oma tõeväärtus. Teisisõnu iga lause on kas tõene või väär, kolmandat võimalust pole. Ja kuna bitil on ainult 2 võimalikku olekut, siis lause tõeväärtust saab esitada biti väärtusega. “Loogilist 1” nimetatakse ka tõeks ja “loogilist 0” vääraks.

Näide:
Võtame lause “väljas on külm”. Formaalse loogika seisukohast on see kas tõene või väär. Kui väljas on tõesti külm, siis ütleme, et selle lause väärtus on 1 (ehk tõene) ja kui väljas on tegelikult hoopis soe, siis “väljas on külm” = 0 ehk selline väide on väär, sest see on vastuolus tegelikkusega.

Loogiliste väärtustega saab teha mõned operatsioonid. Esimesena vaatleme operatsiooni “JA” (AND).
Näide:

Võtame lisaks lausele “väljas on külm” (edaspidi lühendame seda tähega K sõnast ’külm’) teise lause “ma tahan välja minna” (edaspidi lühendame seda M nagu ’minek’).
Kokku on meil neli võimalust:

1. K=0, M=0 ehk mõlemad laused on väärad ehk: “ma ei taha välja minna ” ning “väljas pole külm”
2. K=0, M=1 ehk ma tahan välja minna, väljas pole külm
3. K=1, M=0 ehk ma ei taha välja minna, väljas on külm
4. K=1, M=1 ehk ma tahan välja minna, väljas on külm

Nüüd, teades K ja M väärtusi, tahame teada, kas me peame mütsi pähe panema. Sellele vastab lause “tuleb soe müts pähe panna” (P nagu ’pähe’).
Ilmselgelt me ei hakka mütsi pähe panema kui me tahame kodus olla. Samuti me ei taha mütsiga käia kui väljas on soe. Seega “tuleb soe müts pähe panna ” ainult siis, “kui väljas on külm” _JA_ “ma tahan välja” minna. Teisiti võime seda kirjutada P = (K and M). Selge, et P on tõene ainult siis, kui mõlemad K ja M on tõesed.

Tavaliselt kui inimene loeb seda teksti esimest korda, siis tal tekivad mõtted nagu “mis müts? Ma ei taha mingit mütsi, ma tahan bititehetest aru saada”. Aga tegelikult nende lausete asemel võivad olla suvalised laused, näiteks “AVR Butterfly nupp on alla vajutatud” või “tuleb valgusdiood põlema panna” või midagi muud. Põhiline on see, et biti väärtust saab vaadelda mitte ainult numbrina, vaid ka tõeväärtusena ja siis kõik bititehete nimetused muutuvad kohe loogilisteks (inimloogilisteks mitte formaalloogilisteks ;).

Tehete defineerimiseks kasutatakse ka tõeväärtustabeleid.

K	M	K and M
0	0	0
0	1	0
1	0	0
1	1	1

Loogilise AND tehte kohta saab tuua palju näiteid. Lause “Ma tahan süüa ja ma tahan juua” on ilmselgelt tõene ainult siis, kui ma korraga tahan mõlemat. Kui kas või üks nendest ei kehti, siis antud lause muutub vääraks. On ju vale öelda “Ma tahan süüa ja ma tahan juua” juhul kui te ei taha süüa. Samuti on vale seda öelda siis, kui te ei taha juua.

Samamoodi muutub arusaadavaks tehe OR. Et sellest aru saada, mõelge välja lause, mis koosneb kahest poolest, mis on omavahel ühendatud sõnaga “või” (lause ei tohi olla konstruktsiooniga “kas... või …”). Vaadake, kas see kehtib kui mõlemad pooled on vväärad. Vaadake, kas see kehtib kui üks pooltest on väär ja teine on tõene ja kas see kehtib kui mõlemad pooled on tõesed. Te peaksite saama tulemuseks järgmise tõeväärtustabeli:

Truth Table for Integral Types

Bit1	Bit2	Bit1 | Bit2
0	0	0
0	1	1
1	0	1
1	1	1

(see tabel on juba õige kuna | tähendab OR tehet)

NOT (eitus) on erinevalt ANDist ja ORist ühe operandiga tehe. Näiteks lause “nupp ei ole alla vajutatud” on tõene ainult siis, kui “nupp on alla vajutatud” on väär.

NOT gate	
A	$\overline{A}$
0	1
1	0

Viimane tähtis tehe on XOR (exclusive or). See vastab konstruktsioonile “kas …. või …”. Näiteks “kass on kas elus või surnud” (loomulikult kui ta pole Schrödingeri oma). Ilmselgelt me valetame kui väidame, et kass on korraga surnud ja elus (mõlemad operandid on tõesed) või kui väidame, et kass on korraga mitte surnud ja mitte elus (mõlemad operandid on väärad). Seega XOR tõeväärtustabel on samasugune nagu OR oma, selle erinevusega, et mõlemad väärtused ei saa korraga olla tõesed.

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Enne kui liigume baitide juurde, veel niipalju, et kui uurite internetis, näete mõnikord selliseid tehteid nagu NOR, XNOR ja NAND. Need on lihtsalt kahe tehte kombinatsioonid: NOR = NOT OR, XNOR või NXOR = NOT XOR ning NAND = NOT AND. See on vastavalt tehete OR, XOR ja AND eitus.

Baitide juurde

Me nägime, kuidas bititehted opereerivad bittidega, aga mida teevad nad baitidega?

Kui räägime tehtest, mis võtab ühe operandi (NOT), siis seda rakendatakse lihtsalt igale baidi bitile. NOT 10010110 =

= 01101001

Kui räägime tehetest, mis nõuavad 2 operandi, siis on meil vaja 2 baiti, et neid rakendada. Võtame näiteks tehte AND. Tulemuse 0is bitt on esimese ja teise baidi 0ndate bittide AND. Samamoodi on teiste bittidega.

100**1**0010 AND
010**1**0100 =
000**1**0000

Tulemuses tulevad ühed ainult kohtadele, kus mõlema operandi vastavates bittides olid ühed. Samamoodi rakendatakse ka teisi bititehteid baitidele.
Arvutage, mis on tehte 11001011 OR 11101110 tulemus. XOR samade operandidega?

Selgub, et kui meil ei ole üks bitt, vaid on mitu bitti, saab:
1) kavalalt kasutada bititehteid, et manipuleerida kindlaid bitte
2) leiutada rohkem tehteid

Kaval bittide manipuleerimine

Me kõik mäletame, kuidas eelmises praktikumis pidime piezo kõlarile väljastama heli nii, et selle käigus muutuks ainult ühe jala seisud. Kuidas saaks ühe bititehtega muuta ühe biti seisundi vastupidiseks ilma teisi bitte muutmata ükskõik millise baidi väärtuse korral? Oletame, et tahame muuta 5ndat bitti teisi bitte muutmata (numeratsioon algab 0'st ja paremalt) ehk me tahame saada sellist transformatsiooni:

algväärtus	00**0**00000	10**1**01010	10**1**10000	01**0**11001	????????
soovitud tulemus	00**1**00000	10**0**01010	10**0**10000	01**1**11001	Sama, ainult 5. bitt on vastupidine

Tuleb välja, et niimoodi suvalise baidi muutmiseks piisab seda XORida 00100000ga.
Selleks, et saada aru, miks see on nii, paneme tähele, et ükskõik, millise biti XOR nulliga ei muuda seda bitti (0 XOR 0 = 0 ja 1 XOR 0 = 1) ning ükskõik, millise biti XOR ühega muudab selle biti vastupidiseks (0 XOR 1 = 1 ja 1 XOR 1 = 0). Seega tuleb alati XOR'ida sellise väärtusega, kus iga meie soovi järgi muudetava biti väärtus sellelkohal on 1 ja kõik teised on nullid (seda väärtust nimetatakse maskiks). Kui tahame muuta 0., 2. ja 3. bitti, siis peame XORima maskiga 00001101. Kui tahame muuta 1'st, 4'ndat, 6'ndat ja 7'ndat bitti, siis vajalikuks maskiks on 11010010 jne.
NB!! Näeme, et saame XOR tehet kasutada selleks, et MUUTA KINDLAID bitte baidis selle baidi väärtusest sõltumata, kusjuures teha seda teisi bitte MUUTMATA!

Samamoodi tuleb välja, et me saame näiteks suvalist bitti teha 0ks kui me ANDime seda 0ga.
Näiteks: ???????? AND
00111100 =
00????00
Iga bitt, mille kohal maskis on 0, läheb 0ks ja iga bitt, mille kohal maskis on 1, jääb muutmata.
Koostage mask, mis nullib 0., 2. ja 3. bitti suvalises baidis.
NB!! Näeme, et saame kasutada AND tehet selleks, et NULLIDA bitte baidis selle baidi väärtusest sõltumata, kusjuures teha seda teisi bitte MUUTMATA!

Samamoodi tuleb välja, et saame suvalise biti teha 1ks, kui me ORime seda 1ga.
Näiteks: ???????? OR

00111100 =
??1111??

Siia peaks tulema samasugune punane tekst nagu eelmised kaks, mõelge see ise välja.

NNNNNNB!!! need 3 punast teksti on kõige tähtsam osa kogu sellest dokumendist. Kui te pole kindel, et saite asjast väga hästi aru, siis lugege seda dokumenti iteratiivselt (korduvalt), arusaamiseni. Kindlasti küsige praktikumi juhendajalt abi, kui iteratsioonide arv ületab 3.

Leiutame rohkem tehteid

Veel üks asi, mida saaksime teha, kui meil on rohkem kui üks bitt, on neid nihutada. Näiteks kui meil on olemas arv 00110101, siis saaksime kõiki bitte ühe võrra paremale nihutada ja saaksime 00011010 (näeme, et nullis bitt kaob ja seitsmenda väärtuseks tuleb 0). Samuti saaksime seda ühe võrra vasakule nihutada ja tulemuseks saame 01101010 (näeme, et 7. biti väärtus kaob ja 0. biti väärtuseks tuleb 0). Selliseid tehteid nimetatakse nihutamiseks (logical shift left/right).

Sarnaseid tehted on veel - näiteks pööre (rotate). *Rotate* ja *shift* vahe seisneb selles, et *rotate*‘i puhul baidi äärmine bitt ei kao, vaid läheb teise äärde. Seega 00110101 *rotate right* ühe biti võrra annaks tulemuseks 10011010.

Kui on keegi, kes on lugenud seda dokumenti siini, siis selleks, et teid tappa, ütlen, et on olemas veel kaks võimalikku *rotate*‘i versiooni - *rotate* ja *rotate through carry*. Üldiselt on kasutusel ainult viimane versioon, kus äärmist bitti ei nihutata teise äärde, vaid hoopis *CARRY* lipu sisse (*Status registris*, vaata eelmise loengu slaide kui sellega on probleeme) ja *carry* lipu väärtus läheb juba teise äärde. Seda võib kujutada üheksanda bitina, mis ei asu baidis, vaid kusagil mujal. Siis *rotate through carry* taandub *9-bitiseks rotate*’ks.

Stack, Stack Pointer

Olete kuulanud loengut pointerite ja massiivide kohta. RAMi aga ei kasutata ainult muutujate hoidmiseks. Lisaks pointeriregistritele X, Y ja Z on olemas veel üks spetsiaalse tähendusega pointer register - Stack Pointer (SP) ja instruktsioonid PUSH ja POP.

- PUSH Rd salvestab registri Rd sisu RAMi kohta, mille aadress on SPs ja vähendab SPt ühe võrra
- POP Rd loeb registrisse Rd väärtus RAMi kohast, mille aadress on SPs ja suurendab SPt ühe võrra

Teisisõnu stack on selline mälu regioon, kuhu saab andmeid järjest sisestada (PUSH) ja tagurpidises järjekorras välja võtta (POP).

Näiteks programm:

```
LDI r17, 10
PUSH r17
POP r18
```

Paneb läbi stacki R18 väärtuseks selle väärtuse mis oli r17s (ehk 10). Tundub, et sellest on üsna vähe kasu, aga see pole päris nii.

Alamprogrammid, CALL, RET

Stackil on väga kasulik rakendus – alamprogrammid.

Varem, kui meil on vaja programmi mitmes erinevas kohas viivist, siis me kopeeriks viivise koodi, mis on väga inetu. Selliseid koodijuppe, mida on vaja mitmest kohast kasutada, tehakse eraldi alamprogrammidena ning kutsutakse välja CALL instruktsiooni abil. CALL on mõnes mõtes sarnane RJMPiga, sest ta hüppab ette antud aadressile, aga lisaks hüppamisele PUSHib veel stacki selle aadressi, KUST ta hüppas. On olemas CALL vastandinstruktsioon – RET, see POPib stackist selle aadressi, mille on sinna salvestanud CALL ning hüppab sinna tagasi. Nende kahe instruktsiooni abil saame tekitada alamprogramme.

Näide:

```
;initsialiseerime SP, et see oleks võrdne 0x4FFga
LDI 16, 0xff
OUT SPL, r16
LDI r16, 0x04
OUT SPH, r16

; teeme midagi tarka
Tsykkel:
        LDI r16, 10
        OUT 0x05, r16
        CALL ViiviseProtseduurike
        LDI r16, 20
        OUT 0x05, r16
```

```
        CALL ViiviseProtseduurike
        LDI r16, 25
        OUT 0x05, r16
        CALL ViiviseProtseduurike
RJMP Tsykkel

ViiviseProtseduurike:
        ; teeme midagi tarka, näiteks
        NOP
RET
```

Praks nr 4.

Protseduurid

Arvestades et keegi pole jõudnud kolmandat praktikumi lõpetada, siis 4s praktikum koosneb ühest ülesandest ja ühest lisaülesandest kõige kiirematele. Lisaülesanne on eraldi failina moodlis üleval (kui pole üleval siis tekib).

1) Tekitada seline viivise protseduur, millele saab reigstrite paaris R17:R16 ette anda viivise pikkust millisekundites. Selleks:
    a. Tekitage selline viivise protseduur mis kulutab täpselt 1 millisekundi aega (võtab 2000 +- 10 takti), nimetage seda protseduuri delay_1ms.
    b. Tekitage protseduur, milles on selline tsükkel mis 16bitise aritmeetika abil teeb R17:R16 sees ette antud iteratsioonide arvu. Igal iteratsioonil see kutsub delay_1ms.
    c. Tehke SOS programm selle protseduuri peale ümber.

Praktikum 4 lisaülesanne

Realiseerida FIFO (first in first out) andmestruktuur, kuhu saab andmed lisada ja kust andmed võtta baidi kaupa.

Kuidas töötab FIFO

On olemas RAMis mingi massiiv ntx 100 baiti. Lisaks on olemas 2 muutujat first_used ja first_free. First_used viitab esimesele massiivi elemendile kus on andmed sees ja first_free esimesele elemendile, mis on tühi. Massiivi täidetakse tsükliliselt (kui viimane massiivi element saab täis siis pannakse jälle algusse, kui seal on tühi). Kui see on täis siis loomulikult sinna lisada ei saa. See on nati seletatud järgmise lehe peal.
Fifoga töötamiseks oleksid vajalikud järgmised protseduurid: FIFO_add, FIFO_get, FIFO_isEmpty ja FIFO_isFull. Kõik need protseduurid peavad olema korralikult protseduuridena vormistatud – salvestavad kasutatavaid registre jne.
Selle ülesande lahendust pärast saab veel täiustada kui saate teada katkestutstest ja aine lõpuks on sellel väga hea kasutus ka olemas, aga sellest te saate teada hiljem...

MASSIIV, elemendi numbrid näidatud															first_used	first_free
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		

empty: kui first_free ja first_used on võrdsed siis on tühi

															0	0

lisada bait "30": kas FIFO on täis? Kui pole täis siis paneme first_free elemendi sisse ja suurendame first_free

30															0	1

lisada bait "10": kas FIFO on täis? Kui pole täis siis paneme first_free elemendi sisse ja suurendame first_free

30	10														0	2

lisada bait "20": kas on täis? Kui pole täis siis paneme first_free elemendi sisse ja suurendame first_free

30	10	20													0	3

võta üks bait: kas on tühi? Kui pole siis võtame first_used seest, ja suurendame first used

	10	20													1	3

võta üks bait: kas on tühi? Kui pole siis võtame first_used seest, ja suurendame first used

		20													2	3

võta üks bait: kas on tühi? Kui pole siis võtame first_used seest, ja suurendame first used

															3	3

jne

													5		13	14

lisada bait "20": kas on täis? Kui pole täis siis paneme first_free elemendi sisse ja suurendame first_free

													5	20	13	0

lisada bait "15": kas on täis? Kui pole täis siis paneme first_free elemendi sisse ja suurendame first_free

15													5	20	13	1

lisada bait "25": kas on täis? Kui pole täis siis paneme first_free elemendi sisse ja suurendame first_free

15	25												5	20	13	2

Jne

15	25	13	14	15	16	17	18	19	20	21	22		5	20	13	12

Nüüd on täis, sest kui me lisame veel üks element siis me saame et first_free ja first_used on võrdsed, ja me ei suuda eristada seda seisundit tühj FIFO seisundist. Seega täis seisund on siis kui first_used on ühe võrra suurem kui first_free, või siis first_used on 14 ja first_free on 0.

Praks nr 5.

IO seadmete kasutamine

1) Lugege andmelehe peatüki taimerite kohta. Uurige välja mida tuleb teha taimeri initsialiseerimiseks. Tehke programm mis initsialiseerib taimerit ja läheb lõpmata tsüklisse. Millist taimerit kasutate praegu vahet pole. Veenduda et simulatsioonis taimer töötab. Selleks tuleb vaadata kas vastav TCNT register muutub aja möödudes.
   Ülesande eesmärgiks on õppida initsialiseerima taimer.
2) Teha programm millel on protseduur delay (nagu ka 4'nda praksi esimeses ülesandes, millele ka antakse ette 16bitilise milisekundite arvu) mis aja lugemiseks ei kasuta delay_ms vaid kasutab taimerit. Selleks see programm peab alguses initsialiseerima taimeri nii, et selle väärtus suureneks 0.5 millisekundi tagant. Delay protseduuris tuleb alguses seadistada vastav TCNT register nulliks ja siis oodata kuni selle väärtus saab võrdseks kahekordsele etteantud väärtusele. Võib kasutada ka Compare match võimalust.
3) Teha programm mis initsialiseerib UARTi baud rate'iga 9600, no parity, 1 stop bit, 8 data bits, ja lülitab sisse saatmismooduli. Tehke protseduur mis saadab ühe tähe üle UART'i. See tähta antakse protseduurile mingis registris (näiteks r24), see protseduur peab ootama seni kuni saatmine on lõpenud (tx complete) enne väljumist. Siis tehke et programm tsüklis saadaks teie nimi üle UART'i. Ühendage oma AVRButterfly arvutiga COM pordi abil, pange käima Termite programmi ja nautige oma nime ekraanil ☺
   a. NB! Bootloader jätab UCSR0A sees U2X biti püsti, nii et kui te tahate et UCSR0A oleks null siis seda tuleb sinna kirjutada.

Praks nr 6.

IO seadmete kasutamine

1) Lugeda andmelehe peatükk ADC kohta ja teha programm mis üle UARTi ei saada teie nime, vaid saadab tähe '0'..'9' sõltuvalt sellest mis näidu annab potentsiomeetri pinge mõõtmine. Kui näit on vahemikus 0..100 siis saadetakse '0', kui näit on vahemikus 101..200 siis saadetake '1' jne... '9' saadetakse kui näit on vahemikus 900..1023.
    a. Et lihtsustada ADC debugimist võib alguses teha selline programm, mis tsüklis mõõdab ADCt ja selle näidu alumise baidi paneb PORTB peale. Siis te kohe näete kas ADC töötab või mitte.
    b. NB! Potentsiomeeter on ühendatud ADC 7nda kanali külge.
    c. NB! Mikrokontrolleri igal pinil on mitu funktsiooni. Nii ADC 7s kanal on ühendatud PF7 pini külge, mis on kasutusel ka JTAG liidese poolt. See natuke mõjutab mõõdetavat pinget. Kui tahta tõsta mõõtmise täpsust, siis tuleks lülitada JTAG välja. Selleks tuleb programmi algusse lisada järgmine koodi jupp:

```
in r16, MCUCR
ldi r17, (1<<JTD)
or r16, r17
out MCUCR, r16
out MCUCR, r16
```

2) Teha PWM juhtimine valgusdioodile ADC näidust lähtuvalt.
    a. Valida valgusdiood, mis on ühendatud kas OC1A, OC1B külge.
    b. Konfigureerida Timer/Counter 1 nii et see töötaks „Fast PWM 10 bit" rezhiimis
    c. Konfigureerida ADC potentsiomeetri pinge mõõtmiseks
    d. Peatsükklis mõõta ADC abil potentsiomeetri signaali ja väljastada ADC mõõtmistulemus teie valitud valgusdioodile vastavasse OCR1x registrisse, see peaks muutma PWMi täituvust ja te peaksite nägema et valgusdiood muudab heledust kui te krutite potentsiomeetrit

Katkestused

1) Teha programm mis kasutab taimeri ületäitmise katkestust ja selle abil vilgutab LEDi (katkestuses flipib LEDi seisundit), peatsüklist tuleb kogu aeg saata üle UARTi teie nime.
2) Teha programm mis kasutab taimeri COMPARE MATCH katkestust LEDi vilgutamiseks. Peatsüklis võib ühe nupu vajutamise peale muuta arvu millega võrdlus toimub, seega tehes vilkumise kiirust sõltuvaks nuppu vajutamisest.
3) Teha programm mis seadistab UARTi nii saatmiseks kui ka vastuvõtmiseks. UARTi RxComplete katkestuses tehakse analüüsi mis väärtus tuli : kui tuli sümbol '1'..'9' siis pannakse led vilkuma vastava kiirusega 1..9 korda sekundis (ledi vilgutatakse timer0 katkestusest). Juhul kui saadud sümbol on '0' siis ledi kustutakse ära. Ja juhul kui väärtus on 'X' siis pannakse kogu aeg põlema.