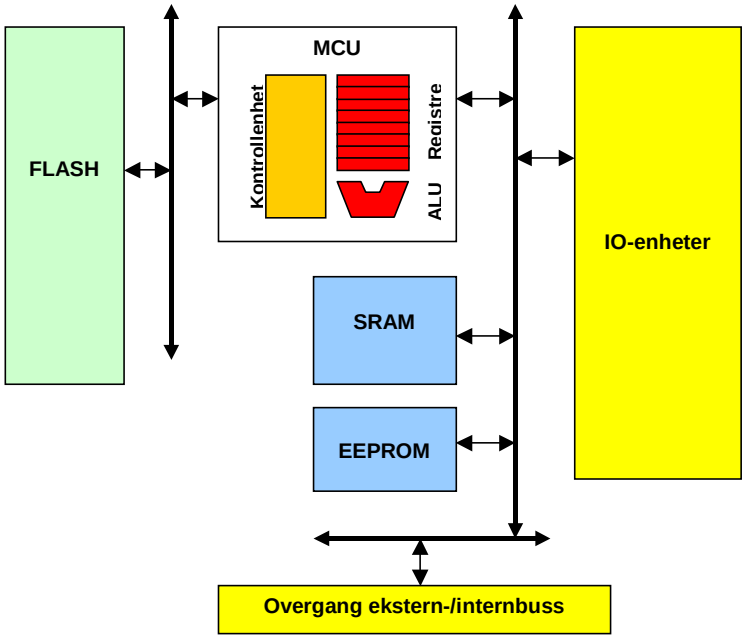
Introduksjon til mikrokontrollere

med hovedvekt på AVR-familien

og

assemblyprogrammering



Høgskolen i Østfold

November 2002

Åge T Johansen

Forord

Dette kompendiet er ment til bruk i emnet IRE 11502 Datateknikk – Hardware og Software ved elektrolinjens 1. år ved Høgskolen i Østfold. Kompendiet er ment å dekke deler av pensum i dette faget – nærmere bestemt den modulen som omhandler mikroprosessorer og assemblyprogrammering.

Sammen med dette kompendiet bør studenten anskaffe kortversjonen av databladet til Atmel AT90S2313 AVR mikrokontroller, hvor bl. a. fullt instruksjonssett er listet opp. I tillegg kan det være en fordel å ha tilgang til fullt datablad for samme komponent og en fullstandig beskrivelse av alle instruksjoner. Alt dette kan hentes fra Atmel på Internett: http://www.atmel.com.

Jeg anbefaler også at man leser gjennom kapittel 9 i MANO/KIME: LOGIC AND COMPUTER DESIGN FUNDAMENTALS som behandler mye av det teoretiske stoffet i dette kompendiet på en mer generell måte enn det som kan gjøres her, hvor gjennomgangen er tett knyttet opp til en enkelt komponent.

Som bakgrunn for kompendiet regner jeg det som en fordel at leseren har en grunnleggende forståelse av følgende temaer:

- Generell kunnskap om en datamaskins oppbygning
- Forskjellig typer programmeringsspråk
- Enkel bruk av flytskjemaer
- Tall og tallsystemer, spesielt det binære og heksadesimale tallsystemet og 2's komplement
- Boolsk algebra og logiske funksjoner

Åge T Johansen

1 Innledning

I dette notatet skal vi studere hvordan en **mikrokontroller** er bygget opp og virker. En mikrokontroller er en digital elektronisk komponent - en IC[1] - som ofte utgjør hele styringsenheten i et elektronisk produkt til industri- eller hjemmebruk. En mikrokontroller er en form for datamaskin.

1.1 Mikrokontrollere

Før vi forklarer nærmere hva en mikrokontroller er, skal vi først se på begrepet **mikroprosessor**. En mikroprosessor er den komponenten som inneholder hele sentralenheten (CPU) i et datasystem. Mikroprosessorer stammer fra 1971, da den første mikroprosessoren Intel 4004 så dagens lys. I dag kjenner vi mikroprosessorene som bl. a. CPU-enhetene i PC-maskiner. Intel Pentium-brikker er en av de ledende her.

I dette kompendiet er det spesielt små mikroprosessorer som er i fokus og spesielt den typen som kalles mikrokontrollere. En **mikrokontroller** inneholder en mikroprosessor i tillegg til de fleste andre funksjoner som en nødvendige i et lite datasystem. Enkelt sagt er en mikrokontroller en liten, (nesten) komplett datamaskin med:

- CPU
- Hukommelse
- IO-enheter
- Interne og eksterne databusser

Avanserte IO-enheter som harddisker og CD-lesere benyttes sjelden i forbindelse med mikrokontrollere, men IO-enheter for kommunisere med annet datautstyr blir mer og mer vanlig.

De første mikrokontrollerne kom på markedet ikke lenge etter mikroprosessorene, men var på langt nær så komplette som dagens mikrokontrollere. De første mikrokontrollerne var ikke på langt nær så enkle å ta i bruk av en elektronikkonstruktør eller en programmerer heller.

Blant dagens viktigste produsenter av mikrokontrollere finner man: Intel, Motorola, Microchips, Atmel.

1.2 Embedded systemer

Mikrokontrollere benyttes i såkalte "**embedded**"[2] systemer. Dette er utstyr der datamaskiner benyttes som et middel til å forbedre utstyret. Det kan dreie seg om å gi utstyret flere funksjoner, forbedre brukervennligheten, øke påliteligheten, forbedre ytelsen eller andre forbedringer. Et embedded system framstår ikke som en datamaskin for brukeren. Eksempler på embedded systemer er:

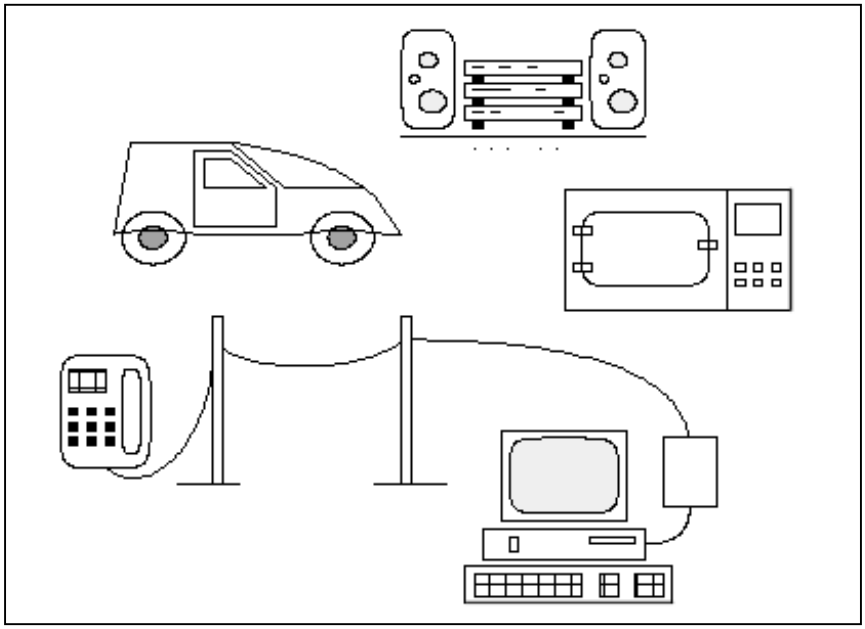
- DV-kamera
- Mikrobølgeovn

---

1 IC → Integrert krets / datachip / brikke

2 Av mangel på et god norsk begrep benyttes det engelske "embedded system". Mulige norske oversettelser vil være innbygget system eller innbyggingssystem – altså et datasystem som inngår i et annet primærsystem.

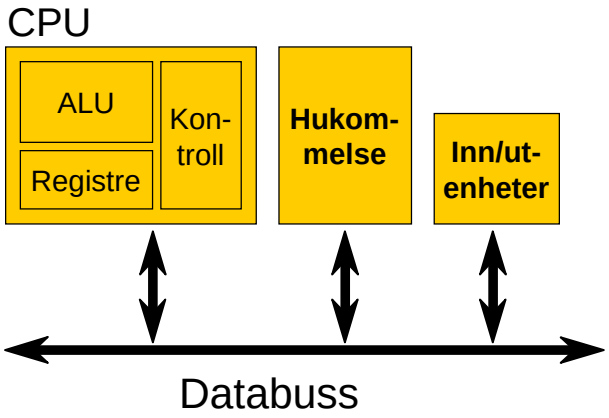
- Bil (flere datamaskiner)
- Stereoanlegg
- Mobiltelefon
- Moderne dukke
- ... og mye mer



1.3 Hovedstrukturen for datamaskiner

Selv om hovedstrukturen i en datamaskin og altså en mikrokontroller sikker er kjent for leseren, skal denne gjentas her for oversiktens skyld. Tradisjonelt representerer man datamaskinen med en modell som består av de tre hovedelementene:

- CPU (CPU)
- IO-enheter
- Hukommelsen



Datamaskiner som er bygget opp rundt disse tre hovedelementene kalles **von Neumann** maskiner etter den ungarsk-tysk-amerikanske matematiker og atomfysiker John von Neumann.

Informasjon må utveksles mellom enhetene som inngår i mikrokontrolleren. Dette kan skje via separate datakanaler mellom CPU-en og hver av de resterende enhetene i systemet. Dette hadde blitt både komplisert plasskrevende og dyrt fordi det som regel finnes flere hukommelsesenheter og mange IO-enheter. En enklere og mer systematisk tilnærming til problemet, er å benytte en **databuss** bestående av et sett av elektriske ledere som snor seg innom alle enhetene i maskinen. CPU-en kan utveksle data med de forskjellige enhetene som etter tur kan benytte databussen.

Data som sendes ut på databussen kan egentlig ses av alle komponentene som er tilkoblet bussen. Imidlertid benytter CPU-en en spesiell teknikk som kalles **adressering** for å sikre at data bare sendes til den ønskede enhet. Bussene i datasystemene fører derfor både adresse- og datasignaler.

1.3.1 CPU / MCU

CPU-en[1] er selve hjernen i mikrokontrolleren. Denne enheten utfører alle beregningene som mikrokontrolleren kan utføre. Det er også CPU-en som bestemmer hvordan og i hvilken rekkefølge programmene skal utføres, og som tar alle kritiske avgjørelser som er nødvendig for at mikrokontrolleren skal kunne fungere. Vi sier den utfører *aritmetiske* og *logiske* operasjoner.

I en mikrokontroller er det vanlig å benytte forkortelsen **MCU**[2] (Microcontroller CPU) som navn på CPU. MCU er integrert på samme brikken (IC) som internminne (program og data) og IO-funksjoner.

I dette kompendiet skal begrepet MCU benyttes videre i teksten.

1.3.2 Hukommelsen

Hukommelsen er absolutt nødvendig for å få mikrokontrolleren til å fungere. Alle **programmer** som skal utføres og alle **data** som behandles, må lagres i hukommelsen.

De hukommelsestypene som er vanlige i mikrokontrollere er:

- FLASH PROM[3] (for permanent programlager)
- UV-PROM[4] (for permanent programlager)
- SRAM[5] (for generell datalagring)
- EEPROM[6] (for permanent lagring av viktige systemparametre)

---

1 CPU → Central Processing Unit

2 MCU → Microcontroller CPU

3 FLASH-teknologien lar oss programmere en hukommelsesblokk permanent. Det er også mulig å mulig å nulle ut området igjen vha en elektrisk puls og å reprogrammere det.

4 UV → ultra violet. Dette er en hukommelsestype som kan programmeres igjen og igjen. Ulempen i forhold til FLASH er at vi må benytte UV-lys for å slette hukommelsen før reprogrammering. Dette kan ta 15 – 20 minuttter. I tillegg er det en ulempe at IC-en må utstyres med et kvartsvindu over brikken for at lyset skal kunne slippe inn. Dette fordyrer komponenten.

5 Statisk RAM. Dette betyr at vi kan lese og skrive til hukommelsen og at innholdet beholdes så lenge spenningsforsyningen til komponenten er på.

6 Elactrically Erasable PROM. Vi kan slette innholdet elektrisk, men i motsetning til FLASH slettes en og en celle. Dette er en omstendelig process som tar lang tid, men

1.3.3 IO-enheter

I teorien kan en mikrokontroller virke uten sine IO-enheter[1]. Uten IO-enhetene vil MCU-en fint kunne utføre programmer som allerede ligger i hukommelsen og hente data fra og skrive data til hukommelsen. Men ingen ting ville noen sinne kunne påvirke programmene. Likeledes ville heller ingen resultater kunne vises fra beregningene. Det er dette IO-enhetene dreier seg om – å mate informasjon til maskinen og å kunne presentere informasjon fra beregningene i maskinen. Informasjonen som utveksles med maskinen, kan utveksles med brukere direkte, med teknisk måle- og kontrollutstyr eller med andre maskiner.

IO-enheter består av et stort utvalg enheter som tjener til å mate data inn og ut av kontrolleren.

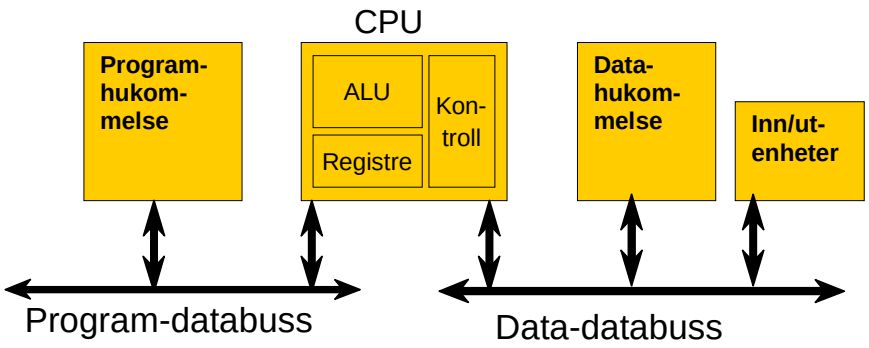
Typiske IO-enheter for mikrokontrollere er:

- Parallellporter for digital IO
- Serielle porter for digital IO
- Analog-digital omformere
- Digital-analog omformere
- Klokker og timere
- Nettverksenheter

1.4 Harvard arkitektur

Mikrokontrolleren som er skissert ovenfor har et felles buss-system for både program og data. Dette kalles *Princeton arkitektur* etter universitetet i USA hvor dette prinsippet først ble tatt i bruk.

I en del sammenhenger ønsker man at MCU-en skal lese program og data samtidig for å øke beregningshastigheten. Da må man benytte et system med to separate busser og to separate hukommelsesmoduler som vist nedenfor. Dette kalles *Harvard arkitektur*.



som til gjengjeld gir oss anledning til å oppdatere innholdet av bare ett ord uten å måtte slette hele PROM-blokken.

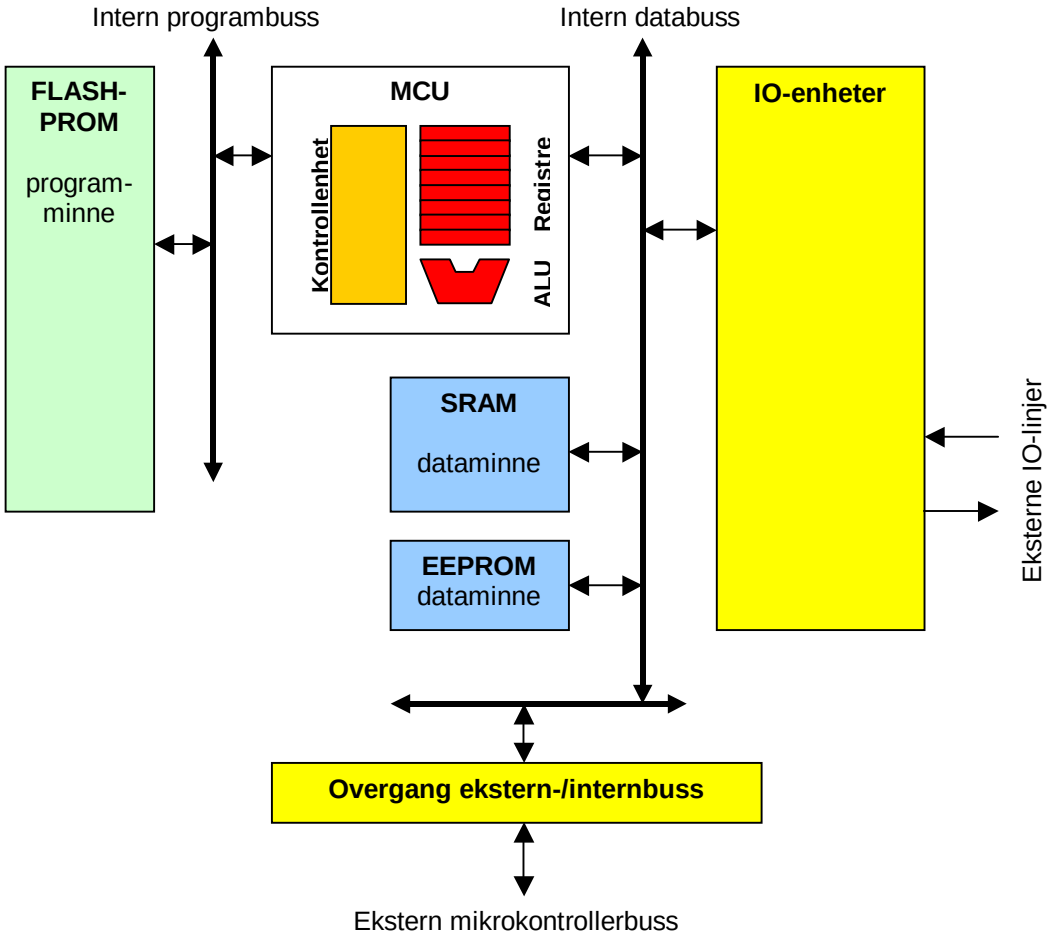
[1] IO står for Input/Output eller Inn/Ut

1.5 Atmel AVR, en første presentasjon

Den komponenten som skal beskrives nærmere i dette kompendiet, er et mikrokontroller fra Atmel kalt **AT90S2313**. Dette er et medlem i en 8-bits mikrokontrollerfamilie, AVR, som benytter **Harvard-arkitektur** og også mange ideer fra **RISC**[1] datamaskiner. I praksis betyr dette at de aller fleste maskininstruksjoner utføres på en enkelt klokkecycle. Motstykket til RISC er CISC-maskiner[2] hvor blant annet Intel Pentium-brikkene hører hjemme.

Vi skal ikke gå mye nærmere inn på RISC/CISC diskusjoner her, men merke oss at RISC-arkitekturen også medfører at alle maskininstruksjoner holdes like lange av effektivitetsgrunner. I AVR-kontrollerne er ordlengden for instruksjoner **16 bit**. De resterende ordlengder i kontrolleren er for det meste **8 bit**.

AVR mikrokontrollerarkitektur



AVR-kontrollerne benytter **FLASH** (16 bit) programminne og **SRAM** (8 bit) dataminne. For permanent lager av parametere og andre størrelser som skal huskes permanent, benyttes **EEPROM** (8 bit). Antall ord som inngår i de forskjellige hukommelsestypene vil variere mellom de ulike typer AVR-kontrollere, men vår 2313-kontroller har **1024 ord programminne**, **128 bytes** dataminne og **128 bytes** EEPROM. Dette er akkurat nok til å kjøre små anvendelser fortrinnsvis programmert i assemblykode.

---

[1] RISC → Reduced Instruction Set Computer.

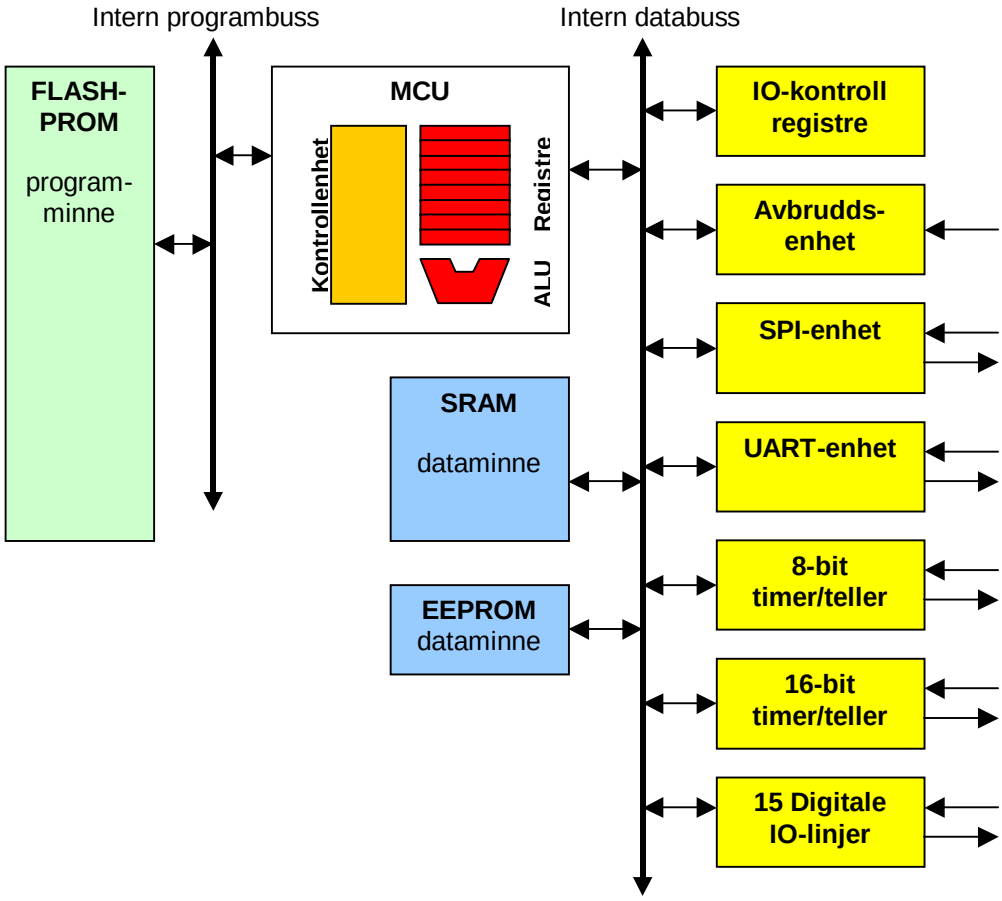
[2] CISC → Complex Instruction Set Computer.

I tillegg til MCU og hukommelse har AVR-kontrollerne et rikt utvalg av integrerte IO-funksjoner.

Som det framgår av figuren ovenfor, har en mikrokontroller interne buss-systemer. For at det skal være mulig å utvide kapasiteten til mikrokontrolleren, er det også mulig å forlenge de interne bussene eksternt. På denne måten blir det f. eks. mulig å koble til ekstra SRAM og ekstra IO-funksjoner. AVR-kontrollerne har ikke muligheter for å utvide programminnet med ekstern hukommelse som det framgår av figuren.

Antall IO-funksjoner vil også variere, men figuren nedenfor viser funksjoner de fleste AVR-kontrollere har.

AVR mikrokontrollere med IO-funksjoner

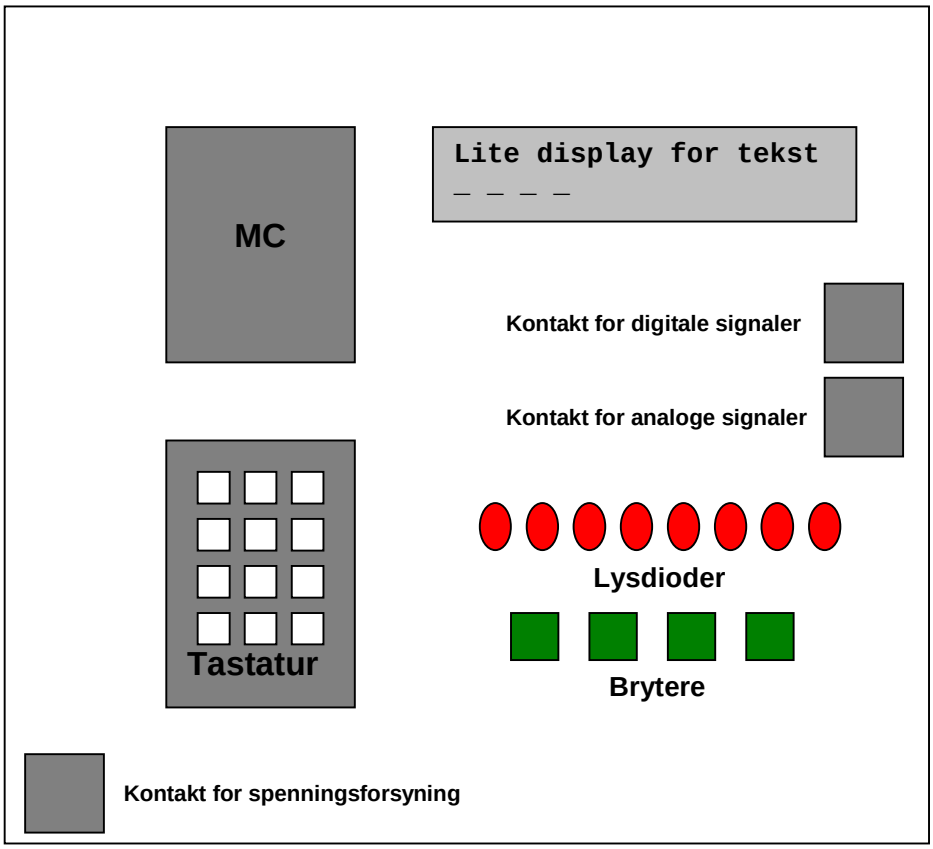


Man får AVR-kontrollere med så få som 8 bein og med så mange som over 100. Vår komponent (2313) har 28 bein i en DIL pakke og er således praktisk å jobbe med i laboratoriet.

1.6 Et enkelt mikrokontrollersystem

Mikrokontroller en mer eller mindre komplett (om enn liten) datamaskin. Likevel vil mikrokontrolleren kreve noen ekstra komponenter for å kunne virke som datamaskin i et gitt utstyr. Tilleggsenheter som ofte er med i en ferdig konstruksjon innbefatter eksempelvis:

- Likespenningsforsyning for å skaffe energi til mikrokontrolleren. Det benyttes ofte en vekselstrømsadapter for tilkobling til 220V nettet eller et batteri. Energiforsyning er absolutt nødvendig.
- Diverse lysindikatorer (LEDs) for å vise brukeren status i utstyret.
- Enkle brytere for at brukeren skal kunne kommunisere med utstyret.
- Et lite tastatur.
- Et lite display (tekstbasert eller grafisk) for å gi meldiger og annen informasjon til brukeren.
- Diverse tilkoblingspunkter for analoge eller digitale følere. Disse kommer som regel fra hovedprosessen i utstyret representerer måle- og statusverdier i utstyret.
- Diverse tilkoblingspunkter for digitale og analoge kontrollorganer som styrer funksjonaliteten i utstyret.



Selve mikrokontrollersystemet bygges som regel opp på et/eller flere elektriske **trykte kretskort** med banene i flere lag (flerlagskort). Kortene kalles trykte

fordi elektriske kobberbaner er preget inn på kortene som om de skulle være trykket. Dette er imidlertid ikke riktig. Kobberbanene er etset ut. Komponentene monteres og forbindes elektrisk vha. en automatisk loddeprosess.

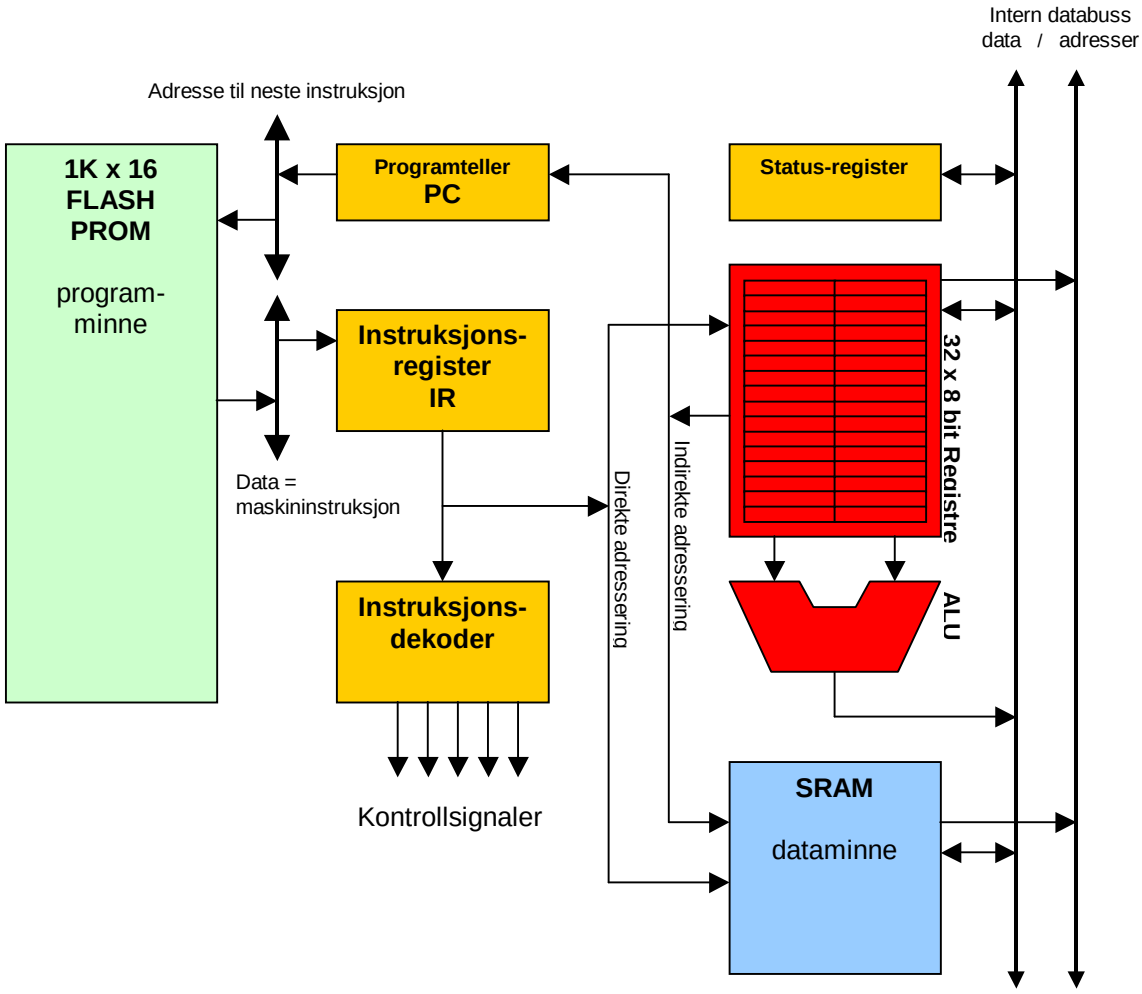
Profesjonelle kort må framstilles med profesjonelt og dyrt utstyr. For hjemme- og labbruk kan man få til en god del med nøyaktighet og tilmodighet med relativt små krav til framstillingsutstyret.

2 AVR – arkitekturbeskrivelse

Arkitekturen til en datamaskiner en beskrivelse av hvordan datamaskinen virker intern sett fra perspektivet til et maskinkodeprogram.
Vi skal altså beskrive hvordan mikrokontrolleren virker internt. Beskrivelsen vil være på det logiske planet. Det betyr at for detaljer rundt beinplassering, strømforbruk, signalnivåer, henvises det til databladet for AT90S2313 som kan hentes fra http://www.atmel.com/

Vi skal heller ikke behandle de fleste IO-enhetene, men begrense oss til de digitale inn-/ut-portene. Vi fokuserer altså på MCU-en og tilknytningen til intern hukommelse i første omgang.

AVR MCU og internhukommelse



NB! Vi skal studere hvordan mikrokontrolleren virker når den er ferdig programmert, dvs. når den allerede har et program installert i programminnet. I denne beskrivelsen skal vi se bort fra problemet med hvordan programmet installeres og de funksjonsenhetene og signalveiene som er nødvendige for å få dette til.

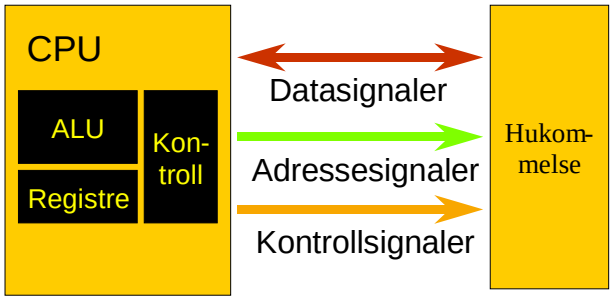
2.1 Interne busser

Intern i mikrokontrolleren er det 2 hovedbussystemer, et for transport av programinstruksjoner og et for transport av data. Disse deles igjen opp i 3 deler, slik at vi totalt sett har 6 delbusser:

Databuss
Dette er bussen som fører data mellom registre og dataminnet og mellom registre og IO-enheter

Adressebuss
Adressene som peker ut data i dataminnet og i registerbanken eller gitte funksjonsregistre i IO-enhetene føres på denne bussen.

Kontrollbuss-data
Fører kontrollsignaler for lesing av skriving av data over databussen. Denne bussen teges ikke inn på noen av skissene i kompendiet.



Programadressebuss
Adressene til instruksjonene som leses fra programminnet føres her.

Instruksjonsbuss
Instruksjonene som leses fra programbussen føres over denne bussen. Det er bare instruksjonsregisteret som leser fra bussen.

Kontrollbuss-instruksjon
Kontrollsignaler for det som føres over instruksjonsbussen.

3 MCU-enheter

MCU-en deles opp i følgende enheter:

Arbeidsregistre
32-stk. 8 bits registre for mellomlagring av data som behandles av mikrokontrollerprogrammet. Arbeidsregistrene er adresserbare fra 00h til 1Fh. Verdier kan leses inn og ut av registrene via databussen.
Noen av arbeidsregistrene kan inneholde adresser (**indirekte adressering**) for å peke ut data i dataminnet eller en maskininstruksjon i programminnet.

ALU for utførelse av de aritmetiske og logiske beregningene i for hver enkelt maskininstruksjon. ALU-en henter operander fra arbeidsregistrene og

leverer resultatet til databussen. Statusinformasjon fra ALU-operasjonene lagres i statusregisteret.

Kontrollenhet

Kontrollenheten er den enheten som får MCU-en til å gjøre jobben. Kontrollenheten er ansvarlig for å hente inn og tolke instruksjoner fra hukommelsen og dermed sørge for at ALU-en hele tiden forsynes med nye regneoppgaver. Inn- og utlesning av data fra/til hukommelsen styres også av kontrollenheten.

Normal arbeidssekvens for kontrollenheten er å hente instruksjonene fra hukommelsen fortløpende. En viktig type instruksjoner, **hopp-instruksjoner**, gjør MCU-en i stand til å avvike fra denne sekvensielle innhenting og hoppe til et annet sted i programmet hvor nye instruksjoner skal hentes fra.

3.1.1 Funksjonsenheter i kontrollenheten

Følgende funksjonsenheter identifiseres i kontrollenheten for en AVR-kontroller:

Statusregister

Et register som inneholder statusinformasjon bl. a. fra ALU. Det kan f. eks. dreie seg om at resultatet av en subtraksjon er negativt, eller at en addisjon har endt med overflow. Statusregisteret er ikke adresserbart på databussen som arbeidsregistrene.

Programteller

Dette er et register som holder styr på hvor i programminnet neste instruksjon befinner seg. Neste instruksjon hentes altså fra den adressen som angis av innholdet i programtelleren. Programtelleren inkrementeres normalt med verdien 1 for hver instruksjon som leses og utføres. Programtelleren styrer innholdet av programadressebussen.

Instruksjonsregister

Instruksjonene som leses ut av programminnet over instruksjonsbussen, lagres her mens de tolkes og utføres av MCU-en. Noen instruksjoner inneholder direkte referanser til dataminnet eller registre (**direkte adressering**). Disse adressene hentes fra instruksjonsregisteret.

Instruksjonsdekoder

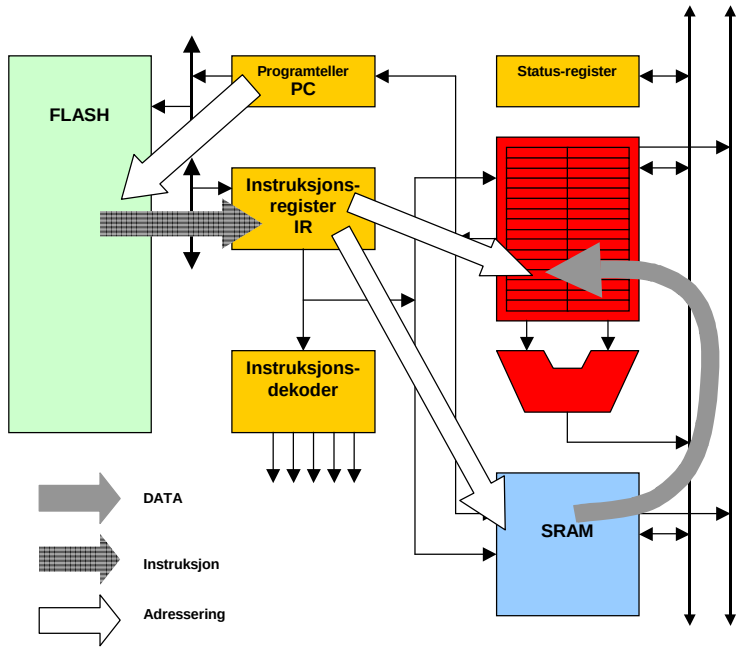
Denne enheten tolker hver enkelt maskininstruksjon og setter ut en sekvens av kontrollsignaler som styrer utførelsen av instruksjonen i ALU og sørger for at data flytte fra register til register eller mellom registre og dataminnet. Instruksjonsdekoderen styrer kontrollbussene nevnt ovenfor.

3.2 Instruksjonsutførelse

Vi setter opp en forenklet punktvis beskrivelse av hva som skjer når mikrokontrolleren utfører en instruksjon. Vi antar instruksjonen har som oppgave å hente en byte fra en adresse 70h i SRAM og kopiere denne verdien til register nummer 5.

1. Programtelleren (PC) inneholder adressen til instruksjonen.
2. Adresses sendes ut på programadressebussen.
3. Instruksjonen hentes ut fra riktig adresse i programminnet til kopieres via instruksjonsbuss til instruksjonsregisteret (IR).
4. Programtelleren inkrementeres med 1.
5. Instruksjonen sendes videre til instruksjonsdekoderen.

6. Instruksjonsdekoderen finner ut at dette er en instruksjon for å hente data fra dataminnet til et register.
7. Adressen til registeret plukkes ut fra IR-register.
8. Adressen til dataminnet plukkes ut fra IR-register.
9. Instruksjonsdekoderen sender ut kontrollsignaler som sørger for datakopiering.
10. Riktig databyte kopieres fra dataminnet til databuss.
11. Innhold av databuss kopieres til adressert register.
12. Prosessen starter på nytt fra punkt 1 med å hente en ny instruksjon.



3.3 Internhukommelse

For å beskrive hvordan hukommelsen i en datamaskin er lagt opp med hensyn til ordlengder, adressering etc., benyttes et såkalt **hukommelseskart** eller "**memory map**" som det også kalles.

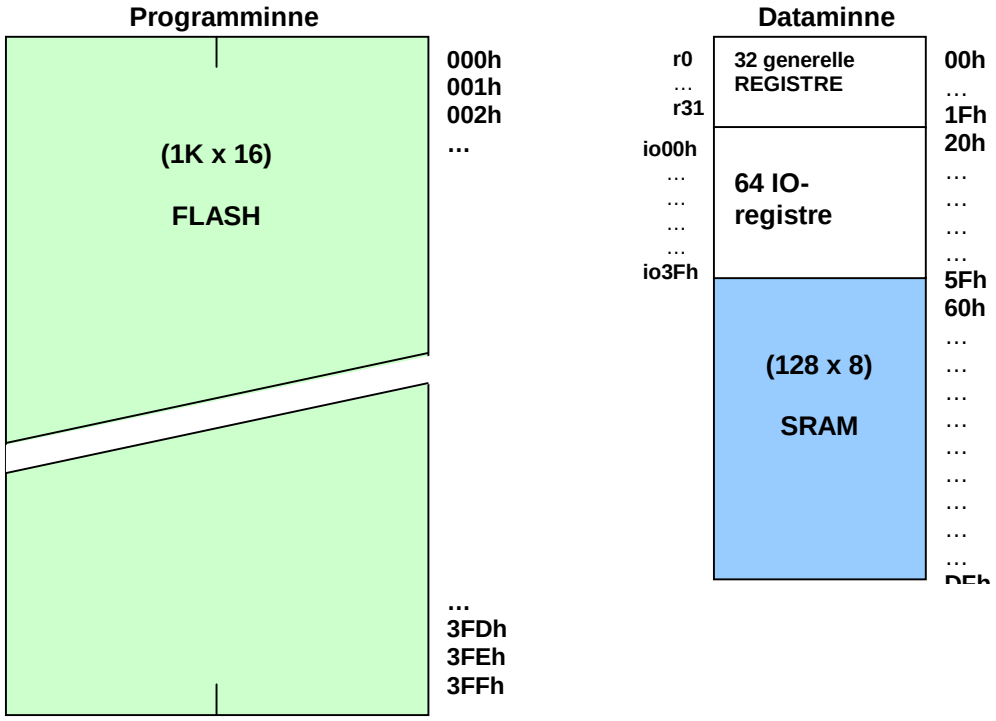
Man tenker seg en modell der minnet er en stor tabell av fortløpende indekserte ord. Ordene tilsvarer minnelokasjonene og kan ha varierende ordlegde fra hukommelse til hukommelse. Hos AVR-kontrollerne har dataminnet 8-bits ordlengde, mens programminnet har 16-bits ordlengde.

Indeksene i tabellen er adressene til hukommelsen. Det er tradisjon for at adresser oppgis på heksadesimal form. Dette angis ofte ved at det settes en liten 'h' bak adressen – eksempel 2Bh, 18h. Et annet alternativ for angivelse av heksadesimale adresser er å benytte prefikset 0x – eksempel 0x2B, 0x18.

Arbeidsregistrene til AVR nummereres som regel desimalt – eksempel r20.

Fra figuren nedenfor ser vi at programminnet adresseres fra 000h til 3FFh – noe som tilsvarer 10254 ord. Dataminnet adresseres fra 00h til DFh, som tilsvarer et adresseringsrom på 224 adresser. Av disse 224 benyttes 32 til arbeidsregistrene, 64 til funksjonsregistre og 128 til SRAM dataminne.

Hukommelseskart for program og dataminne AVR 2313



Ordlengde = 16 bit
Antall ord = 1024
Kapasitet = 2048 Bytes

Ordlengde = 8 bit
Antall ord = 32 + 64 + 128
Kapasitet = 224 Bytes

3.3.1 Generelle arbeisregistre / registerbank

Figuren nedenfor viser strukturen til de 32 generelle arbeidsregistrene i MCU. De fleste maskininstruksjoner i instruksjonssettet har direkte tilgang til alle disse registrene.

Unntakene er de 5 aritmetiske og logiske instruksjonene mellom en 8-bits konstant og innholdet av et register:

`SBCI, SUBI, CPI, ANDI og ORI`

og instruksjonen for å laste inn en 8-bits konstant i et register:

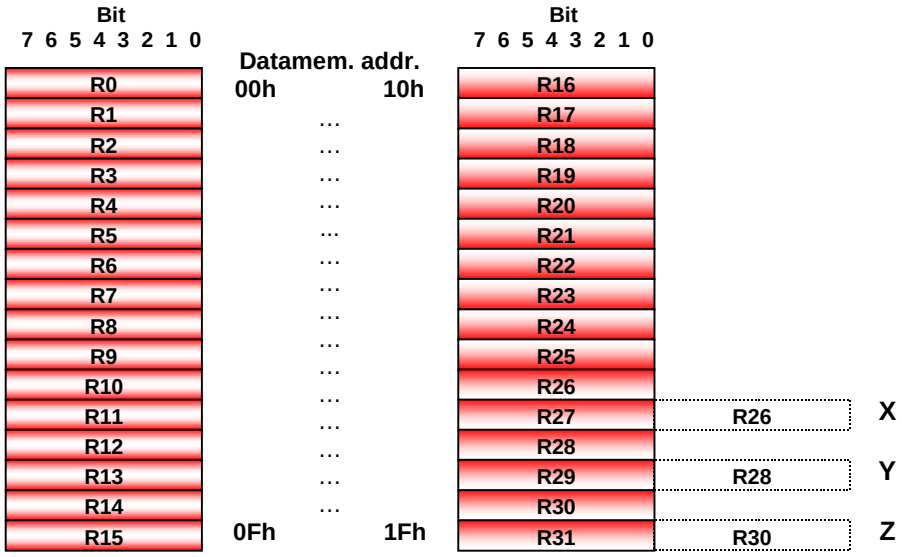
`LDI`

I disse instruksjonene har man bare tilgang til de 16 høyest adresserte registrene: R16 – R31.

Grunnen til disse unntakene finner man ved å studere formatet for instruksjonene. Det er nemlig ikke nok plass i instruksjonsordene for de 5 bitene som må til for å peke ut alle de 32. Det er bare plass til 4 bits registeradresser.

Hvert register har også en unik adresse i det felles adresserommet som gjelder for brukerdata – 00h – 1Fh. Det betyr at adresseregistrene (X, Y, Z) som er beskrevet nedenfor, også kan benyttes til å referere til registre i registerbanken.

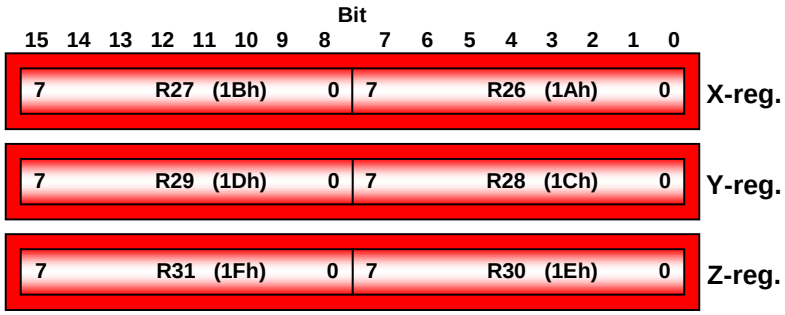
32 generelle arbeidsregistre for AVR mikrokontrollere



3.3.2 Adresseregistre

I tillegg til å være generelle lagringsplasser for data, har registrene R26 – R31 også en tilleggsfunksjon. Disse registrene kan 2 og 2 sammen også benyttes til å lagre adresser. De dobbelte registrene kalles hhv. X, Y og Z. Hvordan de er satt sammen 2 og 2 og hva som blir minst signifikante og mest signifikante bit og byte går fram av figuren nedenfor.

X, Y og Z-registrene



X, Y og Z kalles **adresseregistre** eller **pekerregistre**. De benyttes i forbindelse med indirekte adressering til brukerdata på databussen (ekstern og internt).

Det er spesielle instruksjoner i instruksjonssettet som benytter adresseregistrene. Disse instruksjonene gir også muligheter for å automatisk inkrementere eller dekrementere innholdet i registrene etter en dataaksess.

For å kunne forandre innholdet i et adresseregister, må man normalt sett legge inn nye verdier i mest signifikante byte og minst signifikante byte hver for seg.

3.3.3 FLASH programminne

AVR 2313 inneholder 2048 bytes med FLASH programminne. Disse er organisert i 1024 ord á 16 bit adresserbare i området 000h – 3FFh. Disse adressene er unike for programminnet og kolliderer ikke med adressene i dataområdet (registre, SRAM). Vi sier at program og data befinner seg i hvert sitt adresserom (address space).

De fleste maskininstruksjoner opptar 1 ord i programminne. Noen instruksjoner opptar 2 ord. Dette gjelder i første rekke de instruksjonene som angir en direkte adresse i dataområdet.

Atmel garanterer at FLASH-minnet skal kunne reprogrammeres 1000 ganger. I praksis klarer de mer.

FLASH minnet kan ikke modifiseres byte for byte men blokk for blokk. Det mest vanlige er å slette og programmere hele FLASH-området om gangen.

3.3.4 EEPROM dataminne

AVR 2313 inneholder 128 bytes med EEPROM dataminne. Dette minnet er organisert i et helt eget adresserom (atskilt fra programminnet og dataminnet for øvrig). Det er mulig å lese og skrive enkeltbyter fra dette dataområdet vha av 3 IO-registre:

EEPROM Adresseregister – EEAR

EEPROM Dataregister – EEDR

EEPROM Kontrollregister – EECR

Atmel garanterer at bytene i EEPROM skal kunne reprogrammeres minst 100000 ganger.

Det tar ganske lagt tid å skive en byte til EEPROM. Derfor benyttes dette området mest for statiske data som oppslagtabeller etc. og viktige systemparametere som ikke skal forandres ofte.

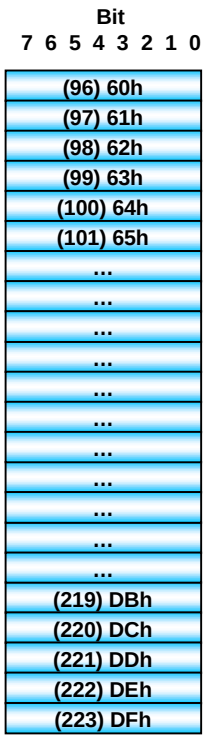
3.3.5 SRAM dataminne

Figuren nedenfor viser hvordan AVR 2313's SRAM dataminne er organisert. Dette er samme typen hukommelse som f. eks. en standard PC-arbeidsstasjon bruker i CHACHE-minnet. Altså en rask hukommelsestype.

AVR 2313 har et svært lite dataminne – bare 128 bytes. Det er derfor ikke mye brukerdata som kan lagres her. I praksis kan derfor AVR 2313 kun benyttes i ganske små systemer der kravet til datalagring er begrenset.

Imidlertid er det mulig å koble til et eksternt dataminne slik at man kan lagre datamengder opp mot 64 kBytes. Dette medfører igjen ekstra komponenter i systemet med krav til ekstra plass og ekstra kostnader. Der er derfor ofte et bedre alternativ å benytte en AVR-kontroller av en annen type med større SRAM.

128 bytes SRAM for AVR 2313



3.3.6 IO-registre

AVR 2313 har 64 8-bits IO-registre. Disse registrene benyttes for å styre og kontrollere de fleste IO-funksjonsenhetene i mikrokontrolleren. For å utføre en IO-operasjon skriver man typisk en kommando til IO-registeret som styrer vedkommende funksjon. Man kan også sjekke status for IO-funksjonene ved å lese av status bit i angitte IO-registre.

IO-registrene har adressene 20h til 5Fh i brukerdataområdet. Her kan alle maskininstruksjonene som leser og skriver i SRAM benyttes til dataaksesser.

IO-registrene kan også nås via spesielle IO-instruksjoner i et eget IO-adresserom: 00h – 3Fh.

De mest benyttede IO-registrene i dette kurset vil være de 2 parallelle digitale IO-portene PORTB og PORTD samt registrene for kontroll av disse: DDRB og DDRD.

4 AVR instruksjoner

Dette kapittelet handler mest om formatet på maskin- og assemblyinstruksjoner og prinsippene for å kunne overføre dataverdier mellom dataminnet og registre.

4.1 Maskininstruksjoner

De instruksjonene som ligger i programminnet er **maskininstruksjoner**. Dette er binærmønstre som MCU-en kan tolke og utføre. Siden MCU-en bare kan forholde seg til maskininstruksjoner, må alla andre former for instruksjoner vi måtte ønske å få mikrokontrolleren til å utføre, først oversettes til maskininstruksjoner.

4.1.1 Instruksjonssett

Summen av alle maskininstruksjoner som kan forstås av en datamaskin, kalles **instruksjonssettet** for vedkommende maskin. Instruksjonssetet kan bestå av alt fra ca. 100 instruksjoner til opp mot 500. AVR 2313 har drøyt 100 mskininstruksjoner og har derfor et relativt lite instruksjonssett.

4.1.2 Maskinkode

Sekvenser av maskininstruksjoner som får mikrokontrolleren til å utføre en bestemt oppgave, kalles et **maskininstruksjonsprogram** eller **maskinkode**.

4.1.3 Format

De fleste maskininstruksjonene er på 16 bit. Vi går derfor ut fra 16 bit som en normal instruksjonslengde.

Hvor mange bit som går med til operasjonskode, til registerreferanser og til dataminnereferanser skisseres ofte ved hjelp av et rektangel som vist nedenfor.

For hver figur er det markert hvor mange bit som hvert felt opptar. I eksempelet ovenfor opptar operasjonskoden 6 bit og to registerreferanser 5 bit hver. Alle

bitposisjonene er ikke nødvendigvis 100% riktig plassert på figurene, men antall bit innefor hvert felt skal være riktig – det er det viktigste for oss nå.

4.1.4 Feltbetegnelser

De enkelte feltene i instruksjonsformatet er betegnet slik:

Op-kode
: Operasjonskode. Dette er feltet som angir hva instruksjonen skal utføre, f. eks. addisjon, subtraksjon, kopiering av data også videre.

Rd Destinasjonsregister i registerbanken

Rr Sourceregister i registerbanken

K Datakonstant.

k Adressekonstant

A IO-adresse

X X-adresseregister

Y Y-adresseregister

Z Z-adresseregister

b bitnummer (0 - 7) i register

Sourceregisteret er det registeret som dataverdien hentes fra. Med andre ord inneholder Rd instruksjonens (ene) operand.

Destinasjonsregisteret angir det registeret et resultat skal skrives til. Det er ikke mulig å angi mer enn 2 registre per instruksjon. I de tilfellene der man trenger 2 registre til å holde operander, må den ene operanden ligge i samme register som resultatet skal legges i. Rd angir i de tilfellene både source og desitinasjon.

4.2 Assemblyinstruksjoner

Ingen programmerer direkte i maskinkode. Det å forholde seg til bitkombinasjonene som danner maskininstruksjonene er svært arbeidskvevende og lite effektivt. Hver op-kode i maskininstruksjonssettet gis derfor et symbolsk navn som er en forkortelse (engelsk) som beskriver hva instruksjonen går ut på.

Disse forkortelsene kalles **mnemonics** (huskesymboler) og er utformet slik at når vi ser instruksjoner som er skrevet med mnemonics, er det ganske lett[1] å huske hva instruksjonene står for. **Maskininstruksjoner** skrevet på **symbolsk** form kalles **assemblyinstruksjoner**. En mnemonic er instruksjonens **navn**. Eksempler på typiske navn er ADD og AND som står for henoldsvis addisjonsinstruksjonen og den logiske AND-indtruksjonen.

Operandadressene får også symbolske navn som r0, r2, r3 … r29, r30, r31, X, Y, Z når det gjelder registernavn. Direkte adresser til dataminnet kan også gis symbolske adresser, men foreløpig skal vi benytte numeriske adresser på hex-form og desimalform her. Fortrinnsvis skal vi benytte hex-fom for adresser. Disse kodes med prefixet 0x etterfulgt av selve adressen – eksempel: 0x123, 0xAB, 0x1F. Vi benytter aldri binærkode for å angi numeriske verdier i assemblyinstruksjonene for AVR mikrokontrollere.

Et eksempel på en komplett assemblyinstruksjon er:

[1] I alle fall er det mye lettere enn maskininstruksjonene direkte; selv om også mnemonics kan være ganske vanskelige å forså betydningen av ved å lese dem.

```
ADD r1, r2
```

Betydningen av denne instruksjonen er: Adder innholdet av registrene r1 og r2. Resultatet av addisjonen skal lagres i r1 (og dermed skrives over det gamle innholdet der).

For å forklare en instruksjon kan man legge inn kommentarer etter et semikolon:

```
ADD r1, r2;   Legg sammen r1+r2. Resultat overskrives r1.
```

Alt som kommer etter semikolonet hører altså ikke til selve instruksjonen.

I dette avsnittet skal noen av de mest benyttede instruksjonene presenteres på tabellform. Den enkelte instruksjonene skal forklares etter hvert.

Tabellene er ikke komplette med hensyn til tilgjengelige instruksjoner, men det er foretatt et utvalg som vil være aktuelle i kurset som dette kompendiet støtter.

4.2.1.1 ASSEMBLYINSTRUKSJONER OG IO-REGISTRE.

Alle IO-registre gis symbolske navn i assemblyinstruksjonene. Disse navnene er oppgitt i databladet for de enkelte komponentene. Disse navnene blir også benyttet i dette kompendiet.

4.2.2 Dataoverføringsinstruksjoner (data transfer)

MNEMONIC	Operander	Beskrivelse	Funksjon	Flagg
IN	**Rd, A**	Les data fra IO-register til et register	Rd ← A	
LD	**Rd, X/Y/Z**	Les data fra dataminne med indirekte adressering til register	Rd ← M[X] / Rd ← M[Y] / Rd ← M[Z]	
LD	**Rd, X+/Y+/Z+**	Les data indirekte med postinkrement til register	Rd ← M[X] , X ← X+1 / Rd ← M[Y] , Y ← Y+1 / Rd ← M[Z] , Z ← Z+1	
LD	**Rd, -X/-Y/-Z**	Les data indirekte med predekrement til register	X ← X-1 , Rd ← M[X] / Y ← Y-1 , Rd ← M[Y] / Z ← Z-1 , Rd ← M[Z]	
LDD	**Rd, Y/Z + q**	Les data indirekte med forskyvning til register	Rd ← M[Y + q] / Rd ← M[Z + q]	
LDI	**Rd, K**	Legg en konstant i et register	Rd ← K	
LDS	**Rd, k**	Les data fra dataminne med direkte adressesering til register	Rd ← M[k]	
MOV	**Rd, Rr**	Kopier verdien av et register til et annet register	Rd ← Rr	
OUT	**A, Rr**	Skriv data fra register til et IO-register	A ← Rr	
POP	**Rd**	Les data fra stack til register	Rd ← STACK	
PUSH	**Rr**	Skriv data fra register til stack	STACK ← Rr	
ST	**X/Y/Z, Rr**	Skriv data fra register til dataminne med indirekte adressering	M[X] ← Rr / M[Y] ← Rr / M[Z] ← Rr	
ST	**X+/Y+/Z+, Rr**	Skriv data fra register til dataminne med indirekte adressering og postikrement	M[X] ← Rr , X ← X+1 / M[Y] ← Rr , Y ← Y+1 / M[Z] ← Rr , Z ← Z+1	
ST	**-X/-Y/-Z, Rr**	Skriv data fra register til dataminne med indirekte adressering og predekrement	X ← X-1 , M[X] ← Rr / Y ← Y-1 , M[Y] ← Rr / Z ← Z-1 , M[Z] ← Rr	
STD	**Y/Z + q, Rr**	Skriv data fra register til dataminne med indirekte adressering og forskyvning	M[Y + q] ← Rr / M[Z + q] ← Rr	
STS	**k, Rr**	Skriv register til dataminne med direkte adressering	M[k] ← Rr	

4.2.3 Aritmetiske og logiske instruksjoner

MNEMONIC	Operander	Beskrivelse	Funksjon	Flagg
ADC	**Rd, Rr**	Addisjon med inngående mente	Rd ← Rd + Rr + C	Z N V C H
ADD	**Rd, Rr**	Addisjon	Rd ← Rd + Rr	Z N V C H
AND	**Rd, Rr**	Logisk AND operasjon	Rd ← Rd AND Rr	Z N V
ANDI	**Rd, K**	Logisk AND med konstant	Rd ← Rd AND K	Z N V
CBR	**Rd, K**	Nullstill bit i register	Rd ← Rd AND (FFh - K)	Z N V
CLR	**Rd**	Nullstill register	Rd ← Rd XOR Rd	Z N V
COM	**Rd**	1's komplement	Rd ← FFh – Rd	Z C N V
CP	**Rd, Rr**	Compare (sammenlign)	Rd – Rr	Z N V C H
CPI	**Rd, K**	Compare Immediate	Rd – K	Z N V C H
DEC	**Rd**	Dekrementer register	Rd ← Rd – 1	Z N V
EOR	**Rd, Rr**	Eksklusiv OR	Rd ← Rd XOR Rr	Z N V
INC	**Rd**	Inkrementer register	Rd ← Rd + 1	Z N V
NEG	**Rd**	2' komplement	Rd ← 00h – Rd	Z N V C H
OR	**Rd, Rr**	Logisk OR operasjon	Rd ← Rd OR Rr	Z N V
ORI	**Rd, K**	Logisk OR med konstant	Rd ← Rd OR K	Z N V
SBC	**Rd, Rr**	Subtraher med inngående lånemente	Rd ← Rd – Rr – C	Z N V C H
SBR	**Rd, K**	Sett bit i register	Rd ← Rd OR K	Z N V
SER	**Rd**	Sett alle bit i register til 1	Rd ← FFh	
SUB	**Rd, Rr**	Subtraher to registre	Rd ← Rd – Rr	Z N V C H
SUBI	**Rd, K**	Subtraher et register og konstant	Rd ← Rd – K	Z N V C H
TST	**Rd**	Test for null eller negativt innhold i reg.	Rd ← Rd AND RD	Z N V

4.2.4 Bitmanipulerende instruksjoner

MNEMONIC	Operander	Beskrivelse	Funksjon	Flagg
SBI	**A, b**	Sett bit i IO-register	IO(P,b) ← 1	
CBI	**A, b**	Nullstill bit i IO-register	IO(P,b) ← 0	
LSL	**Rd**	Venstreskift	Rd(n+1) ← Rd(n), Rd(0) ← 0	Z C N V
LSR	**Rd**	Logisk høyreskift	Rd(n) ← Rd(n+1), Rd(7) ← 0	Z C N V
ASR	**Rd**	Aritmetisk høyreskift	Rd(n) ← Rd(n+1), n = 0..6	Z C N V
ROL	**Rd**	Roter til venstre gjennom C-bit	Rd(0) ← C, Rd(n+1) ← Rd(n), C ← Rd(7)	Z C N V
ROR	**Rd**	Roter til høyre gjennom C-bit	Rd(7) ← C, Rd(n) ← Rd(n+1), C ← Rd(0)	Z C N V
NOP		No operation		
SEC		Sett C-flagg	C ← 1	C
CLC		Nullstill C-flagg	C ← 0	C
SEZ		Sett Z-flagg	Z ← 1	Z
CLZ		Nullstill Z-flagg	Z ← 0	Z

4.2.5 Hopp-instruksjoner

MNEMONIC	Operander	Beskrivelse	Funksjon	Flagg
RJMP	**k**	Relativt hopp	`PC ← PC + k + 1`	
IJMP		Indirekte hopp	`PC ← Z`	
RCALL	**k**	Relativt subrutinekall	`PC ← PC + k + 1`	
RET		Retur fra subrutine	`PC ← STACK`	
SBRC	**Rr, b**	Skip hvis bit i register er 0	`if (Rr(b) = 0)` `then PC ← PC + 2/3`	
SBRS	**Rr, b**	Skip hvis bit i register er 1	`if (Rr(b) = 1)` `then PC ← PC + 2/3`	
SBIC	**A, b**	Skip hvis bit i IO-register er 0	`if (IO(b) = 0)` `then PC ← PC + 2/3`	
SBIS	**A, b**	Skip hvis bit i IO-register er 1	`if (IO(b) = 1)` `then PC ← PC + 2/3`	
BREQ	**k**	Hopp hvis resultat er null	`if (Z=1)` `then PC ← PC + k + 1`	
BRNE	**k**	Hopp hvis resultat ikke er null	`if (Z=0)` `then PC ← PC + k + 1`	
BRCS	**k**	Hopp hvis C-flagg er 1	`if (C=1)` `then PC ← PC + k + 1`	
BRCC	**k**	Hopp hvis C-flagg er 0	`if (C=0)` `then PC ← PC + k + 1`	
BRSH	**k**	Hopp hvis resultat er større eller lik enn 0 (**uten** fortegn) etter CP/SUB	`if (op1 >= op2)` `then PC ← PC + k + 1`	
BRLO	**k**	Hopp hvis resultat er mindre enn 0 (**uten** fortegn) etter CP / SUB	`if (op1 < op2)` `then PC ← PC + k + 1`	
BRMI	**k**	Hopp hvis resultat er negativt (**med** fortegn)	`if (N=1)` `then PC ← PC + k + 1`	
BRPL	**k**	Hopp hvis resultat er positivt (**med** fortegn)	`if (N=0)` `then PC ← PC + k + 1`	
BERGE	**k**	Hopp hvis resultat er større eller lik 0 (**med** fortegn) etter CP / SUB	`if (op1 >= op2)` `then PC ← PC + k + 1`	
BRLT	**k**	Hopp hvis resultat er mindre enn null (**med** fortegn) etter CP / SUB	`if (op1 < op2)` `then PC ← PC + k + 1`	

4.3 Instruksjonseksempler

Typisk for en mikrokontroller av RISC-klassen er at de aritmetiske og logiske operasjonene utføres på operander som ligger lagret i registre, og resultatene skrives tilbake til et register. Det er altså viktig å kunne flytte inn til registrene fra IO-enheter og fra dataminnet, og likeledes å kunne overføre resultater fra registre tilbake til IO-enheter eller dataminne.

Noen eksempler på assemblyinstruksjoner for å flytte data:

MOV Rd, Rs

Kopierer data mellom to registre.

Rd ← Rs

Både Rd og Rs kan være et fritt valgt register r0 – r31.

Eksempel:
MOV r5, r17 ; kopierer data fra r17 til r5.

Instruksjonsformat:

LDI Rd, K

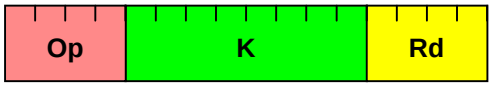
Kopierer data fra en konstant som oppgis i instruksjonen, til et register.
K: 0 – 255 (binært) / -127 - +127 (2's komplement)
Rd: r16 – r31

Rd ← K

Legg merke til at Rd er begrenset til den halvdel av registrene med høyest registerindeks.

Eksempel:
LDI r20, 120 ; Kopierer konstanten 120 til r20.

Instruksjonsformat:



LDS Rd, k

Kopierer data fra en lokasjon i dataområdet som en direkte adresse i instruksjonen peker ut, til et register.

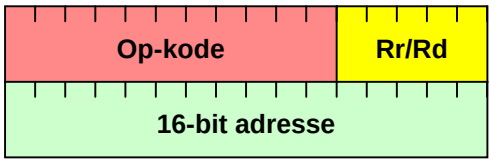
k: 00h – DFh (høyere adresser enn DFh er i ekstern SRAM)

Rd: r0 – r31

Rd ← M[k]

Eksempel:
LDS r12, 90h; Kopierer en databyte fra adresse 90h til r12.

Instruksjonsformat:



STS k, Rr

Kopierer data fra et register til en lokasjon i dataområdet som en direkte adresse i instruksjonen peker ut.

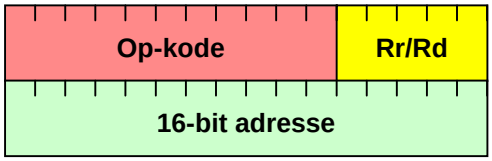
k: 00h – DFh (høyere adresser er i ekstern SRAM)

Rd: r0 – r31

M[k] ← Rd

Eksempel:
STS 80hh, r5; Kopierer en databyte fra r5 til adresse 80h

Instruksjonsformat:



En sekvens av instruksjoner av instruksjoner som kopierer konstanten 75 til adresse A0h i SRAM kan settes opp slik:

LDI r16, 75
STS 0xA0, r6 ; legg merke til hvordan en hex-verdi angis.

Det finnes ingen instruksjoner for å kopierer en konstant direkte til SRAM. Vi må derfor først kopiere konstanten til et register.

4.4 Adresseringsformer for program og data

I dette avsnittet skal de adresseringsformane som AVR benytter forklares. Samtlige av adresseringsformene er velkjent fra litteraturen som standard adresseringsformer.

En adresseringsform kan sies å være en metode som MCU benytter for å rpeke ut eller referere data og instruksjoner i hukommelsen. Et annet ord for det samme begrepet er **adresseringsmodus** (**adresseringsmodi** i flertall).

Merk at mange instruksjoner benytter 2 operander. Disse instruksjonene har da en kombinasjon av 2 av disse adresseringsformene, der register direkte alltid er den ene formen.

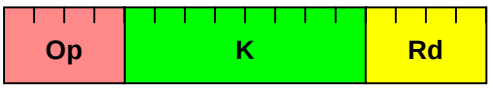
4.4.1 Immediate

Immediate adressering er ikke noe annet enn at den ene operanden som skal benyttes i instruksjonen, befinner seg i instruksjonen selv i form av en konstant. I instruksjonsformatet vist nedenfor, inneholder feltet K en 8-bits konstant. Den andre operanden er et register.

De fleste assemblyinstruksjoner som ender på 'I' (immediate) er av denne typen, som SUBI, ANDI, ORI, LDI er av denne typen. Felles for den alle er at det kun er lov å benytte et register i området r16 – r31 som den andre operand.

Eksempel:
LDI r16, 26;

Instruksjonformat:



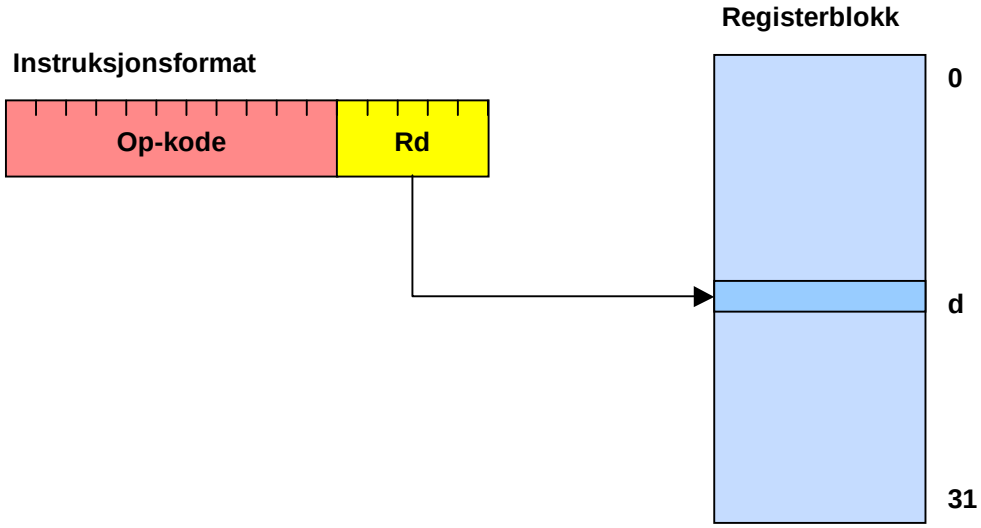
4.4.2 Register direkte – et register

Operanden et register med indeks angitt i et 5-bits felt i instruksjoen. Dette gir mulighet for å nå alle 32 generelle registre i registerbanken. Derimot kan man ikke nå IO-registre eller andre registre i MCU med denne adresseringsformen.

De fleste instruksjonene benytter denne adresseringsformen – alene eller sammen med andre adresseringsformer. Instruksjoner som kun benytter denne formen er eksempelvis: COM, NEG, DEC, INC, CLR, SER, TST, LSR, LSL, ASR, ROR, SWAP.

Eksempel:

```
INC r4 ; inkrementer r4 (legg 1 til verdien av register r4)
```

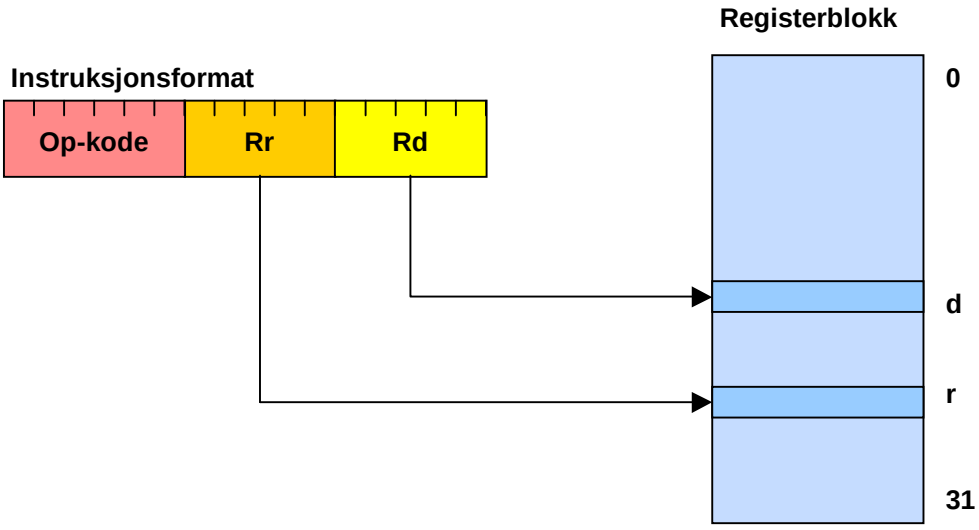


4.4.3 Register direkte – to registre

Denne formen tilsvarer foregående, med forskjellen at 2 registre er involvert. Instruksjonene har to operander, Rd og Rr. Resultatet lagres i register Rd.

To 5-bits felt benyttes for å angi de 2 registrene. Dette betyr at alle 32 registre i registerbanken er tilgjengelig.

Bl. a. disse instruksjonene benytter adressingsformen: ADD, SUB, AND, OR, EOR.



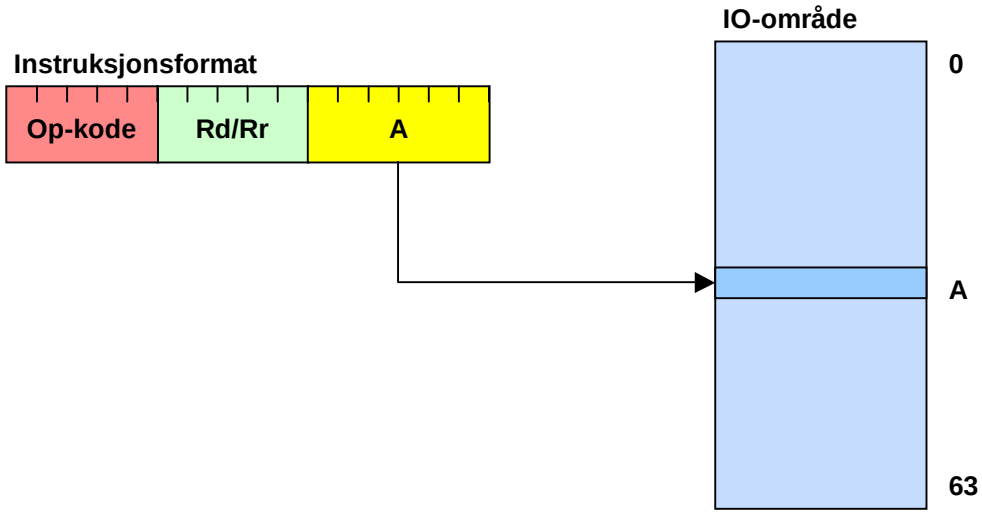
4.4.4 IO-direkte

IO-regsitrene kan adresseres om de var et område i dataminne. Dette tilfellet skal ikke omtales i dette avsnittet (se neste avsnitt). Her skal vi se på tilfellet der man benytter den spesielle IO-adresseringsformen. Dette krever bruk av spesielle IO-instruksjoner som: IN, OUT, SBI, CBI, SBIS, SBIC.

I disse tilfellene må den andre operanden være et arbeidsregister (r0 – r31). En av de 64 IO-adressene spesifiseres i et 6-bits felt i instruksjonsformatet.

Eksempel:

IN r3, PORTB ; r3 ← PORTB



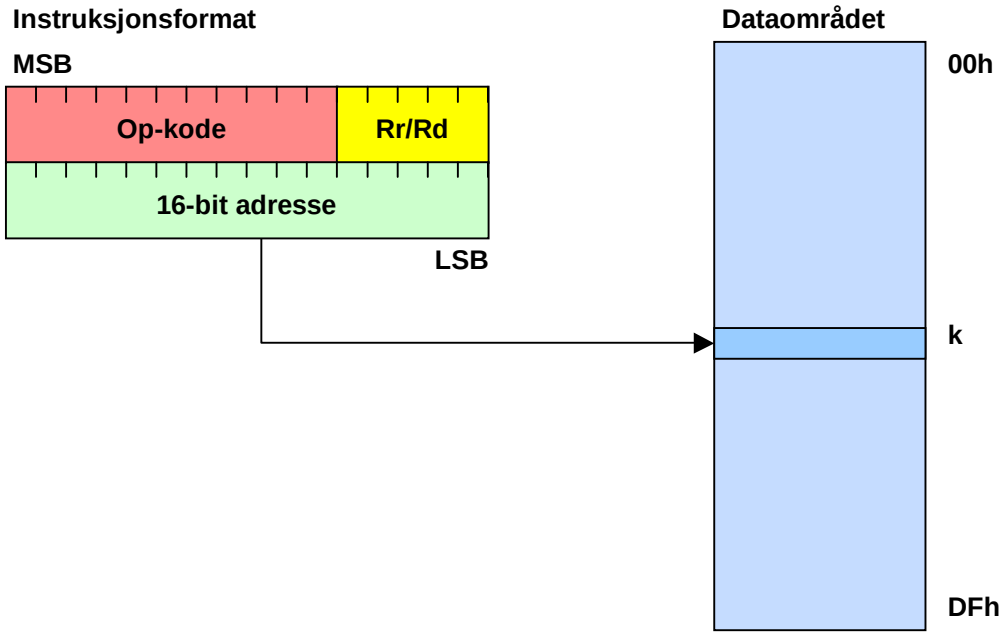
4.4.5 Data direkte

Instruksjonene som benytter data-direkte adressering opptar 2 ord (32 bit) i programminnet. Det minst signifikante ordet i adressen inneholder en 16-bits adresse som peker ut en byte i brukerdata området. Dette området inkluderer SRAM, arbeidsregistre, IO-registre og eksternt dataminne opp til FFFFh.

En direkte registerreferanse inngår også i instruksjonene som benytter data-direkte adressering. Aktuelle instruksjoner er LDS og STS.

Eksempel:

```
LDS r1, 0x7F ; r1 ← M[7F]
```



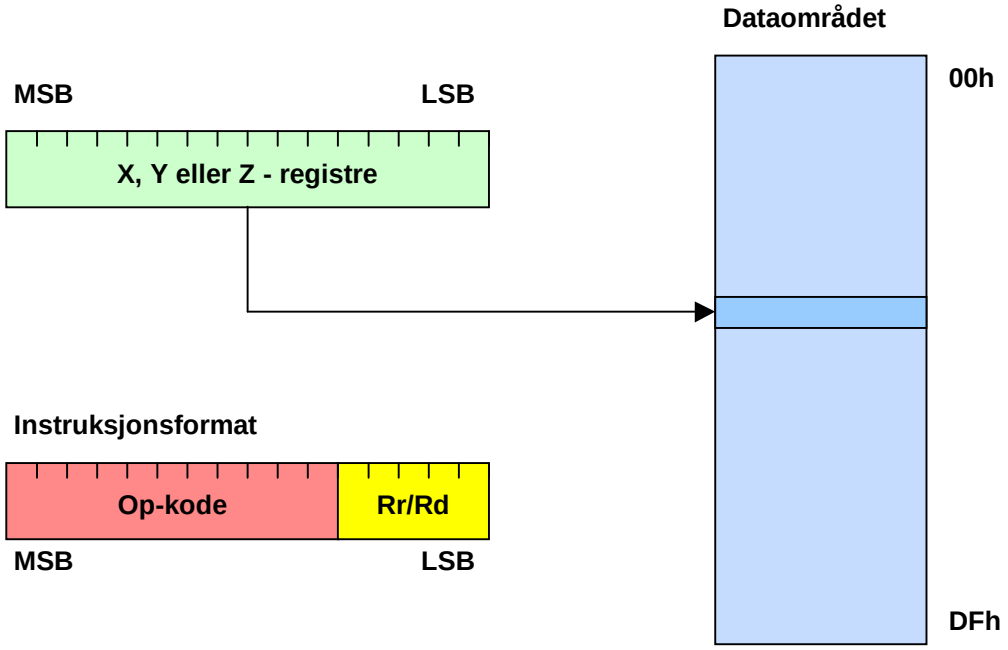
4.4.6 Data indirekte

I data-indirekte adressering hentes en 16-bits operandadresse fra et av adresseregistrene X, Y eller Z. Disse registrene hører jo med i registerbanken, men kan benyttes som dobbeltregister av spesielle instruksjoner som: LD og ST.

Den andre operanden i forbindelse med denne adresseringsformen er alltid et register i registerbanken.

Eksempel:

```
ST 0x80, r1 ; M[80h] ← r1
```



4.4.7 Data indirekte med forskyvning

En av de meste kompliserte adresseringsformene er data-indirekte med forskyvning (engelsk: displacement). Den andre operanden må være et register.

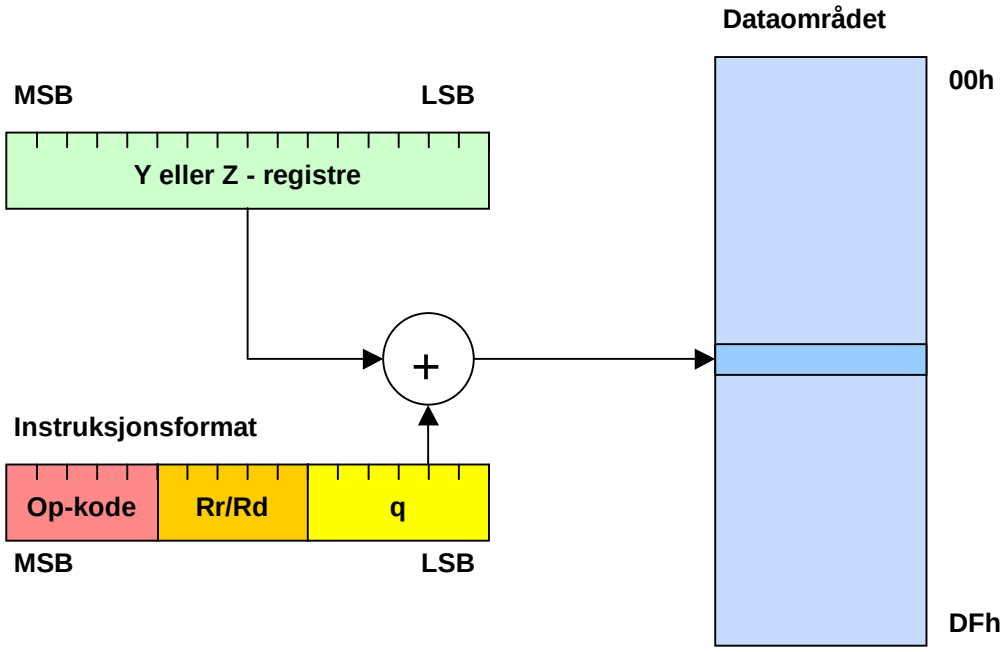
Instruksjonene som er aktuelle er LDD og STD.

Denne adresseringsformen benyttes ofte i forbindelse med å hente ut verdier fra en tabell eller vektor i dataområdet. Selve forskyvningen angis som en 6-bits konstant q. Hvis denne konstanten tilsvarer startadressen til en tabell eller vektor, vil verdien som er lagret i adresseregisteret tilsvare indeksen i tabellen.

En annen måte å benytte denne adresseringsformen på er i forbindelse med datastrukturer. Datastrukturer er et dataområde med verdier som hører sammen på en eller annen måte. Et eksempel kan være vekt, høyde og alder på en person. Z-registeret kan da inneholde basisadressen i denne datastrukturen, mens forskyvningskonstanten vil angi de enkelte medlemmene i strukturen.

Eksempel:
Anta at Z inneholder adressen til datastrukturen og at vekt, høyde og alder henholdsvis har forskyvningene 1, 2, og 3:

```
LDD r10, Z + 1 ; r10 ← M[Z + 1] – f. eks. r10 ← vekt
LDD r11, Z + 2 ; r11 ← M[Z + 2] – f. eks. r11 ← høyde
LDD r12, Z + 3 ; r12 ← M[Z + 3] – f. eks. r12 ← alder
```



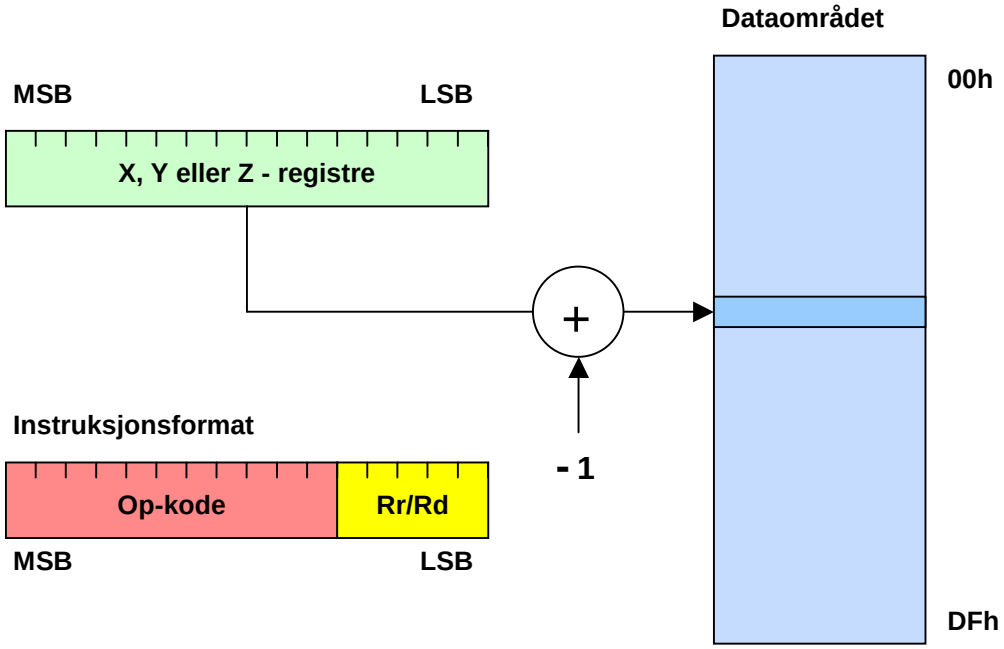
4.4.8 Dataminne indirekte med predekrement

Denne adresseringsformen ligner data-indirekte adressering, men skiller seg fra denne formen ved at innholdet i adresseregisteret dekrementeres[1] før adressen benyttes til å peke ut en byte i dataområdet.

Adresseringsformen er svært nyttig når man skal lese en tabell fra høye mot lave indekser. Aktuelle instruksjoner er LD, ST.

Eksempel:

```
ST –Y, r5 ; Først Y ← Y-1,  så M[Y] ← r5.
```



[1] Dekrementere betyr å subtrahere 1 fra verdien.

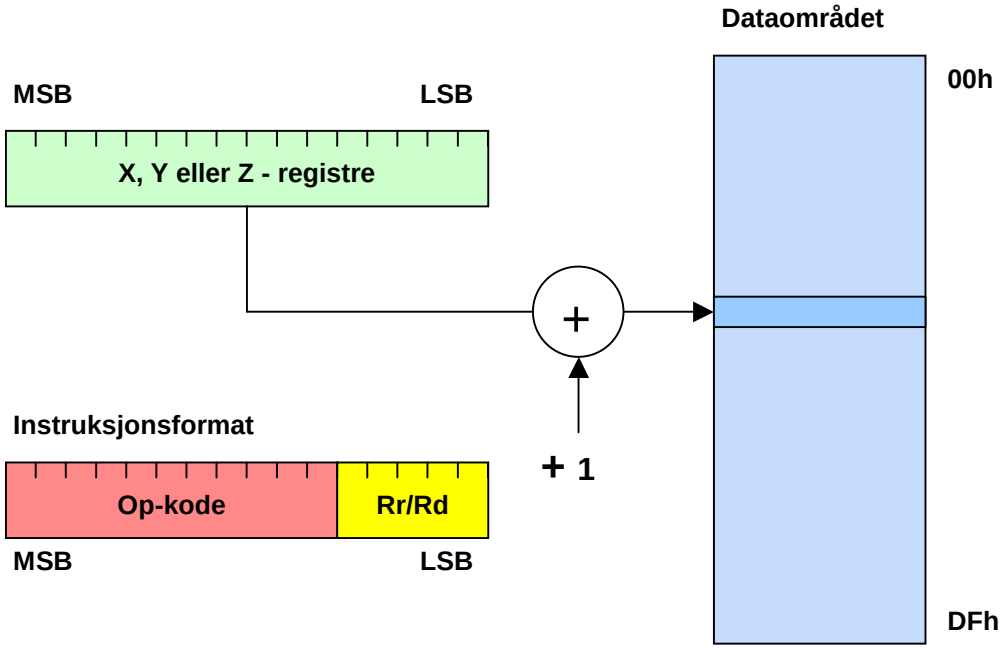
4.4.9 Dataminne indirekte med postinkrement

Som foregående adresseringsform, men skiller seg fra denne formen ved at innholdet i adresseregisteret inkrementeres[1] etter at adressen er benyttet til å peke ut en byte i dataområdet.

Adresseringsformen er svært nyttig når man skal lese en tabell fra lave mot høye indekser. Aktuelle instruksjoner er LD, ST.

Eksempel:

```
LD r6, X+ ; Først r6 ← M[X],  så X ← X+1.
```



[1] Inkrementerer betyr å addere 1 til verdien.

4.4.10 Program indirekte

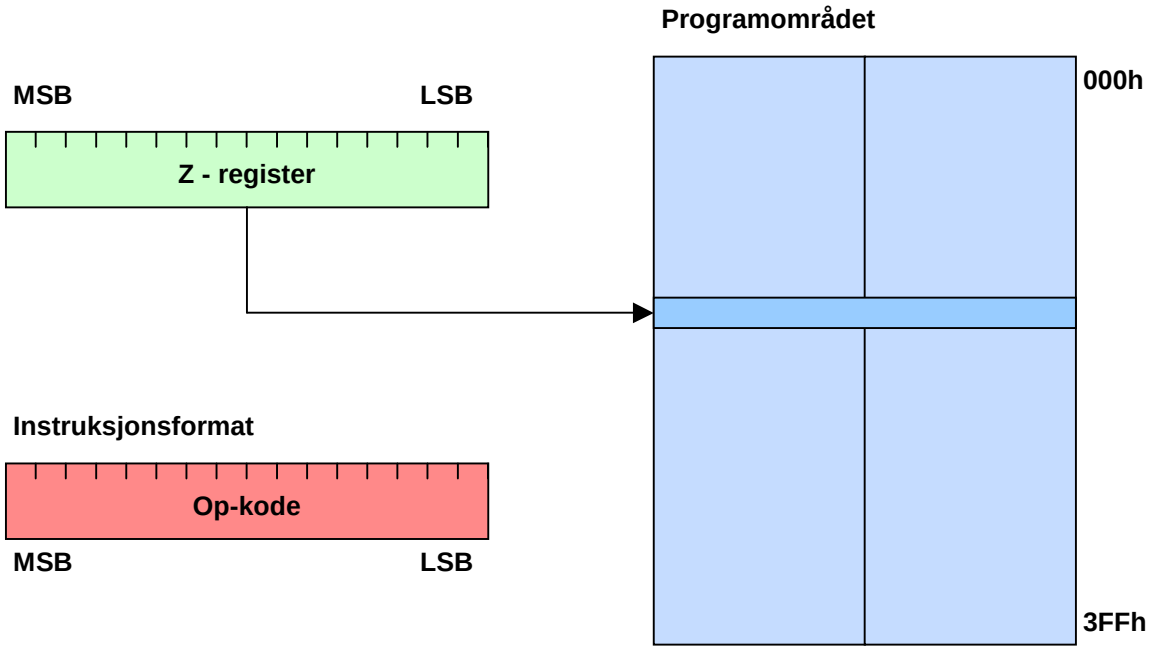
Normal programflyt tilsier at neste instruksjon følger direkte etter forrige. Det betyr at PC ← PC +1.

Denne typen adressering benyttes for å utføre et hopp i instruksjonssekvensen. Et hopp medfører at programtellerregisteret får en annen verdi i forhold til 1 adresse høyere enn forrige instruksjon. En måte å gjøre dette på er å hente neste adresse fra adresseregisteret Z.

Aktuell instruksjon her er IJMP.

Eksempel:

```
IJMP ; PC ← Z
```



4.4.11 Relativ program-adressering

Normal programflyt tilsier at neste instruksjon følger direkte etter forrige. Det betyr at PC ← PC +1.

Denne typen adressering benyttes for å utføre et hopp i instruksjonssekvensen. Et hopp medfører at programtellerregisteret får en annen verdi i forhold til 1 adresse høyere enn forrige instruksjon. En måte å gjøre dette på er å angi en konstant i instruksjonen selv som angir en verdi som skal legges til neste programminneadresse. Hoppet blir med dette relativt til nåværende verdi for PC.

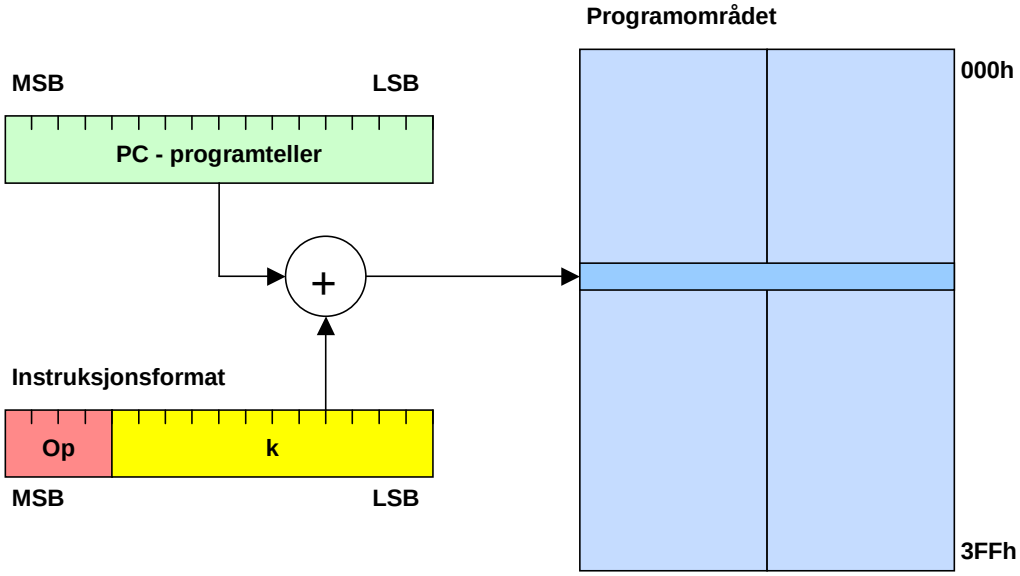
Den konstantverdien som legges til adressen er et 12-bits 2's komplementtall og kan dermed variere mellom –2048 og +2047.

For den aktuelle mikrokontrolleren AVR 2313 er programminnet kun på 1024 ord. Man kan derfor hoppe over hele det aktuelle området vha. relativ adressering. For kontrollere med større FLASH-minne enn 2K ord, blir det derfor nødvendig å benytte et indirekte hopp hvis et hopp større enn 2048 ønskes.

Aktuelle instruksjoner her er RJMP, RCALL.

Eksempel:

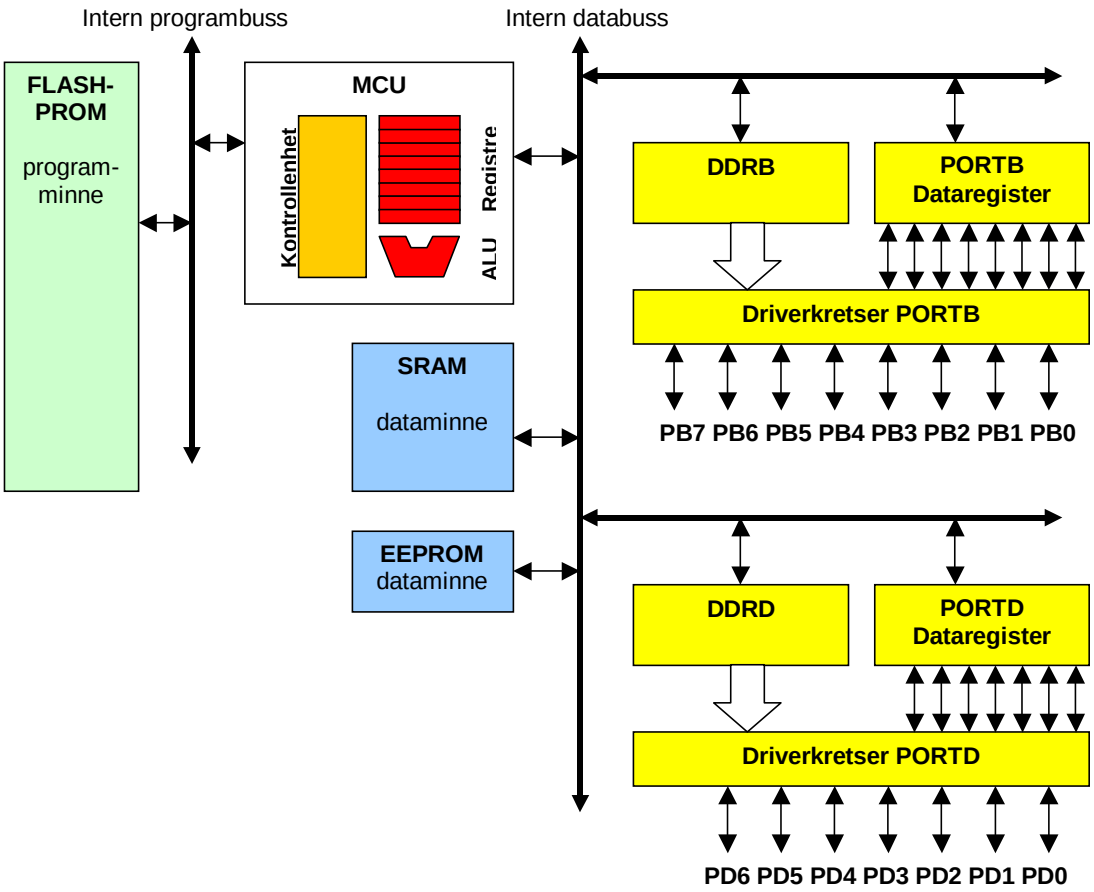
`RJMP 200; PC ← PC + 200 + 1`

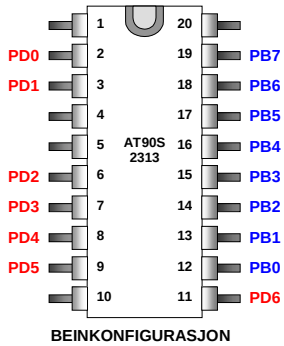


4.5 Digital IO

I dette grunnkurset i mikrokontrollerteknikk skal vi i hovedsak konsentrere oss om digital IO når det gjelder å studere de innebygde IO-funksjonene for AVR-familien av mikrokontrollere. For AVR 2313 dreier det seg om 15 digitale IO-linjer fordelt på 2 porter. PORTB og PORTD.

AVR 2313 Digital IO





BEINKONFIGURASJON

PORTB er en full 8-bits port, mens PORTD kun har 7 IO-linjer. IO-linjene tilknyttet PORTB betegnes PB.0 – PB.7, mens tilsvarende for PORTD er PD.0 – PD.6.

Hver av de digitale portene består av 4 delenheter, et sett av driverkretser, et dataretningsregister, et dataregister for utsignaler og en lesebuffer for innsignaler.

4.5.1 Driverkretser

Driverkretsene danner det fysiske grensesnittet mot omverdenen og er direkte tilknyttet mikrokontrollernes bein. Driverkretsene er styrt av retningskretsene og kan brukes både for å sette digitale signaler ut på mikrokontrollerens bein, og for å lese digitale innsignaler fra mikrokontrollerens bein. Et 3-state buffer er sentral i driverkretsene.

4.5.2 Dataretningregister DDR_

Hvert bit i driverne kan styres uavhengig av hverandre. Det som avgjør om et bein skal benyttes for inn- eller utoperasjoner, er om det er skrevet en 1 (ut) eller 0 (inn) i dataretningsregisteret på aktuell bitposisjon.

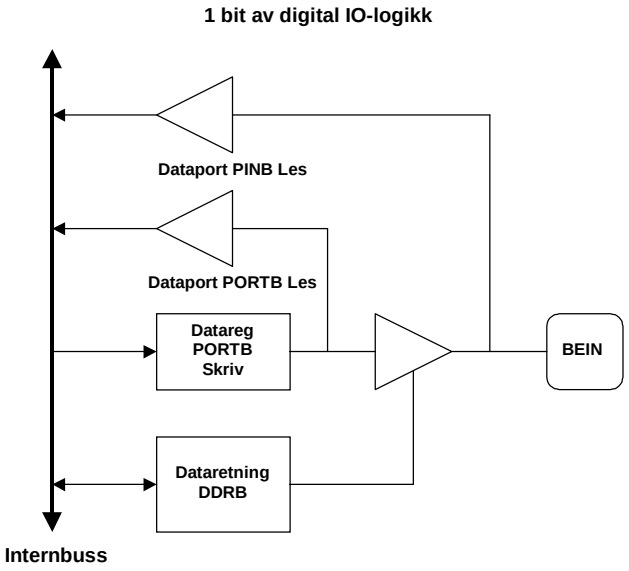
4.5.3 Dataregister PORT_

De enkelte bitverdiene skrives ut via dataregistrene. Å skrive til dataregisteret fører til at de IO-linjene som er konfigurert sum utlinjer setter tilsvarende logiske nivåer på beina til mikrokontrolleren via driverkretsene. IO-linjer som ikke er konfigurert som utlinjer forandres ikke.

Det skjer ingen skade om vi skriver verdier til et bit som er konfigurert som inn. Denne bitverdien kobles da ikke fram til beinet.

4.5.4 Innbuffer PIN_

Når vi skal hente inn logiske verdier fra omverdenen via dataporten, blir de signalene som er tilknyttet mikrokontrollerens bein, lest via et databuffer som kalles PIN_ (f. eks. PINB).



Dette skjer uavhengig av om vi har konfigurert IO-linjene som inn eller ut. Hvis vi har konfigurert en innlinje, er det et ytre påtrykket signal som leses. Hvis vi har konfigurert en utlinje vil vi via PIN_ lese utsignalet.

Se figuren.

4.5.5 IO-instruksjoner

De instruksjonene som i første rekke benyttes i forhold til IO-portene er IN og OUT.

Eksempel:

Vi ønsker å sette opp PORTD til å ha 7 inn-linjer (PD.7-0) og PORTB til å ha 8 ut-linjer (PB.7-0). Fra instruksjonstabellene ser vi at OUT kun kan benyttes i forbindelse med innholdet i et register. Derfor må vi på forhånd leste inn de ønskede verdiene i et arbeidsregister:

```
LDI r16, 0xFF; r16 ← 1111 1111B
OUT DDRB, r16; Sett utretning for PORTB
LDI r16, 0x00; r16 ← 0000 0000B
OUT DDRD, r16; Sett innretning for PORTD
```

Å skrive ut verdien 0xA5 til PORTB utføres slik:

```
LDI r16, 0xA5; r16 ← 1010 0101B
OUT PORTB, r16;
```

Å lese inn verdiene til signalene som er knyttet til PORTD til registeret r10 utføres på følgende måte:

```
IN r10, PIND;
```

Isteden for å skrive en hel byte som ved OUT, kan man også benytte noen spesialinstruksjoner som nullstiller og setter enkeltbit direkte. Vi skal nå forklare instruksjonene SBI, CBI.

SBI – sett bit i IO-register til 1

Denne instruksjonen kam benyttes hvis vi ønsker å sette (til 1) et bit på en utport uten å forandre på verdiene av de andre bitene på porten. Anta at PORTB er konfigurert som en ren utport og at bitmønsteret som ligger på porten er 11110001. Vi ønsker nå å forandre verdien til bitposisjon 7 fra 1 til 0. Vi kan da utføre instruksjonen:

```
SBI PORTB, 7; Merk at operand 2 peker ut bitposisjonen
```

CBI – nullstill bit i IO-register

Denne instruksjonen utfører den motsatte funksjonen av SBI. Anta at man ønsker å sette tilbake bitposisjon 7 fra 1 til 0:

```
CBI PORTB, 7;
```

4.6 Viktige spesialregistre

AVR-familien mikrokontrollere har noen viktige registre som kan adresseres som IO-registre selv om de til dels ikke har så mye med inn-/utfunksjoner å gjøre.

Vi skal beskrive 2 av disse her.

4.6.1 Statusregisteret - SREG

Et viktig register i mikrokontrolleren er statusregisteret. Registeret er direkte knyttet opp mot ALU og avspeiler status for instruksjoner som er utført av ALU. Statusregisteret lar programmereren sjekke og unngå feilsituasjoner som f. eks. overflow, eller divisjon med null. Registeret er på 8 bit og formatert som vist nedenfor.

De enkelte bitene i statusregisteret betegnes **flagg** – de signaliserer en alarm- eller varslingstilstand.

Formatet til SREG

Flagg	Beskrivelse
I	Global Interrupt Enable. Dette flagget settes av programmereren. Bruken av flagget hører ikke hjemme i dette kurset.
T	Bit Copy Storage. Kan benyttes av programmerer som mellomlagring av bit fra et register i registerbanken. Kan benyttes som test i betingede hopp-instruksjoner.
H	Half-Carry Indikerer at en mente er generert mellom trinn 3 og 4 i ALU-en i en aritmetisk instruksjon.
S	Sign bit. Dette er et bit som indikerer fortegnet etter en aritmetisk operasjon.
V	Overflow. Indikerer en overflowsituasjon.
N	Negative Indikerer at et resultat er negativt.
Z	Zero Indikerer at resultatet av en aritmetisk-logisk operasjon er 0.
C	Carry Indikerer at en lånemente er generert fra trinn 7 i ALU.

Ikke alle instruksjoner i instruksjonssettet kan forandre på flaggverdiene i SREG. Sjekk tabellene over maskininstruksjonene for å finne hvilke instruksjoner som kan forandre flagg og hvilke flagg de enkelte instruksjonene kan forandre.

Merk at flaggene i SREG ikke resettes av seg selv. De blir ikke forandret før en ny instruksjon som kan forandre vedkommende flagg.

For å være sikker på at det er de aktuelle flaggene som befinner seg i SREG kan det lønne seg å generere flaggene like før ban benytter dem for testing i programmet.

4.6.2 MCU Control Register – MCUCR

Detter registeret kontrollerer noen MCU-funksjoner som ikke har funnet sin plass i noen av de mange andre kontrollregistrene som styrer AVR-mikrokontrollerne. Ingen av funksjonene i dette registeret er sentrale i dekke kurset, men desto mer viktige i mer avanser bruk av AVR 2313.

Formatet til MCUCR

Bit navn	Standard verdi	Num.	Beskrivelse	
-	0	7	Ledig	
-	0	6	Ledig	
SE	0	5	Sleep enable Må settes til 1 for å for å kunne entre sleep mode når SLEEP instruksjonen utføres.	
SM	0	4	Sleep mode Idle mode velges når SM=0. Power-down mode velges når SM=1.	
ISC11 ISC10	0 0	3 2	0 0	Lavt nivå på INT1 genererer avbruddsforspørsel
			0 1	Ledig
			1 0	Negativ flanke på INT1 genererer avbrudd.
			1 1	Positiv flanke på INT1 genererer avbrudd.
ISC01 ISC00	0 0	1 0	0 0	Lavt nivå på INT0 genererer avbruddsforspørsel
			0 1	Ledig
			1 0	Negativ flanke på INT0 genererer avbrudd.
			1 1	Positiv flanke på INT0 genererer avbrudd.

5 Instruksjonsbeskrivelser

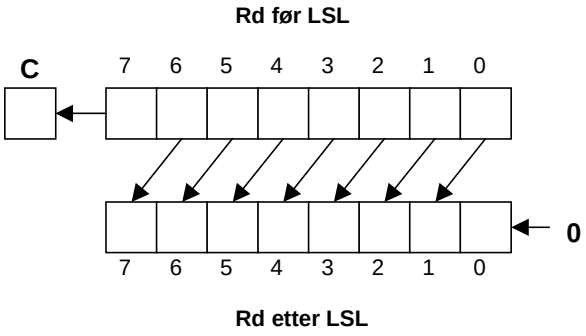
Noen av de mest benyttede instruksjonene beskrives i dette kapittelet. Det er lagt vekt på instruksjonene som ikke er helt selvforklarende som de enkleste aritmetisk-logiske instruksjonene og load/store-instruksjoner. Disse er til dels også forklart i tidligere kapitler.

5.1 Bitmanipulerende instruksjoner

Mange av instruksjonene i instruksjonssettet til AVR arbeider med hele registre som operander. Det gjelder f. eks. alle aritmetiske og logiske instruksjoner. Noen instruksjoner arbeider imidlertid ned på bitnivå, vi kan manipulerer på enkeltbitene i et register.

LSL Rd – logisk venstreskift

Denne instruksjonen skifter alle bit i det angitte registeret en posisjon mot venstre. Rd(7) skiftes ut til C-flagget i SREG. Ny verdi for Rd(0) blir 0.



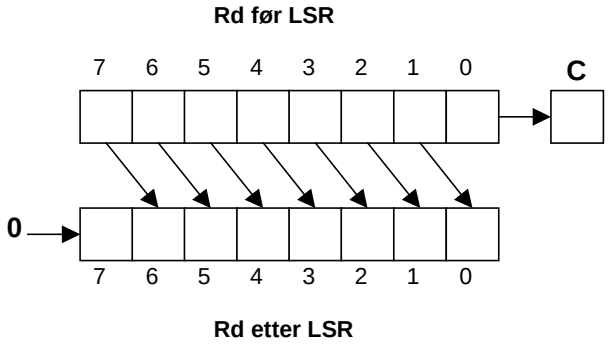
- S: N **xor** V, For tests basert på fortegn og 2's komplement.
- V: N **xor** C (For N og C etter skift)
- N: R7 (mest signifikante bit i resultat etter skift)
- Z: 1 hvis resultatet etter skift is 0x00; 0 ellers.
- C: Rd7 (mest signifikante bit i register før skift)

Eksempel:

```
LSL r8
```

LSR Rd – logisk høyreskift

Denne instruksjonen skifter alle bit i det angitte registeret en posisjon mot høyre. Rd(0) skiftes ut til C-flagget i SREG. Ny verdi for Rd(7) blir 0.



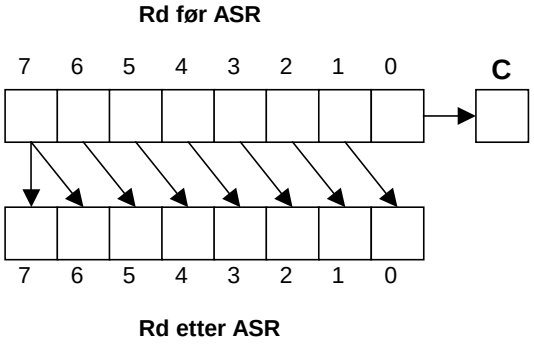
- S: N xor V, For tests basert på fortegn og 2's komplement.
- V: N xor C (For N og C etter skift)
- N: 0
- Z: 1 hvis resultatet etter skift is 0x00; 0 ellers.
- C: Rd0 (minst signifikante bit i register før skift)

Eksempel:

```
LSR r10
```

ASR Rd – aritmetisk høyreskift

Denne instruksjonen skifter alle bit i det angitte registeret en posisjon mot høyre. Rd(0) skiftes ut til C-flagget i SREG. Verdi for Rd(7) blir den samme som før skift. Denne formen for skift vil alltid beholde riktig fortegn (2's komplement) etter et skift. Dette er nyttig hvis man ønsker å betrakte et høyreskift som å dele på 2.



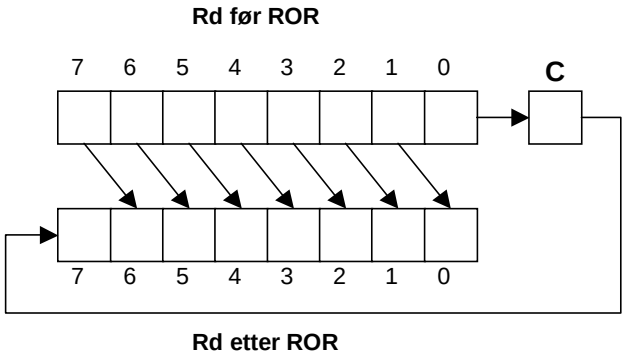
- S: N xor V, For tests basert på fortegn og 2's komplement.
- V: N xor C (For N og C etter skift)
- N: Rd(7) (mest signifikante bit i register før skift)
- Z: 1 hvis resultatet etter skift is 0x00; 0 ellers.
- C: Rd0 (minst signifikante bit i register før skift)

Eksempel:

```
ASR r0
```

ROR Rd – roter register mot høyre gjennom C-flagg

Ligner svært på LSR, men med den forskjellen at innholdet av C-flagget før skift overføres til Rd(7) (mest signifikante bitposisjon) etter skiftet. Ved hjelp av denne instruksjonen kan man teste verdiene til alle bit i registeret via C-flagget uten å ødelegge innholdet. Kan også sammen med LSR benyttes til å høyreskifte et ord bestående av flere bytes.



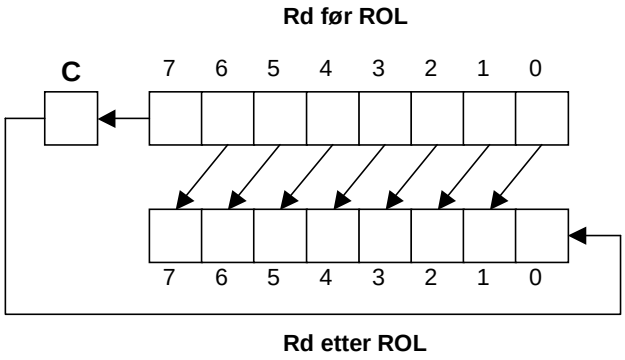
- S: N **xor** V, For tests basert på fortegn og 2's komplement.
- V: N **xor** C (For N og C etter skift)
- N: R(7) (mest signifikante bit i resultat)
- Z: 1 hvis resultatet etter skift is 0x00; 0 ellers.
- C: Rd(0) (minst signifikante bit i register før skift)

Eksempel:

```
LSR r19      ; Divide r19:r18 by two
ROR r18      ; r19:r18 is an unsigned two-byte integer
```

ROL Rd – roter register mot venstre gjennom C-flagg

Ligner svært på LSL, men med den forskjellen at innholdet av C-flagget før skift overføres til Rd(0) (minst signifikante bitposisjon) etter skiftet. Ved hjelp av denne instruksjonen kan man teste verdiene til alle bit i registeret via C-flagget uten å ødelegge innholdet.Kan også benyttes til å venstreskifte (multplisere med 2) et ord bestående av flere bytes.



- S: N **xor** V, For tests basert på fortegn og 2's komplement.
- V: N **xor** C (For N og C etter skift)
- N: R(7) (mest signifikante bit i resultat)
- Z: 1 hvis resultatet etter skift is 0x00; 0 ellers.
- C: Rd(7) (minst signifikante bit i register før skift)

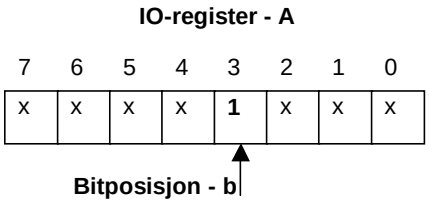
Eksempel:

```
LSL r18 ; Multiply r19:r18 by two
ROL r19 ; r19:r18 is a signed or unsigned two-byte
```

SBI A, b – Set bit i IO-register til 1

Ønsker man å sette et bit i et IO-register (f. eks. PORTB) uavhengig av de andre bitene i IO-registeret, kan denne instruksjonen benyttes.

A er nummeret til IO-registeret og b er bitnummeret (0 – 7). I stedet for å sette inn en tallkonstant for IO-adressen, benyttes det symbolske registernavnet i assemblyprogrammet – f. eks. DDRD.

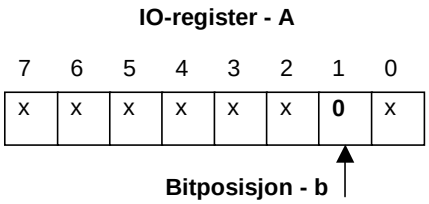


Eksempel:

```
SBI PORTB, 3
```

CBI A, b – Nullstill bit i IO-register

Tilsvarende som for SBI, men for nullstilling av enkle bit i IO-registre.



Eksempel:

```
CBI PORTB, 1
```

5.2 Hoppinstruksjoner

Hoppinstruksjoner benyttes for å forandre dens sekvensielle utførelsen av programinstruksjonene. AVR-familien benytter i hovedsak 5 former for hoppinstruksjoner:

- Ubetingede hoppinstruksjoner
- Betingede hoppinstruksjoner
- Betingede skipinstruksjoner
- Subrutinekall
- Retur-hopp fra subrutiner.

Det er de tre første kategoriene som skal dekkes i dette avsnittet.

5.2.1 Symbolske programadresser

Før noen hoppinstruksjoner forklares skal vi først se på et begrep som benyttes mye i assemblyprogrammering – **label**[1] eller en symbolsk programadresse. Isteden for å forsøke å holde styr på adressene til de enkelte instruksjonene i programmet, kan vi skrive inn et navn etterfulgt av et kolon (:) ut for en instruksjon vi ønsker å huske adressen til. Dette navnet er en label. Assembleren vil selv regne ut den riktige adressen til instruksjonen som følger etter labelen og vi slipper å tenke mer på dette.

Eksempel:

```
        clr r0
        clr r1
        inc r1
adr1:   inc r1
        add r0, r1
adr2:
        dec r1
```

I eksemplet er adr1 og adr2 to labeler. En label kan enten settes inn foran instruksjonen på samme linje, eller på en linje for seg selv. Idet siste tilfellet, er adressen som labelen representerer adressen til første instruksjon etter labelen.

Det er vanlig å sette labelene helt inn til venstremargen, mens programinstruksjonene settes med et visst innykk.

[1] På norsk kunne vi benytte begrepet **merkelapp**, **etikett** eller noe tilsvarende. Jeg velger imidlertid å benytte den engelske betegnelsen **label**.

5.2.2 Ubetingede hoppinstruksjoner

Ubetingede hopp benyttes der vi ønsker å hoppe til et fast sted i programmet uavhengig av hva som har skjedd under den tidligere programutførelsen. AT90S2313 kan benytte instruksjonene RJMP og IJMP for å utføre ubetingede hopp. Vi skal bare studere RJMP instruksjonen.

En ubetinget hoppinstruksjon vil egne seg til å lage *evige* løkker som aldri kan avsluttes.

RJMP k – relativt hopp

Start neste instruksjon fra adressen PC + k + 1. Her er PC (program counter) adressen til RJMP-instruksjonen selv. K er en 12-bits konstant (som kan være negativ eller positiv). Positiv verdi for k indikerer et hopp framover i programmet, mens negativ verdi angir hopp tilbake til tidligere utførte instruksjoner. Setter man k = -1 hoppes det tilbake til RJMP-instruksjonen selv.

Konstanten k er på 12 bit (i 2's komplement) som tilsvarer verdier fra –2048 til +2047. For AT90S2313 som har kun plass til 1024 instruksjoner i FLASH programminnet, betyr det at RJMP kan nå hele programminnet uten videre.

I praksis, når man skriver et assemblyprogram, vil assembleren beregne det relative hoppet for deg automatisk. For Atmel's assembler som følger med **AVR studio** er det rett og slett ikke lov for programmereren å sette in den relative hoppadressen. Man skal alltid sette inn en symbolsk hoppadresse i form av en **label**. Dette gjelder for alle relative hoppinstruksjoner.

Eksempel:

```
start:
      add r1, r25
      …
      …
      rjmp start
```

I dette eksempelet vil programmet alltid hoppe tilbake til labelen start, når det kommer til hoppinstruksjonen.

Hoppinstruksjoner forandrer ikke på noen flagg i SREG.

5.2.3 Betingede hoppinstruksjoner

I de fleste tilfeller man ønsker å benytte hopp, vil dette hoppet være betinget av noe som har skjedd under utførelsen av programmet. Det kan f. eks. være at et signal som kan leses av på en av portene har forandret verdi, eller at resultatet av en beregning overstiger en på forhånd angitt grense.

Avhengig av slike hendelser kan så programmet gå en av to mulige veier:

1. Det kan fortsette rett videre hvis hendelsen ikke inntraff, eller
2. Programmet kan hoppe til en ny adresse som angis i instruksjonen hvis hendelsen inntraff.

Hendelser det går å teste på er i hovedsak om et nærmere spesifisert bit i SREG er 0 eller 1, eller en test på gitte kombinasjoner av bit i SREG.

Formen på alle instruksjonene av denne typen er: **BRxx**. Her er **BR** en forkortelse for det engelske **branch** (avgreining) og benyttes med betydningen hopp i assemblyprogrammering. **xx** er den betingelsen som skal testes for å avgjøre om hoppet skal gjennomføres eller ikke.

Form:

```
BRxx label; hopp hvis test xx er sann
```

Når man benytter betingede hoppinstruksjoner, er det svært viktig å huske på at de verdiene som det testes på i SREG, er det verdiene som ligger i SREG når BRxx instruksjonen starter. Det er derfor viktig å ikke utføre noen instruksjoner som medfører at SREG-flaggene forandres mellom den situasjonen som man ønsker å test oppstod, til selve BRxx instruksjonen utføres.

De mest aktuelle betingede hoppinstruksjonene listes opp her:

BREQ - hopp hvis Z-flagg er 1

BRNE - hopp hvis Z-flagg er 0

BRCS – hopp hvis C-flagg er 1

BRCC – hopp hvis C-flagg er 0

BRSH – hopp hvis større eller lik. Gjelder for bruk av ren **binærkode uten fortegn**. Benyttes typisk etter en SUB Rd, Rr / CP Rd, Rr og gir et hopp hvis Rd >= Rr.

BRLO – hopp hvis mindre. Gjelder for bruk av ren **binærkode uten fortegn**. Benyttes typisk etter en SUB Rd, Rr / CP Rd, Rr og gir et hopp hvis Rd < Rr.

BRGE – hopp hvis større eller lik. Gjelder for bruk av **2's komplement**. Benyttes typisk etter en SUB Rd, Rr / CP Rd, Rr og gir et hopp hvis Rd >= Rr.

BRLT - hopp hvis mindre. Gjelder for bruk av **2's komplement**. Benyttes typisk etter en SUB Rd, Rr / CP Rd, Rr og gir et hopp hvis Rd < Rr.

Eksempel:

```
LDI r16, 0x55
LDI r17, 0x44; her er r16 > r17
SUB r16, r17 ; setter flagg i SREG
BRLO adr1    ; feiler, men modifiserer ikke flagg i SREG
BRSH adr2    ; ok, hopp foretas
```

5.2.4 Skipinstruksjoner

Skipinstruksjoner kan benyttes som alternativ til betingede hoppinstruksjoner. Det spesielle med skipinstruksjonene er at det eneste hoppalternativet er å hoppe over (skippe) neste instruksjon. Det betyr at hvis man ønsker å hoppe lenger, må man sette inn en ubetinget hoppinstruksjon på den plassen man eventuelt kan hoppe over.

Det attraktive med skipinstruksjonene er at de gjør det mulig å teste direkte på enkeltbit i registre og IO-registre.

SBRC Rr, b
– skip neste instruksjon hvis bit b i register Rr er 0.

SBRS Rr, b
- skip neste instruksjon hvis bit b i register Rr er 1.

SBIC A, b
– skip neste instruksjon hvis bit b i IO-register A er 0.

SBIS A, b
- skip neste instruksjon hvis bit b i IO-register A er 1.

Eksempler:

Hopp til label ”lab1” hvis bit 6 på port B (innport) er 1:

```
…
SBIC PINB, 6; Skip hvis bit 6 er 0, utfør neste hvis 1
RJMP lab1
…
```

Hopp til label ”lab2” hvis bit 2 på R19 er 0:

```
…
SBRS R19, 2; Skip hvis bit 2 er 1, utfør neste hvis 0
RJMP lab2
…
```

6 Assemblyprogrammering

Til nå har vi ikke sett hvordan vi skal lage et komplett assemblyprogram. Vi har kun sett på enkeltinstruksjoner og små ufullstendige sekvenser av instruksjoner. Når vi skal lage et større program, må dette planlegges og konstrueres som et hvilket som helt annet teknologisk produkt, med fokus på god struktur og effektivitet.

6.1 Assemblerdirektiver

Når man skal skrive et program i assemblykode, ønsker programmereren å ha god kontroll over hvor i programminnet de enkelte programinstruksjonene blir plassert. Dessuten er det ofte ønskelig å referere til registre og plasser i dataminnet med navn og ikke nummer og adresser. Dette og mer hjelper assemblerdirektivene oss med å administrere.

Assemblerdirektiver er kommandoer i et assemblyprogram som ikke generer noen maskininstruksjoner. I stedet angir de hvordan PC-programmet (Assembleren) som tolker og oversetter kildefilen til maskinkode, skal oppføre seg. Merk at alle direktiver må innledes med et **punktum** (**.**).

Om bruk av tegnsetting i dette kapittelet:

<tekst> En tekst inne i to vinkelparenteser på denne måten angir at en tallkonstant skal settes inn som erstatning for <tekst> (inklusive vinklene). Betydningen av tallet beskrives av teksten.

6.1.1 CSEG – Start Kodesegment

Code / kode er en betegnelse på programinstruksjoner som plasseres (er plassert) i programminnet (FLASH). CSEG-direktivet forteller assembleren at et nytt område med programinstruksjoner skal begynne. Vi kan ha flere kodesegmenter i samme assemblerfil, men de blir slått samme til et felles område under assembleringsprosessen.

Kodesegmentene er tilknyttet en ***lokasjonsteller*** som økes med 1 for hvert instruksjonsord (16 bit) som legges til i programmet. (Noen instruksjoner er 2-ords instruksjoner og da økes telleren med 2). ORG-direktivet kan benyttes til å styre *lokasjonstelleren* direkte.

Hvis vi ikke spesifiserer noe annet antar assembleren at vi som standard har et kodesegment.

Lokasjonstelleren starter som standard på 0 for første kodesegment. For de andre kodesegmentene starter lokasjonstelleren som standard direkte etter siste instruksjon fra forrige kodesegment.

Syntaks:

```
.CSEG
```

Eksempel:

```
.CSEG               ; Start code segment
mov r1, r0          ; On Flash address 0
lds r5, 0x40        ; On Flash address 1
```

6.1.2 DSEG – Start Datasegment

DSEG-direktivet forteller assembleren at et område i SRAM skal defineres. Assemblerfilen kan bestå av mange datasegmenter, men alle disse blir slått sammen til ett fellesområde under assembleringen.

Et datasegment består normalt av BYTE-direktiver og *labels* (etiketter). Datasegmenter har sin egen **lokasjonsteller** som angir SRAM-adressen. SRAM-adressen økes med 1 for hver *byte* som defineres inn i datasegmentet. ORG-direktivet kan benyttes til å styre *lokasjonstelleren* direkte.

Syntaks:

```
.DSEG
```

Eksempel:

```
.DSEG               ; Start data segment
var1:  .BYTE 1      ; reserver 1 byte til var1
table: .BYTE 10     ; reserver 10 bytes.
```

6.1.3 ORG - Sett lokasjonsteller

ORG-direktivet setter lokasjonstelleren for gjeldene segment (nærmeste foranliggende CSEG/DSEG) til en ny verdi. Merk at lokasjonstelleren teller i *bytes* (8 bit) for datasegmenter og i *words* (16 bit) for kodesegmenter.

Standardverdien (default) for lokasjonstellerne er 0 for kodesegment (FLASH) og 0x60 for datasegment (= adressen til 1. plass i SRAM).

Syntaks:

```
.ORG <ny verdi på lokasjonsteller>
```

Eksempel:

```
.DSEG                ; Start data segment
.ORG 0x80            ; Set SRAM address to hex 80
variable: .BYTE 1    ; Reserve a byte at SRAM adr.80H

.CSEG
.ORG 0x10            ; Set Program (location) Counter to hex 10
mov r0,r1            ; Do something
STS variable, r0     ; Store r0 at M[0x80]
```

6.1.4 BYTE – Reserver bytes for en variabel

BYTE-direktivet reserverer plass i dataminnet SRAM. BYTE-direktivet bør alltid benyttes sammen med en *label* (etikett) for at vi skal kunne referere til den reserverte plassen.

Plasser i dataminnet som reserveres på denne måten betegnes **variable**. Dette er altså plasser som programmet både kan lese og skrive til.

BYTE-direktivet kan bare benyttes inne i et datasegment, dvs. at DSEG må være nærmeste foranliggende segmentdirektiv. (Se DSEG.)

Syntaks:

```
<LABEL>: .BYTE <antall plasser/bytes som skal reserveres>
```

Eksempel:

```
.DSEG
var1:    .BYTE 1   ; reserve 1 byte to var1
table:   .BYTE 5   ; reserve 5 bytes

.CSEG
ldi r20, var1      ; Last adressen VAR1
lds r1, var1       ; Last innholdet til VAR1
sts var1, r10      ; Skriv r10 til (innholdet av) VAR1
```

Obs! Merk at når en variable definert ved BYTE-direktivet benyttes i programmet, kan navnet i seg selv enten bety **adressen** til variabelen (f. eks. ved LDI) eller **innholdet** av variabelen (f. eks. ved LDS / STS). Instruksjonene selv bestemmer altså hvordan operandene skal tolkes.

6.1.5 DEF - Sett et symbolsk navn på et register

DEF-direktivet lar programmereren gi navn til registre som benyttes i samme rolle gjennom programmet. Bruker du f. eks. R20 som en teller, kan det øke lesbarheten av programmet hvis du kan referere til navnet *teller* istedenfor R20. Samme registeret kan defineres med flere navn i samme programmet.

Syntax:

```
.DEF <Symbol>=<Register>
```

Example:

```
.DEF temp=R16
.DEF ior=R0

.CSEG
ldi temp,0xf0  ; Load 0xf0 into temp register
in ior, PINB   ; Read SREG into ior register
or temp,ior    ; Or temp and ior
```

6.1.6 INCLUDE – Inkluder en annen fil i programmet

INCLUDE-direktivet befaler assembleren å starte å lese en tekst fra en annen spesifisert fil som om direktiver og instruksjoner som leses skulle komme fra den opprinnelige filen. Den nye filen lese ferdig. Deretter fortsetter assembleren å lese filen den startet med.

Syntaks:

```
.INCLUDE "<filnavn>"
```

Example:

```
.INCLUDE iodefs.asm    ; Include I/O definitions
```

6.1.7 DB – Legg inn byte-konstantverdier i programminnet

DB direktivet reserverer minneressurser i programminnet. For å kunne referere til de reserverte minneområdene, må et DB-direktiv merkes med en label. I DB-direktivet etterfølges av en liste med uttrykk med minst et element i listen. DB-direktivet må plasseres i et kodesegment.

Listen med uttrykk er en sekvens av uttrykk atskilt med komma. Hvert uttrykk må representere en konstant mellom –128 og +255. Hvis et negativt tall benyttes vil en verdi i 8-bits 2's komplement kode settes inn.

Elementene i uttrykkslista plasseres inn i kodesegmentet som konstantverdier (ikke instruksjoner), men programmereren må selv sørge for at konstantene som er lagt inn i programminnet på denne måten, ikke forsøkes utført av MCU som instruksjoner.

Hvis flere enn en konstant legges inn i programminnet på denne måten, blir konstantene pakket sammen på en slik måte, at to påfølgende bytes plasseres i samme ord i programminnet. Hvis listen består av et ulike antall elementer, blir siste kontant lagt i et ord for seg selv, selv om også neste ord i assebly-programmet er DB-direktiv. Den ubrukte halvdelen av ordet settes til 0. Assembleren vil gi en advarsel om dette.

Syntax:

```
<LABEL>:  .DB <uttrykksliste>
```

Example:

```
.CSEG
konstanter: .DB 0, 255, 0x55, -128, 0xaa
```

6.1.8 DW – Legg inn word-konstantverdier i programminnet

DW-direktivet tilsvarer DB-direktivet med den forskjellen at her reserveres hele ord (à 16 bit).

Listen med uttrykk er en sekvens av uttrykk atskilt med komma. Hvert uttrykk må representere en konstant mellom –32768 og +65535. Hvis et negativt tall benyttes vil en verdi i 16-bits 2's komplement kode settes inn.

Syntax:

```
<LABEL>: .DW <uttrykksliste>
```

Example:

```
.CSEG
varlist:  .DW 0, 0xffff, -32768, 65535
```

6.1.9 EQU - Sett et symbol lik et uttrykk

EQU-direktivet tilordner en verdi til en label. Denne label kan deretter benyttes i et senere uttrykk som en symbolsk konstant istedenfor en numerisk konstant. En label som er tilordnet en verdi vha et EQU-direktiv, kan ikke forandres senere i programmet.

Det er ikke nødvendig at EQU-direktivet er tilknyttet noe kode- eller datasegment.

Syntax:

```
.EQU <label> = <uttrykk>
```

Example:

```
.EQU io_offset = 0x16
.EQU portb     = io_offset + 2

.CSEG          ; Start kodesegment
clr r2         ; Clear register 2
out portb,r2   ; Skriv til PORTB
```

6.1.10 SET - Tilordne et symbol til et uttrykk

Hovedforskjellen på SET-direktivet og EQU-direktivet er at labeler tilordnet med SET kan forandres senere i programmet ved et nytt SET-direktiv.

SET-direktivet tilordner en verdi til en label. Denne label kan deretter benyttes i et senere uttrykk som en symbolsk konstant istedenfor en numerisk konstant. Det er ikke nødvendig at SET-direktivet er tilknyttet noe kode- eller datasegment.

Syntax:

```
.SET <label> = <uttrykk>
```

Example:

```
.SET io_base = 0x16
.SET offset  = 2
.SET portb   = io_base + offset

.CSEG          ; Start kodesegment
clr r2         ; Clear register 2
out portb, r2 ; Skriv til PORTB

.SET io_base = 0x10
.SET offset  = 0
.SET pind    = io_base + offset

.CSEG          ; Start kodesegment
in r1, pind    ; Les fra PIND
```

6.2 Struktur for assemblyprogram

I dette avsnittet skal det vises hvordan strukturen for et typisk assembly-program ser ut. Det finnes selvfølgelig mage variasjoner, men den viste strukturen er typisk.

```
; Hent inn standarddefinisjoner for bl. a. IO-PORTER
.include ”2313def.inc”

; Deklarer symbolske registernavn og konstanter
.def r22 = temp;
.equ konstant = 25

; Begynnelse programinstuksjoner
.CSEG
.ORG 0        ; 1. plass i FLASH
RJMP START: ; 1. instruksjon er sopm regel en hoppinstruksjon

; Hopp over eventuelle avbruddsvektorer
.0RG 11       ; Neste (helt) ledige  plass i FLASH

START:
; Sett opp STACK-peker og PORT-konfigurering
LDI temp, 0xDF ; Siste plass i SRAM
OUT SPL, temp  ; Stackpeker

LDI temp, ……   ; Konfigurasjon for B
OUT DDRB, temp

LDI temp, ……   ; Konfigurasjon for D
OUT DDRD, temp

HOVED:
; Hovedløkke i program
---
---
RCALL SUB1
---
---
RCALL SUB2
---
---
RJMP HOVED

; Definisjon av subrutiner
SUB1:
---
---
RET

SUB2:
---
---
RET

.DSEG
; Definisjon av variable
---
---
---
---
```

6.3 Standard kontrollstrukturer

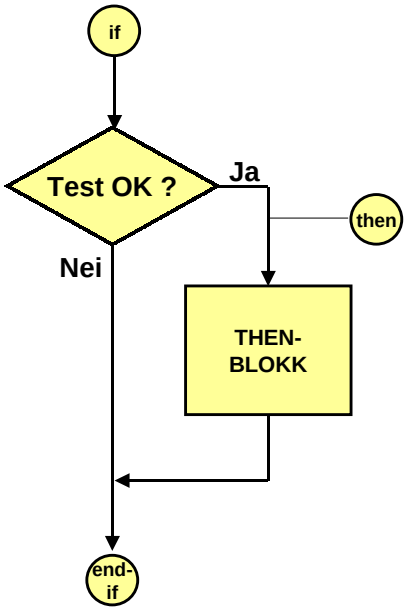
Assemblyprogrammering gir full frihet til å utnytte instruksjonssettet til en gitt mikrokontroller. For å beskrive et assemblyprogram bør man alltid først tegne opp et flytskjema. Dette flytskjemaet kan man benytte som ledesnor når man skriver koden for programmet.

Imidlertid byr flytskjemateknikken sammen med det tilgjengelige instruksjons-settet så mange muligheter, at totalprogrammet kan få en svært komplisert struktur. Det er derfor god programmeringsskikk å begrense mulighetene man har med flytskjemateknikken ved å begrense seg til bruk av et sett standard kontrollstrukturer. Ved hjelp av dette settet kan så de aller fleste programmer konstrueres.

I dette avsnittet beskrives følgende kontrollstrukturer sammen med eksempler på assemblyinstruksjoner som realiserer strukturene:

- IF-THEN
- IF-THEN-ELSE
- WHILE
- DO-WHILE

6.3.1 IF-THEN strukturen



I denne strukturen foretas det den test. Avhengig av resultatet av denne testen skal programkontrollen følge en av to mulige veier.

Programflyten er forutsatt å starte i punktet *if*. Ved positivt svar på testen, følger programmet den såkalte *then-grenen* for deretter å ende opp ved punktet *end_if*. Ved negativt svar på testen går programmet direkte til punktet *end_if* uten at noe utføres på veien.

Blokken i the-grenen kan bestå av enkle sekvensielle flytskjemaelementer, eller de kan bestå av kombinasjoner av tilgjengelige flytskjemastrukturer inklusive subrutinekall.

Denne strukturen kan oppfattes som et spesialtilfelle av IF-THEN-ELSE strukturen, men det er vanlig å oppfatte IF-THEN som en egen type. Programmeringsmessig er det mest effektivt å snu testene i forhold til IF-THEN-ELSE testene.

PROGRAMMERINGSMAL

Alternativ 1 - Bruk av branch-instruksjoner

```
if_x:
     <generer flagg ut fra behov>
     <branch hvis test ikke OK> end_if_x
then_x:
     --- ---
     --- ---
end_if_x:
```

Flaggene som benyttes i testen er f. eks. Z-flagg, C-flagg, N-flagg som angir resultatet av en aritmetisk / logisk instruksjon f. eks. cpi. I praksis kan det av og til være mer hensiktsmessig å snu/invertere testen, slik at then-grenen og den direkte grenen bytter plass i programmet.

Eksempel 1

Hvis R10 >= R9 (regnet uten fortegn) adder R15 og R8

```
if1:
     cp R10, R9          ; Er R10 >= R9 ?
     brlo end_if1        ; C=1? Hvis nei – hopp til end_if.
then1:
     out PORTB, R10
     rjmp end_if1
end_if1:
```

Alternativ 2 - Bruk av skip-instruksjoner

Denne varianten kan benyttes ved testing om et enkelt bit på en PORT har verdien B.

```
if_x:
     <skip if bit lik B>
     rjmp end_if_x
then_x:
     --- ---
     --- ---
end_if_x:
```

Eksempel 2

Utfør R16 ← R16 + 1 hvis bit 2 på PORTB er '1'.

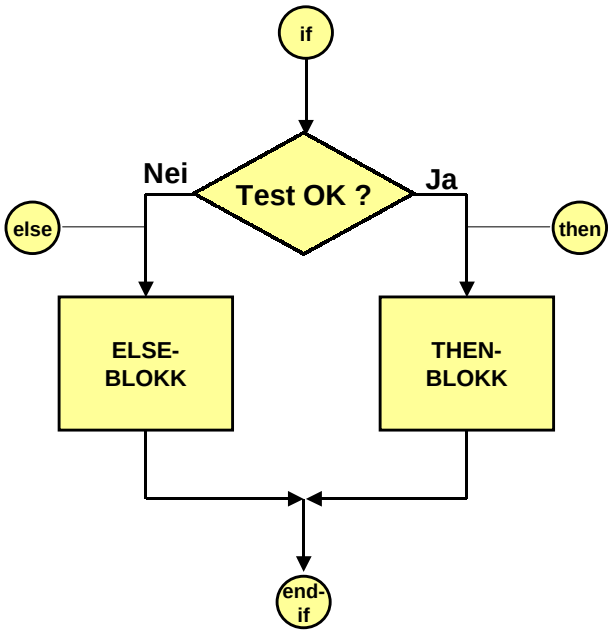
```
if2:
     sbis PINB, 3   ; Er 'bit 2' = '1', så hopp til then2.
     rjmp end_if2   ; 'Bit 2' = '0' :hopp til end_if2.
then2:
     inc R16
end_if2:
```

6.3.2 IF-THEN_ELSE strukturen

I denne strukturen foretas det den test. Avhengig av resultatet av denne testen skal programkontrollen følge en av to mulige veier.

Programflyten er forutsatt å starte i punktet *if*. Ved positivt svar på testen, følger programmet den såkalte *then-grenen*. Ved negativt svar på testen følges *else-grenen*. Begge disse grenene møtes i et felles punkt som er betegnet *end_if* i diagrammet.

Blokkene i de to grenene kan bestå av enkle sekvensielle flytskjemaelementer, eller de kan bestå av kombinasjoner av



tilgjengelige flytskjemastrukturer inklusive subrutinekall.

PROGRAMMERINGSMAL

Alternativ 1 - Bruk av branch-instruksjoner

```
if_x:
     <generer flagg ut fra behov>
     <branch hvis test OK> then_x
else_x:
     --- ---
     --- ---
     rjmp end_if_x
then_x:
     --- ---
     --- ---
end_if_x:
```

Flaggene som benyttes i testen er f. eks. Z-flagg, C-flagg, N-flagg som angir resultatet av en aritmetisk / logisk instruksjon f. eks. cpi. I praksis kan det være mer hensiktsmessig å snu testen, slik at then- og else-grenene bytter plass i programmet.

Eksempel 1

```
if1:
     cpi R20, 7     ; Er R16 = 7 ?
     breq then1     ; Z=0? Hvis ja – hopp til then1.
else1:
     out PORTB, R10
     rjmp end_if1
then1:
     ldi R16, 0x0F
     out PORTB, R16
end_if1:
```

Alternativ 2 - Bruk av skip-instruksjoner

Denne varianten kan benyttes ved testing om et enkelt bit på en PORT har verdien B.

```
if_x:
     <skip if bit = B>
     rjmp else_x
then_x:
     --- ---
     --- ---
     rjmp end_if_x
else_x:
     --- ---
     --- ---
end_if_x:
```

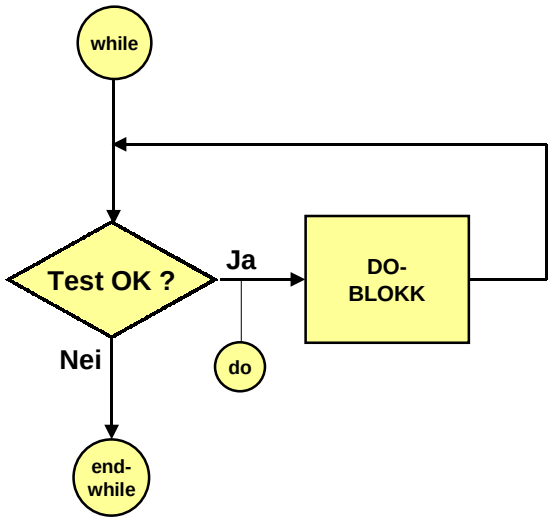
Eksempel 2

Utfør R20 ← R20 – R19 hvis bit 3 på PORTD er 1, hvis ikke – utfør R20 ← R20 + R19

```
if2:
     sbis PIND, 3   ; Er 'bit 3' = '1', så hopp til then2.
     rjmp else2     ; 'Bit 3' = '0': hopp til else2.
```

```
then2:
     sub R20, R19
     rjmp end_if2
else2:
     add R20, R19
end_if2:
```

6.3.3 WHILE LØKKER



En WHILE-løkke er en flytskjemastruktur som utfører en programblokk et antall ganger.

WHILE-løkken avsluttes vha. en test. Feiler denne testen avsluttes løkken. Er testen sann, fortsetter programmet i løkken.

Karakteristisk for denne typen er at programblokken ikke gjennomføres noen ganger, hvis testen feiler første gang.

Antall gjennomløp av løkken n ≥ 0.

Programblokken kan bestå av en eller flere sekvensielle aksjoner eller en kombinasjon av løkke- og test-strukturer.

Avslutningstesten kan være alt fra om en teller er telt opp til en gitt verdi, et bit på en port er nullstilt eller om innholdet i 2 registre er lik hverandre – valget er programmererens.

PROGRAMMERINGSMAL

Alternativ 1 - Bruk av branch-instruksjoner – negativ test

```
while_x:
     <generer flagg ut fra behov>
     <branch hvis test ikke OK> end_while_x
do_x:
     --- ---
     --- ---
     rjmp while_x
end_while_x:
```

Flaggene som benyttes i testen er f. eks. Z-flagg, C-flagg, N-flagg som angir resultatet av en aritmetisk / logisk instruksjon f. eks. cpi.

Eksempel 1

Utfør løkken så lenge til R10 <> R11

```
while1:
     cp R10, R11
     breq end_while1    ; Hvis like – hopp til end_while.
do1:                    ; Hvis foskjellige fortsett her.
     out PORTB, R10
     in R10, PIND
     rjmp while1
end_while1:
```

Alternativ 2 - Bruk av branch-instruksjoner – positiv test

Denne varianten benytter en hopp-instruksjon mer enn forrige, men er kanskje lettere å forstå logisk siden vi kan benytte den positive testen.

```
while_x:
     <generer flagg ut fra behov>
     <branch hvis test OK> do_x
     rjmp end_while_x
do_x:
     --- ---
     --- ---
     rjmp while_x
end_while_x:
```

Eksempel 2

Utfør løkken så lenge til R10 <> R11

```
while2:
     cp R10, R11
     brne do2:     ; Hvis forskjellige – hopp til do-blokk.
     rjmp end_while2    ; Hvis like - avslutt
do2:
     out PORTB, R10
     in R10, PIND
     rjmp while2
end_while2:
```

Alternativ 3 - Bruk av skip-instruksjoner

Denne varianten kan benyttes ved testing om et enkelt bit på en PORT eller et register har verdien B. Utfører løkken så lenge (while) bit = B.

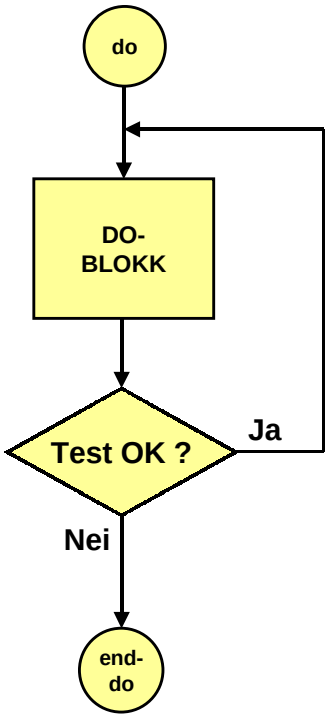
```
while_x:
     <skip if bit = B>
     rjmp end_while
do_x:
     --- ---
     --- ---
     rjmp while_x
end_while_x:
```

Eksempel 3

Utfør løkken så lenge til PORTD.5 = 1

```
while3:
     sbis PIND, 5
     rjmp end_while3    ; Hvis PORTD.5 = 0 - avslutt
do3:
     out PORTB, R10
     rjmp while3
end_while3:
```

6.3.4 DO-WHILE LØKKER



En DO-WHILE-løkke er en flytskjemastruktur som utfører en programblokk et antall ganger.

DO-WHILE-løkken avsluttes vha. en test. Feiler denne testen avsluttes løkken. Er testen sann, fortsetter programmet i løkken.

Karakteristisk for denne typen er at programblokken gjennomføres minst en gang, selv om testen feiler første gang.

Antall gjennomløp av løkken n ≥ 1.

Programblokken kan bestå av en eller flere sekvensielle aksjoner eller en kombinasjon av løkke- og test-strukturer.

Avslutningstesten kan være alt fra om en teller er telt opp til en gitt verdi, et bit på en port er nullstilt eller om innholdet i 2 registre er lik hverandre – valget er programmererens.

PROGRAMMERINGSMAL

Alternativ 1 - Bruk av branch-instruksjoner

```
do_x:
     --- ---
     --- ---
     <generer flagg ut fra behov>
     <branch hvis test OK> do_x
end_do_x:
```

Flaggene som benyttes i testen er f. eks. Z-flagg, C-flagg, N-flagg som angir resultatet av en aritmetisk / logisk instruksjon f. eks. cpi.

Eksempel 1

Utfør løkken så lenge til R10 <> R11

```
do1:
     out PORTB, R10
     in R10, PIND
     cp R10, R11
     brne do1  ; Hvis test OK – hopp til start
end_do1:
```

Alternativ 2 - Bruk av skip-instruksjoner

Denne varianten kan benyttes ved testing om et enkelt bit på en PORT eller et register har verdien B. Utfører altså løkken så lenge bit = B.

```
do_x:
     --- ---
     --- ---
     <skip if bit = B>
     rjmp end_do_x
     rjmp do_x
end_do_x:
```

Eksempel 3

```
Utfør løkken så lenge PORTD.5 = 1
```

```
do2:
     out PORTB, R10
     sbis PIND, 5
     rjmp end_do2  ; Hvis PORTD.5 = 0 - avslutt
     rjmp do2
end_do2:
```

Merk at denne varianten kan gjøres noe enklere med bruk av en negativ test

6.4 Subrutiner

Subrutiner er den viktigste mekanismen for å håndtere store programmer. Selv om man strukturerer programsystemet samvittighetsfullt ved hjelp av flytskjema, kommer man før eller senere til et punkt der programmene blir så store at de blir uoversiktlige. Det blir umulig å studere programmet og forstå virkemåten fullt ut.

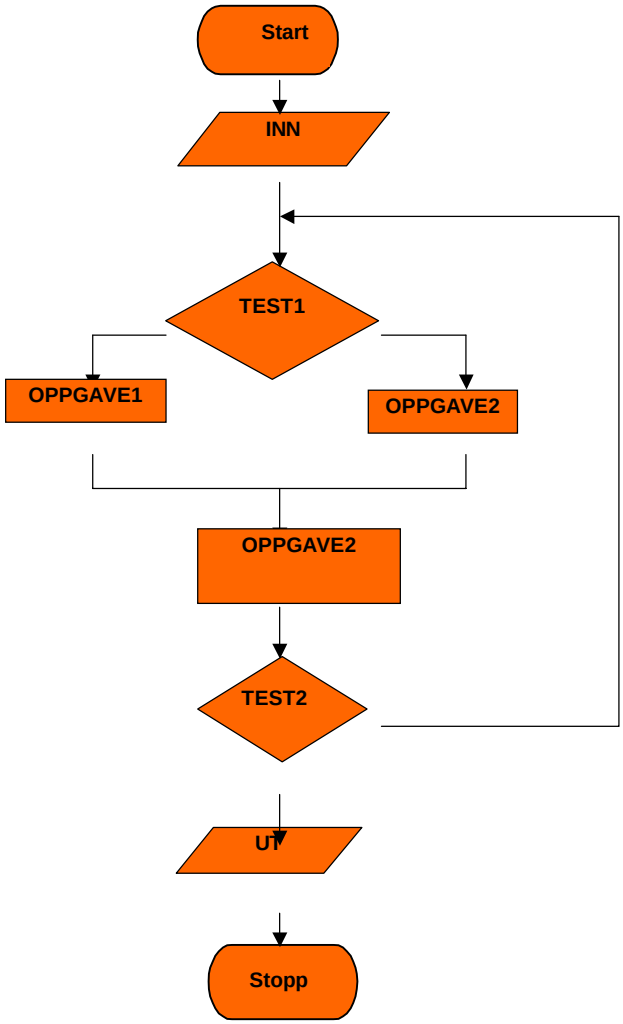
Botemiddelet her er å dele opp programmet i mindre enheter som hver for seg kan struktureres på en oversiklig måte. Det er her subrutinene kommer inn; en subrutine er ikke noe annet en et lite selvstendig program som kan kalles opp fra andre steder i totalprogrammet.

Den delen av programmet som ikke er en subrutine, men som kaller opp og benytter subrutiner, kalles ofte **hovedprogrammet**.

Subrutinene er viktige også av en annen årsak; de gjør det mulig å benytte de samme instruksjonene om og om igjen i det samme programmet uten å måtte duplisere koden.

Tenk at du skal skrive et program som utfører diverse geometriske beregninger. En algoritme som beregner kvadratroten kan utvikles i assemblykode. La oss si at det går med 20 assemblyinstruksjoner til dette. Hvis denne algoritmen trengs 10 ganger i programmet, er det dumt å måtte gjenta de 20 kodelinjene 10 ganger, slik at det totalt går med 200 assemblyinstruksjoner. Det er sløsing med plass og det blir også vanskelig å vedlikeholde programmet. Anta at du etter en stund oppdager at algoritmen må forandres litt. Det blir da nødvendig å forandre alle 10 stedene i programmet hvor algoritmen benyttes.

Løsningen er å konstruere en subrutine som utfører algoritmen og plassere denne algoritmen ett sted i programmet. Algoritmen kan så kalles opp fra de 10 stedene i programmet hvor den benyttes. Dette sparer plass, gir enklere vedlikehold, og gir ikke minst en mye bedre programstruktur.


Start
INN
TEST1
OPPGAVE1
OPPGAVE2
OPPGAVE2
TEST2
UT
Stopp