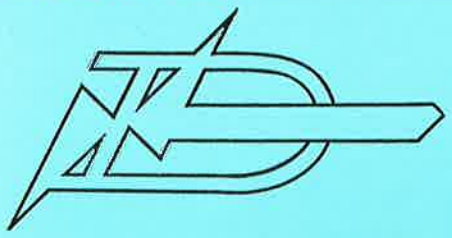
NORD

COMPUTER SYSTEMS

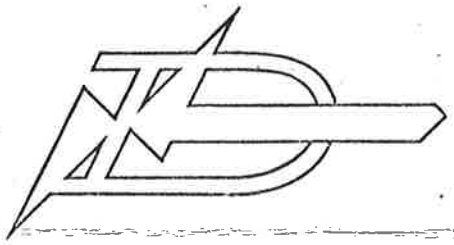
Introduksjon til
datamaskiner og programmering i
maskinkode

av

Lars Monrad-Krohn



A/S NORSK DATA-ELEKTRONIKK
Økernveien 145, Oslo 5



A/S NORSK DATA-ELEKTRONIKK
Økernveien 145, Oslo 5 (Økern)

Introduksjon til datamaskiner og programmering i maskinkode

av

Lars Monrad-Krohn

R E S Y M É

Denne introduksjon er beregnet på å leses av interesserte som ikke tidligere har hatt noen befatning med datamaskiner. De generelle prinsipper for datamaskinens virkemåte og programmering gjennomgås og belyses ved bruk av et eksempel, datamaskinen NORD-1, der dette er hensiktsmessig. For den mere detaljerte informasjon henvises til håndboken for NORD-1.

Nordlysobservatoriet i Tromsø
Tromsø

19. oktober 1967

I N N H O L D

1. REGNEMASKINKONSEPTET

1.1 Algoritme og program

Det mest naturlige eksempel å bruke for illustrasjon av en data-regnemaskins virkemåte, er utførelsen av en ordinær beregning. De moderne datamaskiner er på ingen måte begrenset til numeriske beregninger, men dette var det første maskinene ble utnyttet til fordi algoritmene for numerisk regning har vært utviklet over lengre tid.

Det viktigste ved en regnemaskin er algoritmen. Algoritme er det formalistiske sett med handlinger som fører til det ønskede resultat. Før noe som helst kan gjøres må en ha handlingsmønsteret klart definert.

Ved utførelsen av en ordinær beregning blir de nødvendige handlinger beskrevet v.h.a. sterkt komprimerte formler, samt et kjennskap til hvordan formlene skal tolkes.

De nødvendige handlinger som utføres ved en manuell beregning gjøres en av gangen i en sekvens.

Sekvensen av enkelthandlinger vil vi kalle et PROGRAM og det sees da naturlig at programmet utfører en algoritme.
- Uten algoritme intet program. -
Uten at man har nøyaktig bestemmelse av det nødvendige handlingsmønster, kan man ikke sette opp noen sekvens av handlinger. De forskjellige enkelthandlinger i en sekvens kan være enten aktive utførende handlinger eller styre handlinger som avgjør hvilke andre handlinger skal utføres.

"Å fylle et vannkar med en øse vann" er en aktiv handling, mens å avgjøre om det er nødvendig "å fylle et kar med en øse vann" er en styrehandling. Uten styrehandlinger (beslutninger) kan en ikke lave særlig verdifulle programmer (handlingssekvenser).

Handlingen i et program for numeriske beregninger består naturlig av regnehandlinger og styrehandlinger. Regnehandlingene vil være alminnelige aritmetiske operasjoner som f.eks. addisjon av to tall, mens styrehandlingene kan f.eks. være avgjørelsen om en er kommet til slutten av en serie addisjoner.

De aktive handlinger som er grunnenhetene i et program kan selv være av varierende kompleksitet. For noen formål, f.eks. ved programmering av en regnemaskin der en kun har grunnoperasjonene "addisjon" og "skift tegn" vil en måtte skrive handlingssekvensene kun ved hjelp av disse meget enkle grunnhandlinger, mens det for andre formål som programmering v.h.a. subrutiner, kan være aktuelt å uttrykke handlingssekvensen v.h.a. kompliserte grunnhandlinger.

Den numeriske matematikk er slik oppbygget at den kan definere alle kompliserte operasjoner ut fra de enkle.

Handlingen å kvadrere et helt positivt tall større enn 0 kan derfor beskrives ut fra addisjonshandlinger med passende styrehandlinger omtrent som følger:

Navn: ESPEN

1 Noter tallet to steder (og kall de to nye tall JENS og HANS)
2 Trekk 1 fra tallet i HANS.
3 Dersom tallet i HANS blitt lik 0 stopp, svaret er nu JENS, hvis ikke gå videre.
4 Adder det opprinnelige tall til tallet i JENS.
5 Fortsett fra punkt 2.

Denne sekvens av handlinger beskriver en algoritme for kvadrering av positive hele tall, og representerer en handlingssekvens der 1, 2 og 4 er aktive handlinger og 3 og 5 er styrehandlinger. Denne handling er velkjent og representerer et program for løsning av et problem

å finne flateinnholdet av et kvadrat som er gitt ved dets sidekant.

For f.eks. å finne flateinnholdet av et kvadrat gitt ved dets diagonal kreves en annen algoritme.

Nu er det illustrerte problem så enkelt matematisk at det er bygget inn i folkeskolens matematikkpensum. Det er derfor velkjent at denne algoritme er uttrykt i det spesielle matematiske algoritmespråk som

$$y = x^2 = x \cdot x$$

Operasjonen er definert slik at:

$$x^2 = \sum_{i=1}^{i=x} x_i \quad i \quad x_i = x$$

Ligning (1) er derfor en mere kompakt (og matematisk) formulering av den ovenfor viste handlings-sekvens.

Fundamentalt arbeider datamaskinene etter oppgitt handlingssekvenser som vårt eksempel, disse sekvenser kalles (regnemaskin-)program når de er formulert for en datamaskin.

Algoritmen er nødvendig for å skrive et program, og programmet må skrives med handlinger som er i overensstemmelse med de handlinger (operasjoner) som datamaskinen kan utføre.

(Ved å bruke en egen "oversetter" som f.eks. FORTRAN, kan en ofte formulere mange oppgaver mere kompakt, og overlate til "oversetteren" (som er et meget avansert ikke-numerisk regnemaskinprogram) å oversette de kompakte formuleringer til en korresponderende handlingssekvens).

Oppgave: Finn de grunnoperasjoner som er forutsatt i den ovenstående handlingssekvens kalt ESPEN.

FACIT: Aktive handlinger:

- Notere tallet er en inputoperasjon
- Trekke 1 fra tallet er en operasjon som kan tenkes enten å være en grunnoperasjon eller sammensatt av følgende 3 grunninstruksjoner:

  1. Skaff konstanten 1.
  2. Skift tegn
  3. Adder til tallet

- Adder tallet til et annet
- Svaret er nu der eller der" er en output-operasjon
- Fortsett fra et eller annet sted"
- Stopp (Gå videre er alltid underforstått).
- Dersom tallet der eller der er blitt lik 0 gjør så eller så.

Handlingssekvens inklusive styrehandlinger utføres i stor utstrekning av mennesker for løsning av en oppgave (ikke "problem") eller utførelse av en beregning. Styrehandlingene utføres mere eller mindre ubevisst.

1.2 Regneverk

Holder vi oss til den manuelle beregning som et eksempel så vi i ovenforstående seksjon at enhver utførelse av en beregning krever en algoritme og en handlingssekvens, bestående av aktive handlinger og (ubevisste) styrehandlinger.

Ved rasjonell organisasjon av arbeidet brukes oftest et regneverk som hjelpemiddel. Dette regneverk kan være regnestav, addisjonsmaskin eller mere komplex (konvensjonell mekanisk) regnemaskin.

For de aktive handlinger engasjeres regneverket.

1.3 Mellomresultater

Med manuelle beregninger er det oftest nødvendig å notere ned mellomresultater. Dette representerer en lagring av informasjon. Som regel er det ubevisst hvor på papirarket et menneske noterer ned sitt mellomresultat, men det er helt essensielt at mellomresultatet finnes igjen når det senere trengs. Det kunne derfor vært effektivt å gi mellomresultatet et midlertidig navn eller notere tallet på en linje med nummer. Dette gjør det bl.a. også lettere å beskrive handlingssekvensen.

1.4 Data

Holder vi oss til manuelle beregninger er det nødvendig å presentere for beregneren de tall ut fra hvilke han skal arbeide. Disse kalles data og bringes til beregneren i en prosess som kan kalles input-prosessen. Mellomresultatene kan også sees på som data, på samme måte som resultattallene fra en beregning.

Prosessen med å bringe ut resultattallene kan kalles en outputprosess. Beregneren må ha data lagret et eller annet sted, oftest på papir. Papiret sies da å representere en hukommelse (et lager) for data.

1.5 Lagret program

Ved den meget enkle manuelle beregning vil som regel beregneren kunne huske de forskjellige handlingers rekkefølge i sekvensen.

Er beregningen meget komplisert kan det være nødvendig at beregneren har handlingssekvensen notert på papir foran seg. Handlingssekvensen kan da sies å være lagret i hukommelsen.

Alle linjene på papirarket kan være nummerert, og beregneren kan for å holde rede på hvor han er i handlingssekvensen,

alltid holde linjenummeret for den aktuelle handling notert på et bestemt sted (P).

Vi betrakter papiret foran beregneren som den generelle hukommelse. På dette papiret har alle linjene fått sitt nummer, f.eks. fra 0 til 4096. På dette papiret står handlingssekvensen notert ned, i tillegg kommer også de utgangsdata beregneren har fått, samt mellomresultater (som ikke blir stående og brukt videre i regneverket) og resultatdata som beregneren skriver ned.

Når beregneren utfører de enkelte handlinger utfra listen, er det viktig at hans noterte linjenummer for hver handling er riktig. Dersom beregneren på papiret foran forvekslet handlinger med data ville det blir rare resultater (hvis noen i det hele tatt).

En menneskelig beregner vil gjerne kunne se forskjell på data og handlinger, men det kunne jo hende at handlingene var presentert i en tallkode.

2 REGNEMASKINSTRUKTUREN I PRAKTISK UTFØRELSE

2.1 Lagring og utførelse av program

Datamaskiner kan klassifiseres i to typer, med eller uten selvmodifiserbart program. Den siste typen er spesielle maskiner med "wired-in" program. Den første typen omfatter "general-purpose" maskinene, og det er disse vi skal befatte oss videre med.

Det er først og fremst mulighetene for lagring av forskjellige programmer, og maskinens evne til selv å modifisere disse som er den fundamentale egenskap ved denne klasse datamaskiner.

Programmet lagres i hukommelsen i form av enkelte instruksjoner som hver styrer en handling i maskinen. På samme måte som de enkelte handlinger i en algoritme må utføres i sekvens, så vil instruksjonene bli utført i sekvensiell rekkefølge.

Og på samme måte som en kan tenke seg et papir med nummererte linjer, vil maskinens hukommelse være oppdelt i linjer, hver med sitt nummer. En linje i maskinens hukommelse kalles et ord og linjenummeret kalles for linjen eller ordets adresse. (Address, location).

De enkelte instruksjoner i et program lagres i etterfølgende adresser i hukommelsen.

Det er da fundamentalt å vite hvor i hukommelsen et program starter, (i hvilken adresse programmet starter), og det er nødvendig ved enhver oppstarting av maskinen eksplisitt å fortelle maskinen hvor (i hvilken adresse) det aktuelle program starter.

Når maskinen utfører et program skjer det ved at den leser ut en og en instruksjon av gangen og utfører disse etter hver utlesning.

Maskinen holder notert på en særlig plass kalt P, (Program counter), det nummer (den adresse) som den til enhver tid aktuelle instruksjon har i hukommelsen. Videre holder maskinen til enhver tid rede på hvilken instruksjon den leste ut av hukommelsen og som den er i ferd med å utføre. Denne selve instruksjonen holder maskinen notert på en plass som kalles IR (Instruction Register).

Det som skjer under utføringen av en instruksjon er følgende:

1 P konsulteres for å finne hvor i hukommelsen instruksjonen skal leses ut ifra.

2 Maskinen leser ut til IR instruksjonen fra den adresse i hukommelsen som er spesifisert av P.

3 Maskinen utfører instruksjonen.

4 Det adderes 1 til innholdet av P for derved å gjøre klar til neste instruksjon.

Denne sekvens er den som gjelder for de instruksjoner som beskriver aktive handlinger. Dreier det seg derimot om instruksjoner som beskriver styrehandlinger blir punkt 4 skiftet ut med en annen bestemmelse av det nye innhold av P.

De instruksjoner som beskriver styrehandlinger kalles kontroll-instruksjoner (control instructions) fordi de griper inn i selve sekvensen av instruksjoner ved å bestemme at den neste instruksjon blir noe annet enn den som er spesifisert ved innholdet av adresse (P + 1). Disse instruksjoner kalles hopp- eller skip-instruksjoner. Hopp-instruksjonene overfører "kontrollen" til helt andre steder i programmet, mens skip instruksjonene hopper over bare en instruksjon.

Hopp- og skip-instruksjonene er enten betinget eller ubetinget. Ved den ubetingede hopp-instruksjon utføres endringen i sekvens uansett de interne eller eksterne

forhold i maskinen. Ved de betingede hopp- eller skip-instruksjoner utføres hoppet eller skippet kun dersom den spesifiserte betingelse er oppfylt. Det finnes derfor mange forskjellige av disse instruksjoner for de forskjellige forhold som det kan være hansiktsmessig å kunne avprøve både internt og eksternt. Jfr. punkt 3 i den algoritme som ble gitt i forrige kapitel.

De betingede kontrollinstruksjoner representerer en måte for programmet å "ta inn" (enkel binær) informasjon fra utenverdenen.

2.2 Registre og aritmetikk

De plasser som maskinen bruker til å notere ned forskjellige ting kalles registre. Dette er også en form for hukommelse, bare at registrene brukes mere spesielt og er gjerne lavet elektronisk på en annen måte enn hovedhukommelsen, slik at "noteringene" og "avlesningene" kan skje hurtigere enn inn og utlesning fra hovedhukommelsen. (Hovedhukommelsen er også ofte kalt hurtighukommelsen).

Vi har allerede nevnt to nødvendige registre for maskinen, P og IR for instruksjonsadresse og instruksjon. Antall registre for forskjellige maskiner kan variere, men de fleste har også følgende:

- H (Memory buffer register). Dette register brukes til å ta imot alt som leses ut fra, og til det som skal settes inn i hukommelsen. Dette brukes av programmereren kun til manuelt å undersøke hva som står i hukommelsen.

- R (Memory address register). Dette register spesifiserer for hukommelsen hvilken adresse som det skal leses ut fra eller skrives inn i. Dette register brukes ikke av programmereren.

- A (Accumulator). Dette register er den fundamentale noteringsplass i maskinen. Både aritmetiske og input-output instruksjoner bruker dette register.

Ved siden av disse registre hvorav kun ett (A) er "bevisst" for programmet kan det finnes flere både "bevisste" og "ubevisste" registre i maskinen.

Noen maskiner har et tilleggsregister til A (D) og noen har et såkalt indeksregister(X).

De fleste maskiner har i tillegg til registre et regneverk som kan arbeide sammen med ett eller flere registre. Regneverket kan være av varierende kompleksitet, fra regneverk kun for addisjon til regneverk for "flytende" aritmetikk. (Regning med flytende komma).

2.3 Instruksjoner

Instruksjonene står lagret i hukommelsen og blir overført til IR-registeret når de skal utføres. Instruksjonene er lagret i hukommelsen i form av tallkoder (binær- eller oktal-tall).

Vi skal ikke komme inn på de enkelte tallkoder, men bruke symboler istedenfor. Symbolene er da forsøksvis valgt slik at symbolene for de forskjellige kodene er lette å huske.

Utvalget av tilgjengelige instruksjoner er meget avgjørende for den effektivitet en kan oppnå i et program, både med hensyn til antall instruksjoner det er nødvendig å bruke, men hensyn til letthet og enkelthet i formuleringen, og med hensyn til hurtigheten i utførelsen av programmet.

De forskjellige maskiner har sterkt varierende utvalg av instruksjoner, men noen er felles for de fleste maskiner.

Instruksjonene kan være av forskjellige typer, som allerede nevnt har vi to hovedtyper:

A) Utførende instruksjoner for beskrivelse av aktive handlinger.

B) Kontrollinstruksjoner for styrehandlinger ved overføring av kontrollen.

De utførende instruksjoner kan igjen deles opp i flere typer:

A 1) Aritmetiske instruksjoner
A 2) Logiske instruksjoner
A 3) Input-output instruksjoner
A 4) Registeroverføringsinstruksjoner

Kontrollinstruksjonene deles opp i følgende typer:

B 1) Stopp-instruksjonen (1 stk.)
B 2) Ubetinget hopp-instruksjonen (1 stk.)
B 3) Betingede hopp-instruksjoner
B 4) Betingede skip-instruksjoner

I noen tilfeller finner en også en kombinasjon av A 3 og B 4 instruksjonene for å utprøve eksterne forhold.

Den mest typiske instruksjon i klassen A 1 er instruksjon for addisjon. Dette er nesten i alle maskiner en instruksjon for addisjon av et tall hentet fra hukommelsen til det tall som står notert i A-registeret. Denne instruksjon representeres ved symbolet:

ADD

Det er imidlertid ikke nok å spesifisere dette alene, en må også spesifisere en adresse i hukommelsen for den andre addend. Hvis en f.eks. har den andre addenden i adresse 2000 i hukommelsen blir den fullstendige instruksjon for addisjon:

ADD 2000

Nå kan det tenkes at en gjerne også bruker symboler for adressene og dersom en har definert PER = 2000, vil instruksjonen også kunne skrives

ADD PER

Generelt gjelder at instruksjonene A 1 (og A 2) består av to deler, en instruksjonsdel og en adressedel. Det tall som for dette eksempel på instruksjon virkelig står lagret i hukommelsen kan være oktaltallet:

060407

eller binært: 0110000100000111

2.4 Kommunikasjon med omverdenen

Kommunikasjon med omverdenen foregår ved hjelp av input-output instruksjoner og et sett spesielle elektriske linjer til omverdenen representert ved et eller annet perifert utstyr.

Det perifere utstyr har for hver enkelt installasjon fått tildelt seg et nummer slik at ikke to utstyr i en innstallasjon har samme nummer.

Alle perifere utstyr må være slik konstruert at de hele tiden "lytter" på de elektriske linjer fra maskinen. Det de lytter etter er sitt eget kallesignal d.v.s. sitt nummer og eventuelle kommandosignaler. Utstyrsnummeret (device number) og kommandoene blir sendt ut på parallelle linjer i binær (oktal) tallkode. Disse linjer kan kalles kontrollinjer.

Videre har det eksterne utstyr anledning til på andre linjer å sende spesielle kontrollsignaler til maskinen, enten som svar på kommandoer eller for å be om "service" v.h.a. såkalt "interrupt" signaler.

La oss som eksempel tenke oss at skriverenhet A har device nummer 03. Hvis man ønsker at programmet skal finne ut om denne skriverenhet er opptatt med utskrift vil en i programmet ha instruksjonen:

IOT SKA 03

Her betyr IOT at dette er en input-output-instruksjon, som fører til at de videre skrevne koder blir sendt ut på input-output kontrollinjene, og SKA betyr at en kommando "SKA" blir sendt ut på alle linjer. Device nummer 03 er "kallesignalet" som når det blir sendt ut på de parallelle linjer bare blir oppfattet av skriverenhet A.

Det kan i instruksjonen også være aktuelt å bruke et symbol for koden 03, f.eks. PNT (for print), slik at instruksjonen blir:

IOT SKA PNT

Når maskinen utfører ordren IOT vil den sende en kode ut på kontrollinjene. Denne kode er bestemt av instruksjonens øvrige deler. Koden for PNT (03) vil identifisere skriverenhet A, koden for SKA er en kommando som betyr at enheten skal sende et signal (skip-signal) tilbake til maskinen dersom den er ledig. Dette signal skal sendes på linjen for "skip". Når maskinen mottar dette skip-signal vil den skippe neste instruksjon. Derved kan programmet "bli klar over" om enheten var ledig.

2.5 Input - output gjennom A

I det foregående ble nevnt hvordan programmet kan gi kommandoer til de perifere enheter, og få en enkel ledig informasjon tilbake.

Når det gjelder overføring av data må det flere mekanismer til, nemlig et sett med output-linjer (output-kabel) og et sett med inputlinjer (input-kabel). Dette er linjer for parallell overføring av binære tall (f.eks. 16 bits i parallell). Outputlinjene er direkte koblet til A-registeret og viser til enhver tid innholdet i dette register. Inputlinjene kan under en IOT-instruksjon kobles som input til A-registeret dersom den perifere enhet signalerer på en spesiell linje (f.eks. Data Ready-linje). Den perifere enhet kan kommanderes til å gjøre dette. Vi må nå se på input-output linjene i sammenheng med kontroll-linjene med de overførte devicenummer og kommandokoder.

Dersom programmet ønsker å gi data ut til en perifer enhet med symbolsk kode PNT kan en i programmet skrive

IOT ACT PNT

Samtidig som en med det foregående program har sørget for at den riktige output informasjon (data) står i register A.

Når maskinen utfører instruksjonen IOT vil den sende ut på kontrollinjene devicenummeret for PNT (03) sammen med kommandoen ACT (Activate). Enheten for PNT oppfatter kommandoen "ACT" som en kommando til å avlese output-linjene fra register A (og å starte utskriften). På tilsvarende måte vil den perifere enhet for en input-enhet med koden f.eks. RKE kunne oppfatte kommandoen "ACT" som en kommando til å sende sin informasjon (data) over input-linjene til A samtidig med at den signalerer "Data Ready" til maskinen.

Denne form for input-output gjennom A kalles også for "Programmer input - output". Det kan tilkobles et stort antall enheter. (Hvilke begrensninger finnes?).

2.6 Input-output til hukommelsen

For store mengder data som input eller output kan det være effektivt å ha en kommunikasjonsmulighet direkte med hukommelsen uten å ha programmets og maskinens oppmerksomhet hele tiden. Man bruker da en data-kanal til kommunikasjonen.

Data-kanalen konkurrerer med maskinen om adgang til hukommelsen, og er som regel satt opp med første prioritet. Det betyr at om datakanalen er i ferd med å lese ut et ord fra hukommelsen, så vil hukommelsen være sperret for maskinen inntil utlesningen av ordet for datakanalen er

ferdig. Datakanalen er en separat enhet som driver sitt virke uavhengig av maskinens og dets program. Når datakanalen arbeider betyr det for maskinen at den må vente noe lengre ved hver bruk av hukommelsen. Dette kalles "Cycle-stealing".

Selv om datakanalen arbeider autonomt for overføring av større mengder data, må den allikevel styres fra maskinen. Denne styringen foregår som en programmert overføring av data fra maskinens A til til datakanalen.

For å styre data-kanalen trenger programmet også informasjon om datakanalens tilstand. Denne informasjon fåes fra data-kanalen som en programmert overføring til A. Datakanalen blir da som en perifer enhet i forhold til den programmerte input-output, men den informasjonen som overføres programmert tjener bare til å styre data-kanalens funksjon som er å overføre data direkte fra hukommelsen til et eller annet som trenger hurtig overføring av større datamengder, f.eks. en disk-file eller magnetbånd.

3 PROGRAMMERING I MASKINKODE

3.1 Instruksjoner i hukommelsen

Datamaskinen har sitt program lagret i hukommelsen. Den prosess å få et ført program inn i hukommelsen er et spørsmål som vi foreløpig utsetter, nu skal vi bare se i større detalj på det program som står i hukommelsen, og hvordan det er hensiktsmessig å skrive dette.

Instruksjonene står etter hverandre i hukommelsen i en binær tallkode. En sekvens av instruksjoner kan utgjøre et program når også startpunktet for sekvensen er kjent for den som skal bruke programmet. Sekvensen av instruksjoner står lagret i etterfølgende adresser i hukommelsen. Startpunktet for et program er gitt ved en adresse. Startpunktet for programmet kan være programmets laveste adresse.

Ved skriving av et program er det vanlig å skrive instruksjonene etter hverandre for økende adresser slik at den laveste adresse er programmets skrevne begynnelse.

Programmets skrevne begynnelse er altså ikke nødvendigvis det samme som programmets startpunkt, den adresse som maskinen må starte programmet med.

Programmet står i hukommelsen som binær kode. Det binære tall lar seg enkelt omforme til oktaltall og vi vil i det følgende regne med oktaltall. Adressene spesifiseres også enklest i oktaltall.

Et program lagret i hukommelsen kan skrives på oktalform som følger:

```
000010/ 000001
        044377
        060376
        124377
```

Her betyr det første 10, tallet foran skråstreken at programmets skrevne begynnelse er i adresse oktalt 10, og at de andre instruksjonene følger etter.

Oppgave: Finn ut fra håndboken for regnemaskin hva dette programmet gjør ved å finne betydningen av hver instruksjon.

3.2 Bruk av symboler

Det er opplagt ikke menneskevennlig å representere et program i oktalkoder som i den forrige seksjon. Det er mere forståelig å bruke symboler for sammensetning av de enkelte instruksjoner. Vi vil derfor i så stor utstrekning som praktisk operere med symboler. Symbolene sies å ha en "verdi" lik den tilsvarende oktale koden. Når symbolene skrives etter hverandre på en linje betraktes dette som en addisjon av deres verdier. Som eksempel på bruk av symboler uten nærmere forklaring setter vi:

```
10/  HANS, 1
     START, LDA HANS
     PER,   ADD HANS
            JMP PER
```

3.3 Adressering

Som nevnt tidligere vil instruksjonene i klasse A 1 og A 2 kreve en referanse til hukommelsen. Det samme er tilfelle for instruksjonene i klasse B 2 og B 3. Vi kaller disse instruksjoner for hukommelses-referanse-instruksjoner. Det kan her være verd å merke seg at det er antallet hukommelsesreferanseinstruksjoner som det er vanskeligst å oppnå i datamaskiner med kort ordlengde. Det er dette antall (og ikke det totale antall forskjellige) instruksjoner det er mest signifikant å vurdere spørsmålet "antall instruksjoner" etter.

Adressene i hukommelsen spesifiseres enstydig med et tall. Når det skal være en mulighet å adressere større hukommelser vil dette tallet være større og trenge flere sifre. Ved de moderne maskiner med kortere ordlengde kommer kravet om flere sifre i adressen i konflikt med kravet om flere sifre til spesifikasjon av et større antall instruksjoner. Det vil også være lite optimalt å representere adresser med større antall sifre enn det som er praktisk ved skrivning av typiske programbiter. D.v.s. at det ikke er optimalt å la hver instruksjon inneholde nok sifre til å adressere hele hukommelsen. En vil jo aldri skrive et program uten oppdeling for hele hukommelsen.

De forskjellige regnemaskinfabrikanter løser dette spørsmålet på forskjellige måter, men med mange felles trekk. Vi tar her for oss et eksempel.

Som instruksjon velger vi den tidligere nevnte ADD. Reglene for vårt eksempel er nu:

A) Skriver vi et tall mindre enn 177 f.eks. 55, som adresse betyr dette at instruksjonen finner sin operand 55 plasser lengre ned i programmet. D.v.s. den aktuelle adresse finnes ved å legge tallet 55 til adressen for vedkommende instruksjon. Har vi f.eks. følgende i hukommelsen

```
        10/ ADD 55
og      65/  1
```

vil tallet en bli lagt til A når instruksjonen i adresse 10 blir utført.

Denne form for adressering kalles relativ adressering, fordi den angitte adresse er relativt til det sted instruksjonen befinner seg. Til denne adressering brukes 8 bits som et binært (oktalt) tall med 2'er komplements representasjon av negative tall.

B) Vi kan også ha en adressering relativt til et spesielt register i maskinen som kalles baseregisteret, B. Instruksjonen ser da slik ut:

ADD 55 ,B

Dersom baseregisteret inneholder tallet 7010 vil den effektive adresse i dette tilfelle bli adresse 7065.

Baseregisteret som er på 16 bits kan inneholde store tall, og det er ved hjelp av dette at den fulle hukommelse kan adresseres. Før instruksjonen utføres i programmet, må programmet ha satt et riktig tall inn i baseregisteret.

C) Det finnes nok et register i vårt eksempel som kan brukes til adressering, nemlig indeksregisteret, X. Dette kan brukes på samme måter som baseregisteret ved f.eks.

ADD 55, ,X

D) Indirekte adressering er et meget ofte brukt trick. Med indirekte adressering menes at maskinen først finner en adresse, førsteadressen, nemlig den som uten spesifikasjon av indirekte adressering vil være den aktuelle adresse, dernest brukes det tall den finner i denne adresse som den nye adresse.

Ved indirekte adressering brukes enten relativ adressering eller baseregisteradressering til å finne første-adressen. Dersom en spesifiserer bruk av indeksregisteret i tillegg ved indirekte adressering vil innholdet av indeksregisteret bli lagt til det maskinen finner i første-adressen. Indirekte adressering spesifiseres ved bruk av:

ADD I 55

Den mere detaljerte oppsetting av de regler hvoretter den aktuelle adresse beregnes finnes i håndboken.

Vi setter opp her alle mulige former som kan forekomme:

```
ADD     55        Relativt (P)
ADD     55 ,B     Relativt (B)
ADD     55 ,X     Relativt (X)
ADD I   55        Indirekte gjennom (P) + 55
ADD I   55 ,B     Indirekte gjennom (B) + 55
ADD I   55 ,X     (Indirekte gjennom ((P)+55)+(X)
ADD I   55 ,X ,B  (Indirekte gjennom ((B)+55)+(X)
ADD     55 ,X ,B  Relativt (B) + (X)
```

E) Symbolske adresser er innført ved skrivning av program for å lette programmering med relative adresser. Når programmet skrives kan en definere nye symboler ved å skrive et symbol på maks. 4 bokstaver etterfulgt av komma til venstre på den linjen en instruksjon eller en konstant står på.

Eksempel:

```
20/  OLE,  0
     PER,      ADD OLE
```

Her blir OLE og PER av assembleren definert til å ha verdien henholdsvis 20 og 21.

3.4 Bruk av registre

De fleste datamaskiner har minst et programbevisst register, A. Ved siden av dette har maskinene også et halv-bevisst register P. Dette kan programmet ikke bruke generelt fordi dette register brukes til å holde rede på neste instruksjon. Flere maskiner har imidlertid andre bevisste registre, og vårt eksempel har ialt 6. Disse er:

A	Akkumulator
D	Forlenget akkumulator
B	Baseregister
X	Indeksregister
L	Linkregister
T	Temporært register

Av disse brukes A og D i aritmetiske og logiske instruksjoner, B og X brukes vesentlig til adressering, X registeret brukes også til telling, og L og T registeret er nærmest beregnet til å være generelle hurtige noteringsposisjoner. Bruken av registerne fremgår ellers mest av de enkelte instruksjoner.

3.5 Instruksjoner

Vi skal nå komme inn på de konkrete instruksjoner som finnes i vårt eksempel. Vi tar først for oss gruppen A 1, Aritmetiske instruksjoner. Disse er alle hukommelsesinstruksjoner. Av disse har vi allerede omtalt ADD.

STZ setter 0 i aktuell adresse i hukommelsen.

Denne instruksjonen er hendig for initsialiseringer.

STA, STT, STX er alle "store" instruksjoner som overfører innholdet fra et register (A, T og X) til hukommelsen i den aktuelle adresse. Minst en av disse instruksjoner er nødvendige, de andre er "hendige".

STD overfører innholdet fra to registre, nemlig A og D til to etterfølgende adresser i hukommelsen. Denn instruksjon sparer i mange tilfelle både plass og tid.

LDA, LDT, LDX er såkalte "load"-instruksjoner. Disse overfører innholdet av den aktuelle adresse i hukommelsen til et register, henholdsvis A, T og X.

LDD overfører fra aktuell adresse to etterfølgende ord til de to registrene A og D.

MIN er en meget spesiell instruksjon. Den teller i aktuell adresse og sørger for en skip dersom resultatet derved blir lik 0. D.v.s. den adderer 1 til innholdet i adressen. Dette er en meget verdifull instruksjon som gjør det mulig å skrive såkalte programløkker med få instruksjoner og som er hurtige. Den instruksjonen kunne også regnes under kontroll-instruksjonene.

ADD og SUB er de eneste "egentlige" aritmetiske instruksjoner. Det er mulig å utstyre maskinen med flere aritmetiske instruksjoner som f.eks. MUL, DIV etc. Blant hukommelsesreferanseinstruksjonene har vi to instruksjoner i klassen A 2, logiske instruksjoner.

AND utfører en logisk "and" operasjon mellom de enkelte bit i A og innholdet i den aktuelle adresse, og setter resultatet i A.

ORA utfører tilsvarende en logisk "or" operasjon.

Av klassen A 3, input-output instruksjoner finnes egentlig bare en instruksjon, IOT, men som til gjengjeld er en såkalt mikroprogrammerbar instruksjon. Det betyr at denne instruksjonen, som ikke har noen hukommelsesreferanse, har en rekke undersymboler som kan kombineres i nesten vilkårlige sammenstillinger. De forskjellige variasjoner en får under denne instruksjon bestemmes av de tilkoblede perifere utstyr. Disse variasjoner blir derfor nevnt senere under 8 Input-output.

Av klassen A 4, registeroverføringsfunksjoner har vi ialt 3 stykker. Disse er alle meget komplexe og skal bare nevnes kort, summarisk her.

SHT er en såkalt "shift"-instruksjon. Den er mikroprogrammerbar med undersymboler som bestemmer retning, koblinger og indeksmodifikasjon av antall shift. Shiftoperasjonen kan ha en logisk eller en aritmetisk betydning. Selve operasjonen består i at det bit-mønster som står i vedkommende register, A eller D, skiftes et antall plasser til siden.

ROP er en operasjon for overføringer og addisjoner mellom registrene inklusive P. Dette er også en meget komplex instruksjon som kan utføre meget. Den er selvsagt mikroprogrammerbar med undersymboler for

valg av registre
spesifikasjon om addisjon
spesifikasjon om logiske operasjoner
spesifikasjon om addisjon av carry
spesifikasjon om addisjon av 1

Denne instruksjonen "overføldiggjør" mange av de andre. Flere av de spesielle kombinasjoner er så meget brukt at de har fått egne symboler.

MIS er "diverse"-instruksjon som har diverse underkoder. Disse kombineres bare i noen få tilfelle, slik at de fleste underkombinasjoner har fått egne symboler. Denne instruksjon med underkode for registeroverføring til A heter TRA med et tilleggssymbol for å identifisere kilden. Den hyppigste bruken av TRA er for overføringer fra håndswitchregisteret OPR som betjenes av operatøren.

Instruksjonen i klassen B 1 heter WAIT. Denne instruksjonen stopper programmet på det aktuelle prioritetsnivå. Er det lavere prioriterte programmer som venter på tur startes disse automatisk. På laveste prioritetsnivå stopper maskinen.

Instruksjonen i klasse B 2, ubetinget hopp, er en hukommelsesreferanseinstruksjon.

JMP betyr at maskinen skal ta sin neste instruksjon fra den aktuelle adresse.

Instruksjonen i klasse B 3, betingede hopp, har flere underkoder med egne fullstendige symboler. Disse instruksjoner refererer til hukommelsen, men bare med relativ adressering.

JAP betyr hopp dersom A inneholder et positivt tall, JAN og JAZ betyr det tilsvarende for negativt tall eller 0 i A.

JXN, JXZ hopper dersom indeksregisteret inneholder negativt tall eller 0.

JPC adderer 1 til indeksregisteret og hopper dersom dette register etter addisjonen er positivt.

JNC gjør det samme, men hopper dersom X blir negativt.

I klassen B 4, betingede skip instruksjoner finnes en mikroprogrammerbar instruksjon med mange kombinasjoner. SKP har underkoder for forskjellige betingelser som:

valg av registre for sammenligning
spesifikasjon av sammenligning
utprøving av spesielle betingelser.

3.6 Programløkker

Vi skal så ta for oss noen av teknikkene ved skrivning av programmer.

Dersom en ved skrivning av program bare var henvist til å la hver instruksjon bli utført en gang kom en ikke særlig langt. Noe av de viktigste teknikker er derfor å skrive programløkker d.v.s. programbiter som utføres et antall ganger. Ofte vil en ha flere løkker inn i hverandre. Som eksempel tar vi en programmering av algoritmen i Kap. 1:

```
ESPEN,  LDA   TALL
        STA   JENS
        STA   HANS          % PUNKT 1
LOOP11, LDA   HANS
        SUB   ( 1
        STA   HANS          % PUNKT 2
        JAZ   STOP          % PUNKT 3
        LDA   TALL
        ADD   JENS          % PUNKT 4
        JMP   LOOP 11       % PUNKT 5
JENS      0
HANS      0
STOP,   WAIT                % HER STOPPER PROGRAMMET
```

Det som skrives til høyre for %-tegnet på linjen betraktes kun som kommentar til programmet.

Som for alle programmer er det for dette eksempel noen forutsetninger for at det skal virke (dess "bedre" programmer, dess ferre forutsetninger). Disse er her:

- "Tallet" som skal kvadreres står i adresse TALL
- TALL må ikke være nær dette program
- "TAllet" i TALL må være positivt

Ved å sløyfe den første instruksjon blir forutsetningene endret til at "tallet" må være i A, noe som er mere "naturlig". Videre vil resultatet også ved hoppet til STOP stå i A noe som også er "naturlig".

I dette eksemplet var det størrelsen av "tallet" som var det avgjørende for hvor mange ganger løkken skulle gjennomløpes. Det finnes mange andre måter å bestemm antall gjennomløpninger på.

```
        COPY CM2 SA DA      % skift tegn i A
        STA COUNT1
LOOP1, BLA BLA
        BLA BLA
        MIN COUNT11         % tell opp
        JMP LOOP1
VIDRE, FORTSETT
```

Ved dette eksempel vil antall ganger loopen skal gjennomløpes bestemmes av tallet i A som her må være positivt og større enn 0. (Skal 0 også tillates må en sette inn en instruksjon JAZ VIDRE foran LOOP1).

Det forutsettes at en adresse kalt COUNT1 er innen rekkevidde.

Dersom antall ganger loopen skal gjennomløpes står i X-registeret kan vi skrive

```
            COPY CM2 SX DX      % Skift tegn i X
LOOP2       BLA BLA
            BLA BLA
            JNC LOOP2
VIDRE,      FORTSETT
```

Ved bruk av instruksjonen JNC var vi her istand til å spare 35 % plass i hukommelsen (8:5)

Oppgave: Skriv det samme uten å bruke hverken MIN eller JNC.

3.7. Subrutiner

Subrutiner er et meget vesentlig begrep i maskinkodeprogrammering.

Når et program for løsning av en oppgave skrives må og bør programmet deles opp i underdeler. Det bør være et kort hovedprogram som "administrerer" bruken av underdelene. Underdelene blir da såkalte subrutiner. Videre er det ofte at en underoppgave skal gjøres mere enn en gang, men med forskjellige parametre. Programmet for denne underoppgave skrives en gang men slik at den tar hensyn til de forskjellige parametre. Dette program blir da en subrutine.

Vi kan betrakte praktisk talt all programmering som en programmering av større eller mindre subrutiner. Hovedprogrammene kan nesten alltid gjøres trivielle (og bør gjøres trivielle).

Bruk av en subrutine i et hovedprogram gjøres ved at hovedprogrammet har et hopp til subrutinen, og subrutinen har et hopp tilbake til hovedprogrammet. Det er karakteristisk for subrutinen at den skal kunne brukes på flere steder i hovedprogrammet, følgelig må den kunne hoppe tilbake til det riktige sted i hovedprogrammet. For å kunne gjøre det må den fra hovedprogrammet få informasjon om hvor den skal returnere når den er ferdig.

Dette gjøres med spesielle konvensjoner og instruksjoner. Ofte slik at returadressen blir stående i L-registeret.

Videre må subrutinen få overført sine data og parametere fra hovedprogrammet. Her kan konvensjonene være mange, men mest brukt er å ha en parameter stående i A ved hopp til subrutinen, og ha resultatet i A ved retur fra subrutinen.

Som typisk eksempel setter vi

```
        BLA BLA
        LDA ARGUMENT            % GJØR KLART ARG

        JPL SINUS
        STA RESULTAT            % SUBRUTINEN
                                  RETURNERER HER

        BLA BLA
```

Denne skrivemåte forutsetter at SINUS står i nærheten, det er sjeldnere tilfelle, og generelt må en skrive hoppet med indirekte adressering.

```
LDA  ARGUMENT
JPL  I (SINUS
STA  RESULTAT
— —
— —
) FILL
```

3.8 Input - output

Som tidligere nevnt vil de enkelte instruksjoner under input-output avhenge av det enkelte perifere utstyrs konstruksjon. Vi skal først ta for oss det enkeleste som er input fra hurtig papirbåndleser.

Det er her snakk om overføring fra papirbåndleseren til A.

Statusinformasjonen når det gjelder papirbåndleseren er således enkel, enten er leseren "opptatt", d.v.s. iferd med å bevege båndet frem til avlesning av neste karakter, eller klar d.v.s. klar til å avgi 8 bits informasjon fra hullbåndet.

Papirbåndleseren har koden REA.

Innlesning og oppstarting av båndet til neste karakter har koden: ACT.

Skip dersom leseren er klar har koden: SKA

Disse kodene kan nå kombineres til følgende tre signifikante instruksjoner:

```
a)  IOT SKA  REA        % SKIP DERSOM LESEREN ER KLAR
b)  IOT SKA ACT REA     % DERSOM LESEREN ER KLAR
                        % SKIP OG LES INN
c)  IOT ACT REA         % LES INN
```

Med tilfelle c vil det vare bli lest noe inn hvis leseren er klar, dersom leseren ikke var klar vil programmet bare gå videre uten å gjøre noen ting.

Da leserens tider er høyst variable skrives de fleste innlesninger slik at programmet venter på at leseren skal bli klar i en liten venteløkke som følger:

```
-
IOT ACT SKA REA
JMP * - 1
-
```

Adressen * - 1 betyr den overforstående adresse. Når programmet her går videre har A fått innlest en 8-bits karakter.

Når det gjelder Teletype har denne to funksjoner, en skrivefunksjon med koden PNT og en lesefunksjon med koden RKE.

Den elektroniske kontrollenhet for Teletypen er slik konstruert at kodene SKA og ACT har samme funksjon som i det foregående eksempel. Det betyr at innlesning med venting gjøres med instruksjonene:

```
-
IOT ACT SKA RRE
JMP * - 1
-
```

Programmet vil her ikke komme videre før en tast er trykket ned. Når programmet går videre har den koden for den nedtrykkede tast i A.

Utskrift av en kode gjøres med følgende venteløkke:

```
-
LDA KODE              % KODEN PLUKKES OPP
IOT ACT SKA PNT       % UTSKRIFT
JMP * - 1             % VENTELØKKE
-
```

I tillegg til funksjonene for ACT og PNT har kontrollen innebygd forståelse av kodene PIN og SNI.

Disse kodene kontrollerer interrupt-givningen fra Teletype-kontrollen. Når PIN skrives i instruksjonen betyr dette "Prepare interrupt" og Teletypen vil gi en interrupt når den blir klar.

Hver gang man bruker ACT må PIN gis dersom en ønsker interrupt når Teletypen blir klar (ellers blir vedkommende utstyrs kontroll for interrupt 0-stilt).

En interrupt fra Teletype fører til at ethvert program av lavere prioritet avbrytes, og at det spesielle program for Teletypen startes opp og kjøres til første WAIT instruksjon. For å få dette til må det skrives noe spesielt program som også benytter koden SNI. SNI betyr "Skip on no interrupt". I tilfelle flere interruptgivere er koblet sammen må et program for hver interrupt finne "den skyldige". Dette gjøres med bruk av SNI.

Mere detaljert forklaring på bruk av interrupt finnes under seksjonen om interrupt.

Når det gjelder den videre programmering av input-output, er dette avhengig av hvert enkelt utstyrs kontrollenhet.

Generelt skal resymeres betydningen av de tilgjenelige koder:

Alle input-output instruksjoner har et format:

IOT SSS DEV

der SSS er et eller flere styresymboler og DEV er symbolet for vedkommende enhet.

SSS består av en kombinasjon av

ACT - Generell inn- og utlesning, samt oppstarting
SKA - Skip dersom utstyret er klart.
PIN - Gi interrupt når utstyret blir klart
SNI - Skip dersom dette utstyr ikke har gitt interrupt. SNI kan ikke kombineres med noe annet symbol.

Programmeringen av datakanal og utstyr tilkoblet til denne krever spesielle koder i A, og at hensyn tas til flere forskjellige statussignaler som innleses til A med instruksjoner som nevnt ovenfor.

3.9 Interrupt og Multiprogramsystem

Interruptsystem finnes i noen grad på de fleste datamaskiner. De fleste maskiner har et primitivt system hvorved ethvert program avbrytes ved innkomsten av et interruptsignal og kontrollen overføres til adresse 1, mens det gamle innholdet av P registeret lagret i adresse 0. Programmet som starter i adress 1 må så "ordne opp" for det avbrutte program, identifisere "den skyldige", bestemme de relative prioriteter, og starte opp det eventuelt ønskede program. Noen maskiner har flere interruptkanaler for en enklere identifikasjon av den skyldige og for en enklere beregning av prioriteter.

Noen maskiner har en automatisk sammenligning mellom prioriteten av det løpende program og de innkommende signaler.

Meget få har også en automatisk overgang mellom prioriterte programmer og muligheter for endring av de relative prioriteter efter at en avbruddsrekke har etablert seg. (D.v.s. flere programmer har avbrutt de lavere prioriterte programmer).

Det system hvis brukher skal skisseres kan også kalles et automatisk multiprogrammeringssystem. Dets hensikt er

A) A kunne gi service til flere externe interruptkilder med et minimum av generelt program.

B) Muliggjøre kvasi-parallell-kjøring av flere programmer med forskjellig prioritet med bruk av et lite overvåkingsprogram.

Generelt for dette multiprogramsystem har vi at program kjøres på ett av 16 programnivåer. Overgangen mellom programmene er enten av typen overgang til høyere prioritet eller overgang til lavere priorftet.

Overgang til lavere prioritert programnivå skjer vanlig ved at det arbeidende program kommer til en WAIT-instruksjon. Overgang til lavere prioritert program kan også skje ved at et høyere prioritert program avbryter og stiller programmet "utenfor rekkefølge" slik at når det høyere prioriterte program oppgir ånden, d.v.s. kommer til en WAIT kan det bli et nominelt lavere prioritert program som overtar inntil vårt program igjen blir satt"i rekkefølgen". Dette er forøvrig et sjeldnere tilfelle. Uten slik dynamisk inngripen i de relative prioriteter skjer overgangen til et lavere prioritert programnivå ved at det høyere program kommer til en WAIT.

Overgang til høyere prioritert program skjer bare ved interrupt. I multiprogramsystemet kommer det interruptsignaler hvert 20 m.sek. på et passende topp-nivå. Disse signaler engasjerer et lite overvåkingsprogram som eventuelt starter opp høyere prioriterte programmer. Det som må gjøres for denne oppstarting er følgende:

```
TRA  PID     % LESER STATUS FOR VENTENDE NIVAER
             % INN I A
ADD (NN      % ADDERER KODE FOR NYTT NIVA
MST PID      % SETTER NY STATUS
             % FOR VENTENDE NIVAER
WAIT         % OVERGANG TIL HØYEST PRIORITERT
             % VENTENDE NIVA
```

Overgang til høyere nivå styres ellers generelt ved at en høyere prioritert interruptsignal mottas utenfra. Hvert av de 16 nivåer har sin elektriske terminal som kan tilkobles de forskjelligste enheter.

Koblingen mellom nivå og program skjer ved at hvert nivå har sin faste adresse i hukommelsen for lagring av en nivå-peker. Nivå-pekeren gir adressen til et nivå-hode som inneholder 8 ord for lagring av de 6 arbeidsregistre A, D, X, B, L og T samt P og statusinformasjon (Overflow og carry) som hører til et program.

Når et program avbrytes, enten ved WAIT eller interrupt, blir nivåhodet fylt, en ny nivå-peker valgt og registre satt fra det nye nivåhodet.

Når det gjelder input-output enheter er "ferdig"-interruptene fra disse gjerne koblet sammen på et nivå. Det betyr at programmet på dette nivået må identifisere den enhet som gav interrupt. Dette gjøres ved input-output instruksjonen og symbolet SNI. Skal en finne ut hvilken av de 3 enheter, PNT, RKE eller DEV som var skyldig, d.v.s. hoppe til en av tre behandlingsprogrammer for disse enheter, kan en bruke følgende programbit.

```
-
-
IOT SNI PNT
JMP PRINTREADY 1
IOT SNI RKE
JMP READYREADY 2
```

```
IOT SNI DEV
JMP DEVICEREADY 3
JMP INTERUPTERROR 1
```

En vil bare få interrupt fra en input-output enhet dersom en ved en siste ACT-instruksjon også spesifiserte PIN. Dersom en har spesifisert PIN er interruptsignalet det samme signal som sier fra at enheten er klar.

4 ASSEMBLER

4.1 Generelt om assembler

Programmering i maskinkode forutsetter at det eksisterer et oversetterprogram som kan avlese det skrevne program og sette de tilsvarende binære koder på plass i hukommelsen. For at oversetterprogrammet, assembleren, skal kunne lese det skrevne program må dette være skrevet på en maskinlesbar form. Oftest brukt for mindre maskiner er papirhullbånd. Hullkort kan også brukes. Oversettelsen er ofte en prosess som må gå i flere trinn. For mange maskiner må assembleren få innlest papirbåndet 2 ganger, derpå punches et nytt papirhullbånd, en "binærtape" som så til slutt leses inn av et såkalt "loader"-program. Programmet skrevet i maskinkode kalles "source"-program. Det ferdige program i binærkode kalles objekt-program. Assemblere som krever innlesning 2 ganger kalles 2-pass assemblere.

De fleste assemblere i dag er symbolske, d.v.s. at source-programmet kan skrives med symboler d.v.s.mnemonics for maskinkonstruksjonene og fritt valgte symboler for adresser.

De videre egenskaper i assemblere skjuler seg bak de tillatte former for tegnsetting og hvilke kommandoer assembleren adlyder.

Den assembler som vi velger som eksempel er kalt SMIL og representerer et meget raffinert siste ledd i en lang utviklingskjede. SMIL er 1-pass og kan assemble enten direkte inn i hukommelsen eller ut på et papirhullbånd. Den er normalt tilstede i hukommelsen under hele tiden et program både assembles og kjøres, og kan også brukes i rollen som et hjelpeprogram under feilfinningen i objektprogrammer. Den virker også som en bibliotekar til å plukke ut ønskede subrutiner fra et subrutinebibliotek.

Assembleren tar opp ca. 2000 ord i hukommelsen inklusive alle tabeller.

Visse kommandoer er felles for assembleren og det som forstås direkte av maskinen i en spesiell innebygget primitiv oktal-assembler. Disse er så fundamentale at de vises her:

YY/XX

Skråstreken her betyr at tallet XX skal settes inn i adresse YY.

Vognretur - ny linje

Vognretur og ny linje betyr avslutningen av det som skal inn i et ord i hukommelsen.

XX!

Utropstegn betyr at assembleren skal slutte med å assemble og istedenfor starte maskinen i adresse XX.

Begge assemblere leser oktaltall og benytter de siste 6 skrevne siffer.

Begge assemblere leser source-program fra on-line Teletype.

4.2 Symbolske formater

Av de symboler en bruker legger assembleren merke til de 4 siste bokstaver. Assembleren adderer sammen verdien av de symboler som står sammen. Egendefinisjon av symboler gjøres enten som symbolsk adresse eller direkte ved bruk av =.

En spesiell egenskap er bruk av

"(XX

Dette er et signal til assembleren om at den når kommandoen)FILL gis skal finne plass i hukommelsen til verdien av XX og derpå sette adressen til denne plass istedenfor det skrevne uttrykk (XX.
Dette sparer programmereren for meget skrivearbeid.

4.3 Kommandoer

Ved siden av den tekst som skal bli til program oppfatter også assembleren kommandoer til seg selv. Disse er dels skrevet v.h.a. spesielle tegn dels ved bruk av

")"

foran et gitt, reservert symbol.

)TAPE er således kommandoen for å starte assembler med input fra papirbånd.

Programmereren kan selv definere nye ønskede kommandoer.