7. august 2012

Programmering for
Nintendo Entertainment System

En introduksjon

Johan Fjeldtvedt

Innhold

INNHOLD


Nintendo
ENTERTAINMENT SYSTEM
POWER
RESET
SELECT START
Nintendo
B
A

Kapittel 1

Introduksjon

<her kommer en introduksjon>

1.1 Før vi setter ut ...

Denne guiden tar for seg å programmere en nesten 30 år gammel spillmaskin. Jeg går ut i fra at de fleste som vil prøve seg på dette har en interesse for, og grunnleggende kunnskaper innen programmering. Det forventes ikke at du kan assembly-språk for 6502 eller hvordan NES-en fungerer internt – det skal vi ta fra bunnen av. Men grunnleggende programmeringskonsepter som programflyt, bits og bytes o.l. vil tas i bruk uten noen videre forklaring. Det er derfor en stor fordel om man i det minste har kjennskap til et annet mer vanlig programmeringsspråk. Hvis det er noen ildsjeler som elsker NES, men som dessverre aldri har programmert – fortvil ikke – det bør gå an å lære seg dette likevel, men beregn en del mer kløing i hodet.

Over til noen mer håndfaste krav. For å programmere til NES trenger du i grunn tre verktøy:

- En assembler. Dette er et program som gjør om maskinkode i menneskelig form, og så kalt assemblykode, om til binær kode som kan forstås av prosessoren i NES. I guiden legges det opp til å bruke assembleren «ca65», men assemblykoden er lik uansett assembler som brukes. Det er likevel noen små forskjeller i hvilke tegn som brukes for spesielle *assemblerdirektiv* (kommandoer til assembleren, egentlig ikke en del av maskinkoden.)

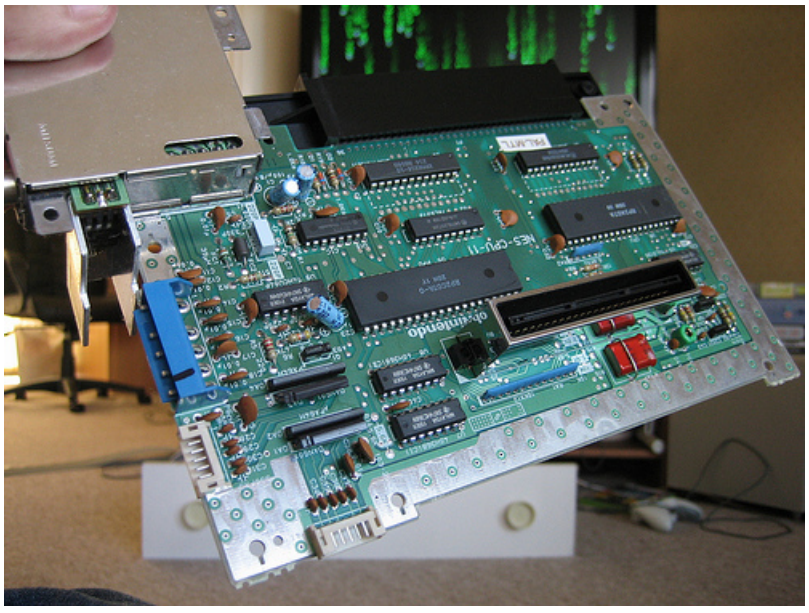
- Et grafikkprogram (dvs. tegneprogram). Dette gjør det mulig for deg å lage grafikk på en sivilisert måte, og heller få et program til å gjøre det om til binærkoder som grafikkprosessoren forstår. I guiden legges det opp til å bruke programmet «Tile Molester».

- En NES-emulator. Med mindre du har en såkalt flashcart du kan legge inn programmene dine på, for så å prøve dem ut på en ekte NES. En anbefalt emulator er «Nintendulator» for Windows, og «Nestopia» på Mac. I Linux er det flere alternativer.

1.2 Grunnleggende om NES

Før vi i det hele tatt kan tenke på å tilnærme oss å programmere NES, må vi i det minste ha et visst innblikk i hvordan den fungerer, og det i større grad enn vi trenger for programmering til moderne maskiner. Dagens datamaskiner og konsoller har operativsystemer (OS) og firmware. Disse fungerer som et *abstraksjonslag* mellom programmene vi lager, og maskinvaren. Vi trenger aldri å kommandere maskinvaren direkte – vi trenger bare å be OS-et eller firmwaren om å gjøre en oppgave, og detaljene blir tatt hånd om av dem. Slik er det ikke med NES. Det er ingenting som kjører mellom spillet vi lager og maskinvaren. Det ville ha vært alt for ressurskrevende med den beskjedne maskinvaren som sitter i NES – og ressurser er noe vi etter hvert skal få erfare at det ikke er ubegrenset av. Resultatet av dette blir at vi må gjøre veldig mye fra bunnen av. Vi må programmere både operativsystem og spill på en gang!

Dette høres kanskje skummelt ut, men det er også en del av sjarmen og det som er utfordrende ved NES-programmering. Noe så trivielt som å få en figur fram på skjermen – som kan gjøres med et fåtall linjer C-kode på en moderne maskin – blir et lite prosjekt i seg selv. Et av de enkleste programmene man kan lage til NES, et program som viser en liten 8x8 pikslers figur på skjermen, innebærer eksempelvis en hel mengde arbeid: Man må sette opp grafikkprosessoren riktig, kopiere en *fargepalett* inn i minnet dens, kopiere informasjon om figuren til «figurminnet» og så videre. Alt dette må vi instruere prosessoren om å gjøre med veldig enkle instruksjoner av typen «hent en byte fra minnet til den interne lagringsplassen kalt A, pluss tallet 3 med tallet i A og lagre det i A, legg verdien i A inn i minneadresse 1234».



Figur 1.1: *Hovedkortet i NES. Den store brikken til venstre er CPU-en, den store til høyre er PPU-en. Noen av de mindre brikkene er RAM-brikker til CPU-en og PPU-en, mens andre er logiske kretser for dekoding av adressebussen til CPU-en og PPU-en.*

Siden vi må jobbe så tett med maskinvaren er det kanskje en idé å i det hele tatt begynne å se på hva som faktisk er inni den. På det nærmest rent teoretiske er ikke NES så ulik dagens

datamaskiner. Den består av de samme grunnleggende komponentene: Den har en prosessor, en grafikkprosessor, og så videre. For å være mer spesifikk har NES:

- En 8-bits prosessor (CPU)
  - Bygd på 6502-arkitekturen (samme prosessorarkitektur som ble brukt i andre legendariske 70- og 80-tallsmaskiner som Atari 2600, Commodore 64 og Apples første maskiner)
  - Klokkehastighet på hele 1.79 MHz (amerikansk/japansk versjon) eller 1.66 MHz (europeisk/australsk versjon.)
  - Prosessoren har innebygd lydprosessor (APU) bestående av fem lydgeneratorer som kan lage forskjellige typer lydbølger.
- 2KiB arbeidsminne (RAM).
- 2KiB grafikkminne (VRAM).
- En grafikkprosessor (PPU – Picture Processing Unit) med støtte for 52 farger (25 samtidig på skjermen) og en oppløsning på opp til 256x240 piksler.
- To serielle kontrollporter.

1.2.1 Regionale varianter

Som det står i listen ovenfor, er bl.a. klokkefrekvensen til prosessoren forskjellig i forskjellige utgaver av prosessoren. Dette har med TV-standardene rundt om i verden å gjøre.

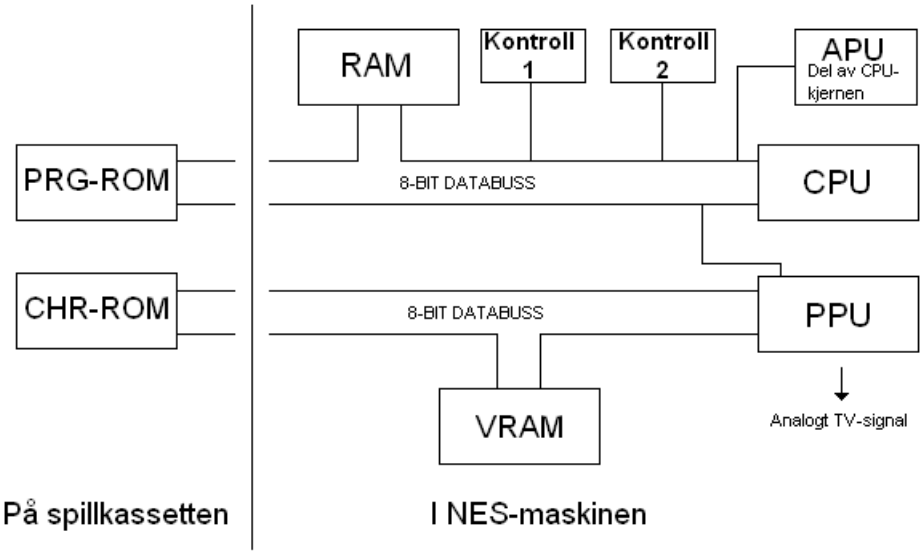
Prosessorhastigheten og skjermoppløsningen avhenger av hvilken *region* NES-en er laget for. Analoge TV-signaler, som er det NES-ens PPU genererer, er nemlig ikke kodet i et universalt format verden over. Dette forklares av litt historie: USA var, som med mye annet, først ute med farge-TV. TV-signalene ble kodet i et spesielt format kalt «NTSC» for å kunne inneholde fargeinformasjon og samtidig være kompatibelt med et vanlig svart-hvitt TV-signal. Da europeiske land hengte seg på farge-TV-bølgen, oppdaget de i midlertid at NTSC hadde noen svakheter. Signalet tålte eksempelvis nesten ikke forstyrrelser før fargene ble forandret. De ble utviklet et forbedret format, kalt «PAL», i stedet, som ble tatt i bruk i Europa. Uten å gå nærmere innpå de tekniske forskjellene mellom de to, innebærer de blant annet at NTSC-signaler har en noe høyere frekvens enn PAL-signaler. Dette krever at PPU-en i NES må jobbe litt raskere. NES-en er slik designet at både CPU-en og PPU-en blir drevet av samme *klokkegenerator*. Det betyr at hvis PPU-en jobber noe raskere, gjør også CPU-en det. NTSC-konsoller har altså en noe raskere CPU. I tillegg til denne forskjellen, har TV-bilder i PAL-format en noe høyere oppløsning enn NTSC. Mer detaljert:

- PAL: CPU-hastighet på 1.66 MHz, oppdateringsfrekvens 50Hz.
- NTSC: CPU-hastighet på 1.79 MHz, oppdateringsfrekvens 60Hz.

I denne guiden skal vi for det meste forholde oss til ting som ikke er avhengig av hvilken TV-standard vi bruker. Der forskjellene spiller en rolle, kommer vi tilbake til det.

1.2.2 Komponentene i NES

Bildet på forrige side gir et dårlig bilde av hvordan komponentene henger sammen. Det er ikke så interessant hvordan brikkene ser ut eller hvordan de ligger på hovedkortet, det som er viktig for oss som programmerere er hvordan de er koblet sammen. Denne illustrasjonen viser en grov oversikt over det:



Figur 1.2: *Maskinvaren i NES*

Her er det nok noen nye ord og uttrykk. Komponentene er som man ser koblet sammen via en *databuss*. Denne kan du se for deg som «hovedveien» mellom komponentene og CPU-en. Fysisk sett består databussen av åtte elektriske ledere som er koblet på hver sin pinne på prosessorbrikken. Disse lederne går ut til de forskjellige komponentene og har som oppgave å frakte hver sin bit (binært siffer) mellom komponentene og CPU-en. De går f.eks. til PPU-en og til RAM-brikkene som vi ser av figuren. Det kan kanskje virke uklart hvordan den samme databussen muliggjør kommunikasjon med forskjellige komponenter. Hvordan velger prosessoren hvilken komponent som skal sendes til eller hentes data fra? Dette er har den en såkalt *adressebuss* til. På samme måte som databussen består denne av, ikke 8, men 16 ledere. Disse lederne er koblet mellom komponentene og pinnene på prosessorbrikken på en slik måte at ved å sette strøm på forskjellige kombinasjoner av dem, velges de forskjellige komponentene ut. Når en komponent er valgt ved å *sette en adresse*, kan prosessoren lese eller skrive på databussen, og dataene vil hentes fra eller sendes til riktig komponent.

Mer spesifikt hvordan dette fungerer, og hvordan vi bruker adressebussen, skal vi komme tilbake til. Vi vil senere se at hver av adresselinjene representerer binære sifre, og at hver adresse dermed kan representeres av et tall.

CHR-ROM og PRG-ROM er også to nye ord. ROM («Read Only Memory») er minnebrikker som bare kan leses fra. Når strømmen slås av, mister ikke disse innholdet, slik som er tilfellet med RAM. Derfor brukes ROM-brikker til å lagre permanente ting. I PC-er brukes de f.eks. til å

inneholde BIOS-programmet som kjøres når PC-en starter. I NES ligger de på spillkassetten og inneholder selve spillets kode og grafikk. På en vanlig spillkasett for NES, er det to ROM-brikker: PRG-ROM og CHR-ROM.

PRG-ROM («Program ROM») er ROM-brikken som inneholder spillets kode. CHR-ROM («Character ROM») er ROM-brikken som inneholder spillets grafikk. Merk hvordan disse to er koblet direkte til CPU og PPU vi databussene. Når et spill er satt i NES, kobles ROM-brikkene rett på data- og adressebussene til CPU og PPU. CPU-en er programmert til å starte og utføre instruksjonene som ligger lagret i PRG-ROM. PPU-en er i selve kretsdesignet programmert til å hente grafikkdataene fra CHR-ROM.

Som du kanskje ser på diagrammet, er det to separate system i NES. Det er CPU-en, og komponentene som er tilkoblet den via data- og adressebussene, og så er det PPU-en, og komponentene som er tilkoblet den via dens busser. PPU-en er koblet til CPU-en via databussen dens, men komponentene som i sin tur er koblet til PPU-en, er ikke «synlige» for CPU-en. Det betyr at CPU-en ikke har direkte tilgang til VRAM (grafikkminnet) og CHR-ROM. All tilgang til disse fra prosessoren må skje *via* PPU-en. Dette sier deg kanskje ikke så mye ennå, men det skaper en del problemer som vi skal se.

1.3 Prosessoren

Nå har vi sett en grov skisse av maskinvaren i NES og hvordan de forskjellige komponentene henger sammen. Men vi må gå langt mer i detalj dersom vi skal kunne bruke den til noe. Vi begynner med å se på prosessoren (CPU-en) i NES, da det utvilsomt er den mest vitale delen. I alle datamaskiner er prosessoren som kjent hjernen. Det er denne vi programmerer – instruerer hva skal gjøre – når vi skriver et program. Prosessoren i seg selv spiller en mye større rolle for oss enn når man driver med moderne programmering. Vi skal programmere på et mye *lavere nivå*, hvor vi instruerer prosessoren om nøyaktig hva den skal gjøre.

Prosessoren i NES, RP2A03 (NTSC) / RP2A07 (PAL), fins som antydet i to varianter, avhengig av om det er en amerikansk eller europeisk/asiatisk NES. NTSC-versjonen har som vi så en klokkehastighet på 1.79MHz, mens PAL-versjonen har en klokkefrekvens på 1.66MHz. Dette er ikke noen enorm forskjell, men man merker det hvis man prøver et NTSC-spill på en PAL-NES og omvendt. Spesielt lyden forandres, da lyd-prosessoren (APU-en) er en del av CPU-kjernen og dermed også har en noe raskere klokkefrekvens. Man vil også merke at eksempelvis teling av sekunder vil gå kjappere.

Prosessoren er en 8-bits prosessor (i kontrast til dagens 32- eller 64-bits prosessorer). Dette betyr at de største tallene den kan regne med er tall som kan skrives med åtte bits (binære siffer). Det vil si tall mellom 0 og $11111111_2 = 255$. Dette betyr ikke at prosessoren ikke kan regne med større tall, men den kan ikke regne med dem i én operasjon. Å gjøre det tar både lenger tid, og krever at vi programmerer den til å ta deler av tallet om gangen. En annen følge er at databussen bare har en *bredde* på 8 bits. Som vi var inne på er det åtte ledere som går ut fra prosessoren og til komponentene. Det betyr at den kun kan hente eller sende åtte og åtte bits, eller rettere sagt, én og én byte om gangen. Det kan ofte påkoste oss ekstra kode.

Men hvordan fungerer egentlig en prosessor? På et grunnleggende elektrisk nivå vet ikke jeg svaret selv, men det er heller ikke så interessant. Det er nok for oss å vite hvordan prosessoren opererer på et program-nivå og hvordan vi skal programmere den. Prosessoren i NES er ikke så veldig ulik andre og mer moderne prosessorer i både oppbygning og virkemåte.

Figur 1.3: *Ricoh RP2A07, prosessoren i PAL-versjoner av NES*

En fundamental del av alle prosessorer er *registrene*. Et register er en liten intern lagringsplass som brukes til å lagre midlertidige verdier som prosessoren skal bruke når den utfører en operasjon. Hvis man eksempelvis skal legge sammen to tall, må man først hente inn det ene tallet til registeret kalt «<A», for deretter å be prosessoren om å legge sammen verdien i A med det andre tallet. Resultatet vil så lagres i registeret A. Etterpå kan man bruke verdien i dette registeret videre – f.eks. lagre den et sted, sende den til PPU-en, eller utføre en ny operasjon. Registrene spiller altså en vital rolle i hvordan vi tenker når vi programmerer på et så lavt nivå som vi gjør her.

For å komme med et konkret eksempel kan vi tenke oss at vi har et enkelt program som skal ha som eneste oppgave å flytte en figur bortover skjermen (dette er et program vi skal se nærmere på, og faktisk programmere i kapittel 5.) I et slikt program er det flere oppgaver prosessoren må utføre. Den viktigste er kanskje å foreta selve forflytningen. Som vi skal komme innpå i neste delkapittel, har PPU-en et sted i minnet sitt hvor den holder informasjon om objektene på skjermen – deriblant koordinatene deres på skjermbildet. Prosessorens oppave blir da å hente den horisontale koordinaten til figuren fra minnet, deretter øke denne med én, og så lagre den nye koordinaten i minnet til PPU-en, slik at posisjonen blir oppdatert. Som programmerere er det vår jobb å instruere prosessoren om hvordan den skal gjøre dette. Den faktiske koden som gjør dette ser slik ut:

```
LDA #0                      ; Sett A til 0
STA $2003                   ; Lagre i addresseregisteret til PPU-en
LDA horisontal_koordinat    ; Hent den horisontale koordinaten fra
                            ; minnet og legg i A
ADC #1                      ; Legg 1 til koordinaten i A
STA $2004                   ; Lagre innholdet i A i PPU-ens minne
```

Å be prosessoren om å gjøre ting, gjør man via *instruksjoner*. I koden ovenfor er ordene på tre bokstaver til venstre slike instruksjoner. (Alt bak semikolon i hver linje er *kommentarer* til koden.) Ordene er forkortelser for navnet på den operasjonen de representerer. `LDA` er kort for «LoaD Accumulator», `STA` er kort for «STore Accumulator» og `ADC` er kort for «ADd with Carry». Accumulator er som vi skal se senere navnet på et register som vanligvis kun kalles A.

I bunn og grunn kan ikke prosessoren gjøre noe annet enn å utføre instruksjoner. Prosessoren gjør egentlig bare tre ting om og om igjen:

1. Henter en instruksjon fra *adresseområdet*
2. Henter data som skal brukes i instruksjonen
3. Utfører instruksjonen

Denne prosessen gjentas om og om igjen. Hver gang en instruksjon er utført, går prosessoren videre og utfører instruksjonen som kommer etter den forågende i minnet.

1.3.1 Adresseområdet

Prosessoren må hente instruksjoner og data fra et sted. Dette bringer oss inn på temaet adresseområder. Som nevnt brukes *adressebussen* til å «nå frem til» forskjellige komponenter. Men hvordan fungerer dette fra prosessorens ståsted – og viktigere – fra vårt ståsted?

Når prosessoren vil ha tak i noe, sender den en *adresse* ut på adressebussen. Hver eneste komponent koblet på adressebussen har en eller flere unike adresser – tall mellom 0 og 65535. Disse adressene er bestemt av måten lederne i adressebussen er koblet mellom komponentene. [1]

Dette kan virke noe vagt, men for å ta et eksempel så er RAM-brikkene i NES koblet på adressebussen slik at hver byte i minnet får en unik adresse mellom 0 og 2048 (det er 2KiB = 2048 bytes RAM.) Den første byten i minnet har adresse 0, og så videre. Dersom CPU-en vil lese noe fra byte nummer 9 i minnet, sender den altså adressen 9 ut på adressebussen og leser fra databussen. For å ta et annet eksempel er PPU-en også koblet på CPU-en sin adressebuss slik at man kan kommunisere med den. Det er åtte adresser – adressene fra 8192 til 8199 – som går til PPU-en. Disse har hver sine formål. Adressen 8192 går for eksempel til et av PPU-ens *kontrollregistre*. Denne adressen kan man skrive til for å konfigurere hvordan PPU-en skal oppføre seg; om den skal slå av tegningen, tegne uten farger, og så videre.

Det totale spekteret av adresser fra 0 til 65535 kalles prosessorens *adresseområde*. Grunnen til at adresseområdet kun har en størrelse på 65536 bytes er at adressebussen har en bredde på 16 bits (16 ledninger ut fra CPU-en.) Det største tallet man kan uttrykke med 16 binære siffer er tallet $1111111111111111_2 = 65536$ (Totalt 16 ett-tall.)

Når en spillkassett er satt i NES-en, kobles PRG-ROM (minnet der programmet er lagret) inn på adressene fra 32768 til 65535. Det er her CPU-en henter instruksjoner fra når den slås på. Men det er viktig å merke seg at det ikke er noen forskjell på disse adressene og andre adresser. CPU-en kan like godt bli instruert til å kjøre instruksjoner som ligger et annet sted, for eksempel i minnet. Dette er et viktig poeng; *prosessoren vet ikke hva som ligger i forskjellige adresser*. Den henter slavisk en instruksjon fra en gitt adresse og utfører denne, hopper til neste instruksjon og utfører denne, og så videre. Vi skal gå mye nærmere innpå dette i neste kapittel. Da skal vi blant annet se nærmere på en oversikt over alle viktige adresser vi må forholde oss til.

1.4 Grafikk og PPU-en

En nesten like vital del i NES som prosessoren er grafikkprosessoren (PPU). Oppgaven til denne brikken er kort sagt å hente grafikkdata fra CHR-ROM og VRAM, og omgjøre dette til analoge

[1] Hovedprinsippet går ut på at addressepinnene er tilkoblet dekoder-chipper som aktiverer og deaktiverer komponentene basert på hvilke adresser som settes på bussen. I NES-en er det f.eks. slik at den høyeste adresselinjen (den høyeste biten i adressen) fungerer som et aktiveringssignal for ROM-chippen på kasetten. Når denne er 1 aktiveres ROM-chippen. Når den er 0 deaktiveres ROM-chippen. Resultatet blir at alle adresser som har 1 i den høyeste biten (dvs. alle adresser over $8000) vil være adresser som går til ROM-chippen på kasetten.

TV-signaler. PPU-en i NES er ikke en prosessor i samme ånd som CPU-en. Den kjører ikke gitte instruksjoner slik prosessoren gjør, og kan dermed ikke programmeres. Den er forhåndsprogrammert, i selve kretsdesignet, til å gjøre det samme om og om igjen: hente grafikk fra CHR-ROM, sette opp et skjermbilde etter informasjon lagret i VRAM, og omgjøre dette til TV-signaler.

Selve grafikkdataene ligger lagret i CHR-ROM på spillkassetten, men der ligger det ingen informasjon om hvordan disse skal arrangeres på skjermen. Dette må legges inn i VRAM av prosessoren. Det er altså prosessoren som må stå for alt arbeidet som har med oppdatering av grafikk å gjøre. Hvis ingen endringer skjer i VRAM, genererer PPU-en det samme skjermbildet om og om igjen.

1.4.1 TV-en

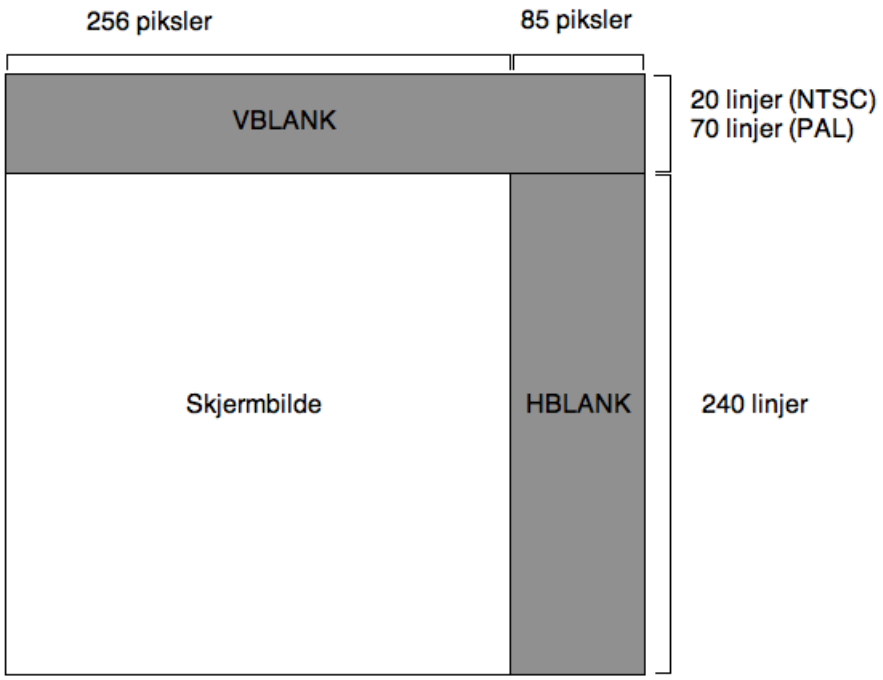
Det analoge TV-apparatet med CRT-skjerm spiller en større rolle enn man kanskje tror når det kommer til gamle konsoller og datamaskiner. Derfor er det viktig å ha en grunnleggende forståelser for hvordan det fungerer – selv om vi knapt bruker slike 'gammeldagse' TV-er lenger!

CRT-TV-en er et analogt apparat. Den har ingen prosessor eller digitale kretser som tar seg av tegning av bildet. TV-en mottar kun et signal som styrer den til å tegne skjermbildet. Det sendes 50 (PAL) eller 60 (NTSC) skjermbilder, kalt *frames*, i løpet av et sekund. En frame består av 288 (PAL) eller 262 (NTSC) linjer, kalt *scanlines*. [2] En scanline kan bestå av et varierende antall dotter (som vi skal kalle *piksler*.) I motsetning til antall scanlines per frame, er det ikke definert hvor mange dotter det kan være i hver linje. Teoretisk sett kan det være så mange man klarer å presse inn fort nok. Grafikkprosessoren i NES klarer å presse ut 256 piksler på én linje.

TV-en består i enkle trinn av en fosforbelagt skjerm, som sender ut lys når det treffes av elektroner, og en elektronkanon som skyter elektroner på skjermen. Elektronkanonen starter med å sikte mot øvre venstre hjørne og beveger seg bortover mot høyre mens den skyter elektroner med forskjellig kraft, basert på innholdet i TV-signalet. Dette resulterer i forskjellige lysstyrker. Når den når enden av skjermen på høyresiden, slås strømmen av, og kanonen flytter seg fort tilbake til venstre side igjen og sikter mot linjen under. Denne forflytningsprosessen kalles *horisontal blanking* (forkortet hblank.) Tegningen av linjer etterfulgt av horisontal blanking, gjentas helt til et skjermbilde, bestående av normalt 288 (PAL) eller 243 (NTSC) linjer er tegnet. Deretter slås elektronkanonen av og forflytter fokus tilbake til øvre venstre hjørne, slik at den er klar til å tegne et nytt bilde. Denne perioden kalles *vertikal blanking* (forkortet vblank).

Det er PPU-en som styrer hele tegningen av et skjermbilde, inkludert blankingen. TV-en i seg selv kan ikke gjøre stort annet enn å slavisk følge TV-signalet som blir gitt. Figur 1.4 viser prosessen fra PPU-ens ståsted. En ny frame starter med at PPU-en sender et *VSYNC-signal*. Dette forteller TV-en at den må posisjonere elektronkanonen øverst i venstre hjørne. Deretter

[2] Det nevnte antallet scanlines virker kanskje noe lite når man tenker på oppløsningen på vanlige TV-overføringer. Det stemmer – et vanlig TV-bilde fra en TV-kanal, DVD-spiller og så videre, består nemlig av *to* frames som vises samtidig ved hjelp av en teknikk som kalles *interlacing*. Med denne metoden kombineres to frames til ett skjermbilde med det dobbelte antallet linjer. Dette fungerer ved at den første framen tegner annenhver scanline, og så tegnes den neste på linjene i mellom. På denne måten oppnår man en dobbelt så høy vertikal oppløsning, på henholdsvis 576 (PAL) og 488 (NTSC) scanlines. NES benytter i midlertid ikke dette for å oppnå høyere grafikkoppløsning, da det ville krevd en betraktelig kraftigere maskinvare for å generere skjermbildet raskt nok.



Figur 1.4: *Signalene som sendes av PPU-en og varigheten av disse. 85 piksler for HBLANK vil si at grafikkprosessoren genererer et HBLANK-signal i løpet av samme tid som den ville rukket å tegne 85 piksler.*

sendes det et blankt signal (elektronkanonen slås da av) i totalt 20 (NTSC) eller 70 (PAL) scanlines. Deretter begynner renderingen av det faktiske skjermbildet. TV-signalet til en linje består av 256 piksler etterfulgt av horisontal blanking. Den horisontale blankingen varer like lenge som det tar å tegne 85 piksler, og i denne perioden sender PPU-en først ut et såkalt HSYNC-signal til TV-en om at den må posisjonere elektronkanonene til å begynne på neste scanline. Deretter sender den svarte piksler (PAL) eller piksler med bakgrunnsfargen (NTSC). En frame generert av PPU-en består alltid av 240 linjer med grafikk, uansett om det er en PAL- eller NTSC-modell. Dermed blir oppløsningen på det synlige bildet tegnet av PPU-en, 256x240 piksler.

I neste kapittel skal vi se at det er viktig å være klar over hvordan TV-bildet tegnes. Det er nemlig kun når PPU-en er i VBLANK-modus at vi kan lese og skrive data til og fra VRAM!

1.4.2 Grafiske elementer

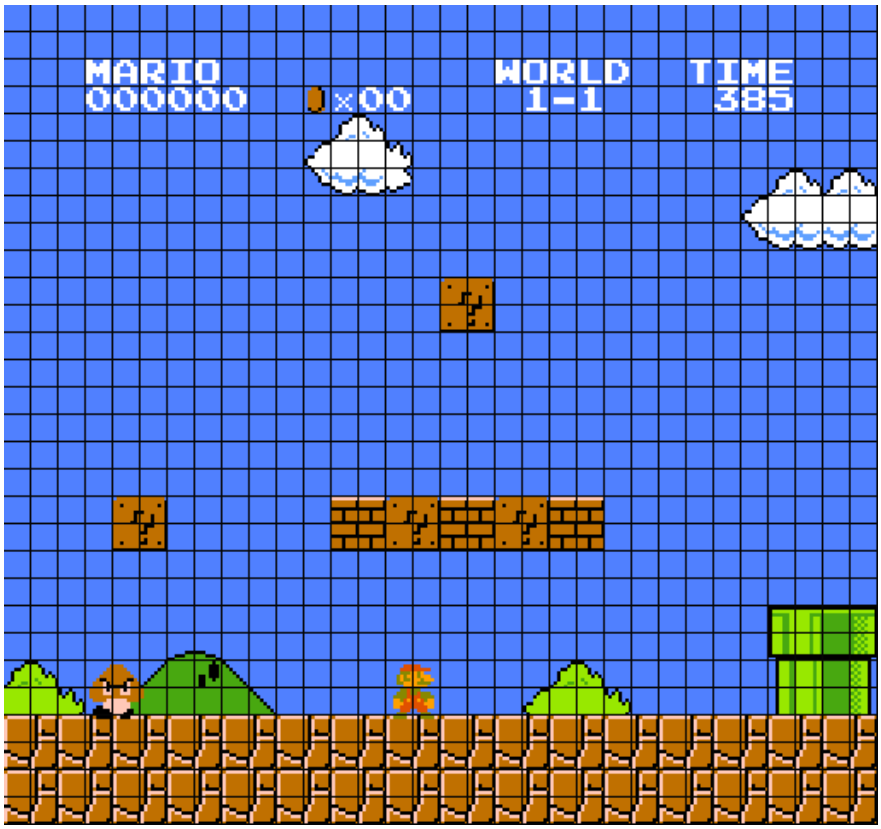
I dagens datamaskiner og konsoller, er pikslene de fundamentale grafiske enhetene. Man kan relativt enkelt endre hver eneste individuelle piksel på skjermen. Dette er derimot ikke mulig med NES. Den har rett og slett ikke nok grafikkminne til å lagre hver eneste piksel i minnet. [3]

[3] Å lagre et bilde på 256×240 piksler ville krevd $256 \cdot 240/2 = 30720$ bytes hvis vi regner med at hver piksel opptar en 4-bits palett-indeks. Sammenlignet med de 2KiB VRAM som er i NES er dette et ganske enormt tall. Så mye RAM ville ført til at produksjonskostnadene for NES ble betraktelig høyere. I tillegg ville det krevd mange flere prosessorinstruksjoner for å manipulere hver eneste piksler individuelt, noe som ville krevd en raskere (og mye

De fundamentale byggesteinene til PPU-en er i stedet såkalte *tiles*. En tile er et kvadrat på 8x8 piksler som ligger lagret i CHR-ROM på kassetten. Det er plass til 512 tiles i CHR-ROM. Tilene kan brukes til to ting: bakgrunner og *sprites*. All ikke-bevegelig grafikk er del av bakgrunnen, mens alle figurer som beveger seg er sprites.

Bakgrunner

En bakgrunn er bygd opp av et rutenett på 32 x 30 tiles. Siden en tile er 8 piksler i lengde og høyde, ser vi at det dekker hele skjermbildet på 256 x 240 piksler. Hvilke tiles som skal plasseres hvor i rutenettet, lagres i VRAM av prosessoren før PPU-en begynner å tegne framen. (**Merk** at å lagre denne informasjonen tar betraktelig mindre plass enn å lagre informasjon om hver eneste piksel på skjermen.) De fleste statiske objekter, altså objekter som ikke flytter seg, er del av bakgrunnen i NES-spill.



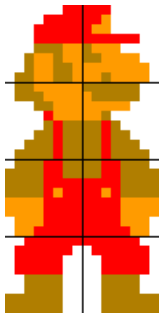
Figur 1.5: *Bakgrunnen er bygd opp av et rutenett på 32 x 30 tiles. Tilenes utseende er låst, men hvordan tilene skal plasseres på skjermen er lagret i VRAM.*

Sprites

En sprite er som nevnt en bevegelig tile. I motsetning til bakgrunnen, som er et rutenett hvor tilene plasseres, er spritene uavhengige objekter som kan plasseres akkurat hvor man vil på

dyrere) CPU enn 6502.

skjermen. PPU-en kan holde orden på opp til 64 sprites. Den har et eget minne for å lagre informasjon om hver av dem, kalt OAM («Object Attribute Memory».) I OAM lagres fire bytes med informasjon om hver sprite: Hvilken tile som skal brukes som spritens grafikk, X- og Y-koordinatene på skjermen, samt forskjellige instillinger for farger, speiling og annet. Spritene kan altså eksempelvis flytte på seg ved at prosessoren forandrer på X- og Y-koordinatene, eller animeres ved at prosessoren bytter mellom forskjellige tiles et visst antall ganger i sekundet.



Figur 1.6: *Mario-figuren består av 8 sprites plassert inntil hverandre*

1.4.3 PPU-ens adresseområde

PPU-en har som vi så i figur en egen data- og adressebuss. Disse spiller akkurat samme rolle som data- og adressebussen til CPU-en – det er gjennom disse PPU-en kan hente og skrive data til sine egne RAM- og ROM-brikker.

Adressebussen har en bredde på 14 bits. Det vil si at adresseområdet er $11111111111111_2 = 16384$ bytes stort. I dette adresseområdet finner vi:

- CHR-ROM, som tar opp 8192KiB. Her lagres selve tile-grafikken som brukes i bakgrunner og sprites. CHR-ROM-brikken ligger altså på spillkassetten
- VRAM, som er 2KiB stort. Her lagres layouten til bakgrunnen, i form av informasjon om hvilke tiles som skal plasseres hvor, samt hvilken *fargepalett* som skal benyttes
- Palett-RAM, som er 32 bytes stort. Her lagres 8 forskjellige fargeutvalg (paletter) som vi vil bruke til forskjellige tiles. 4 for bakgrunnen og 4 for sprites

I tillegg har PPU-en som nevnt et minne for lagring av sprites, kalt OAM. Dette befinner seg inni selve PPU-chippen, og har sin egen 8-bits adresse- og databuss.

1.5 Lyd og APU-en

Lyd vil ikke være det første vi ser nærmere på i guiden, men det kan være greit å få noen begrep om hvordan lyd i det hele tatt produseres i NES-en. En betydelig del av prosessorkjernen tas opp av APU-en (Audio Processing Unit). Det er denne som har som oppgave å ta i mot data fra CPU-en og bruke disse til å produsere lyd.

1.5.1 Lydbølger og svingninger

APU-en fungerer på et såpass lavt nivå at vi først og fremst må ha det klart for oss hva lyd i det hele tatt *er*. Som de fleste sikkert har lært på skolen er lyd svingninger (trykkvariasjoner) som brer seg i lufta. Disse produseres som regel av en eller annen form for høyttaler (drevet av en forsterker) som har som oppgave å gjøre om elektriske lydsignal til luftsvingninger. Det er disse elektriske signalene som produseres av APU-en.

Et lydsignal er rett og slett en potensialforskjell (spenning) som varierer med tiden. Vi må få på plass noen begreper som brukes mye i sammenheng med svingninger og bølger:

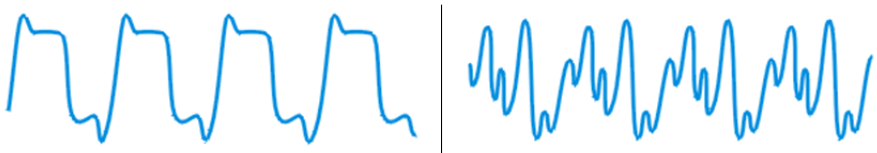
- **Frekvens**: Hvor mange svingninger som skjer i løpet av et sekund. Oppgis med enheten Hertz (Hz). 5 Hz vil altså bety 5 ganger per sekund. Det vanlige symbolet for frekvens er f.
- **Periode**: Hvor lang tid én svingning tar. Oppgis vanligvis i sekunder, men i NES-sammenheng tenker vi oftest i *klokkesykluser*. Det vanlige symbolet for periode er T.

Relasjonen mellom frekvens og periode er enkelt nok $f = 1/T$ og tilsvarende $T = 1/f$ (dette gir mening: For å få perioden deler vi ett sekund på hvor mange ganger i sekundet svingningen gjentar seg).

Hvorfor er dette viktig? Det har seg slik at toner egentlig ikke er noe annet enn lyder med forskjellige *frekvenser*. Desto raskere en lydbølge svinger, desto lysere oppfatter vi tonen.

1.5.2 Bølgeformer

Frekvensen bestemmer altså *tonen* til lyden. Men en lyd har også en *klang*; hvordan den høres ut. Et piano og en fiolin kan godt spille samme tone, det vil si at de produserer lydbølger med samme frekvens, men de har vidt forskjellig klang. Det som avgjør klangen er hvilken *form* lydbølgen har:

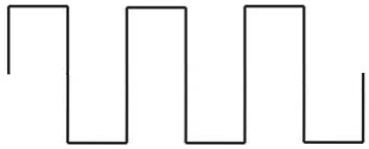


Piano Fiolin

APU-en i NES har fem *lydkanaler* som produserer lydbølger:

- To *firkantbølge*-kanaler (square). Disse brukes som de lysere instrumentene, eller til lydeffekter (den velkjente hoppelyden til Mario lages eksempelvis av en square-kanal).
- En *triangelbølge*-kanal (triangle). Denne brukes hovedsaklig som bass, eller til dype lydeffekter.

- En støykanal (noise). Denne produserer hvit støy av forskjellige slag. Brukes som regel til perkusjon («trommer») eller lydeffekter (typisk eksplosjoner).
- En *deltamodulasjons*kanal (delta). Denne kan ved hjelp av såkalt deltamodulasjon, lage langt mer sofistikerte lydbølger enn de andre kanalene (men krever mer av prosessoren). Brukes i hovedsak til å spille av *samples*.



Firkantbølge

Triangelbølge

Å få APU-en til å lage lyd er relativt enkelt. Prosessoren trenger stort sett bare å aktivere den ønskede lydkanalen og oppgi ønsket periode for lydbølgen. Det som er vanskelig er å gjøre alt til riktig tid, holde tonenes lengde og så videre.

Kapittel 2

Oppsett av utviklingsmiljøet

Nå skal vi se nærmere på dataprogrammene vi skal benytte for å utvikle for NES.

2.1 Assembleren, ca65

Det viktigste programmet vårt er *assembleren*. En assembler tar inn tekstfiler med assembly-kode (prosessorinstruksjoner skrevet med ord), og gjør disse om til den binære koden som instruksjonene oversettes til for at prosessoren skal forstå dem. Det finnes flere assemblere for prosessoren som sitter i NES, men jeg har valgt å bruke assembleren *ca65* i denne guiden. [1]

2.1.1 Installasjon og grunnleggende bruk

ca65 følger med i en større pakke som kalles *cc65*. cc65 er en C-kompilator, der ca65 er assemblerkomponenten av denne kompilatoren. For å få tak i ca65 er det altså bare å laste ned cc65, som finnes her: `ftp://ftp.musoftware.de/pub/uz/cc65/`

Nyeste versjon er ferdig kompilert for Windows, slik at det bare er å laste ned og pakke ut .zip-filen (i skrivende stund cc65-win32-2.13.3-1.zip.) ca65 er et kommandolinjeprogram, så det kan lønne seg å pakke det ut i samme mappe som du kommer til å ha kildekoden din. Hvis ikke bør du legge banen til mappen du har pakket det ut i til PATH-variabelen, slik at du kan kjøre NESASM ved å skrive kommandoen **ca65**.

For Linux- og Mac-brukere: ca65 er ikke ferdig bygd for disse platformene, så det må kompileres fra scratch. Heldigvis er det et relativt enkelt og ukomplisert program, så kompileringen bør by på få problemer. I kildekodepakken (cc65-sources-2.13.3.tar.bz2) ligger det ferdige makefiles slik at man kan kompilere cc65-pakken ved å benytte kommandoene **make** og deretter **make install**. Mac-brukere må ha kompilatoren GCC installert for å gjøre dette. Denne distribueres av Apple og kan lastes ned fra utvikler-sidene deres.

[1] I tidligere guider har jeg brukt assembleren NESASM, men denne har vist seg å ha flere svakheter. ca65 er et proffesjonelt verktøy som brukes for alle maskiner som benytter 6502-prosessorer.

Det er mye informasjon om bruk av **make** og tilføyelser til PATH-variabelen i Windows på internett dersom dette byr på problemer.

Når ca65 er installert er det i grunn klar til bruk. Selve kompileringsprosessen kommer vi straks tilbake til. Det vil være to programmer i cc65-pakken vi kommer til å bruke: `ca65` og `ld65`. Førstnevnte er selve assembleren, sistnevnte er en *linker* som tar forskjellige kompilerte filer og setter dem sammen til en `.nes`-fil.

Når du har satt opp ca65 kan det være en god idé å teste ut en av eksempelfilene som ligger ute her:

`http://home.no/johan-af/nesguide`

La oss se på hvordan vi går frem for å kompilere eksempel 1. De nødvendige filene for å kompilere dette eksempelet er `eks1.asm` og filen `linker_config.cfg`. Når disse er lastet ned åpner vi et terminalvindu, navigerer til mappen de ligger i, og skriver:

```
>> ca65 eks1.asm
>> ld65 -C linker_config.cfg eks1.o -o eks1.nes
ld65: Warning: linker_config.cfg(20): Segment 'GFX' does not exist
ld65: Warning: linker_config.cfg(22): Segment 'RAM' does not exist
>>
```

(>> representerer kommandolinjen.)

De to Warning-linjene kan ignoreres. Vi kommer tilbake til hva disse betyr senere. Hvis ikke noe annet dukket opp på skjermen så gikk kompileringen bra. Da skal det ha bli opprettet en ny fil `eks1.nes`. Denne kan kjøres på en NES-emulator, og hvis alt gikk bra bør det vises en behagelig blå bakgrunn.

2.1.2 Oppbygningen av en kodefil

En av de aller mest elementære tingene vi kan gjøre med NES er å sette en bakgrunnsfarge. Vi skal ta en kikk på koden som utfører dette for å få en bedre forståelse av hvordan man arbeider med NES-kode. Hva selve koden gjør skal vi kikke nærmere på senere i dette kapittelet.

```
; NES spillutviklingsguide

; Eksempel 1 - Oppsett av PPU

; Dette eksempelet gjør følgende:
;   - Venter to frames til PPU-en er klar
;   - Setter opp addresseregisteret i PPU-en til å peke til addressen
;     der bakgrunnsfargen ligger
;   - Sender en byte (en farge) til den oppsatte addressen
;   - Konfigurerer PPU-en til å starte å rendere
;   - Setter CPU-en til å kjøre i en uendelig løkke


; INES-segmentet inneholder det som skal komme helt først i den ferdige
; .NES-filen. Her ligger informasjon som forteller NES-emulatorene hvilken
```

```
; type kasett vi vil at de skal emulere.

.segment "INES"
    .byte "NES", $1A,1,0,9


; VECTORS-segmentet inneholder tre såkalte adressevektorer. Disse peker til
; tre adresser som prosessoren må vite om. I dette eksempelet er det bare
; Start-addressen som er nødvendig for prosessoren å vite om.

.segment "VECTORS"
    .word 0, Start, 0

.segment "RAM"

lol:    .byte 0

; CODE-segmentet inneholder selve koden.

.segment "CODE"

Start:
    lda lol
    ; Vent til det har gått en frame

    lda $2002       ; Hent verdien i PPU-ens statusregister inn til
                    ; A-registeret.
    bpl Start       ; Hvis den syvende biten er satt så er ikke
                    ; PPU-en klar. Da går vi til Start igjen og
                    ; venter igjen.

    ; Vent én frame til:

:   lda $2002       ; Kolonet markerer en anonym label
    bpl :-          ; Hopp til forrige anonyme label (linja ovenfor)

    ; Sett opp addressen som fargen skal kopieres over til:

    lda #$3F        ; A = $3F
    sta $2006       ; Lagre As verdi i PPU-registeret $2006
    lda #$00        ; A = $00
    sta $2006       ; Lagre As verdi i PPU-registeret igjen

    ; Nå har vi fortalt PPU-en at vi ønsker å sende en verdi til det stedet
    ; i addresseområdet dens som har addresse $3F00. Dette er addressen som
    ; peker til det stedet i palett-minnet hvor bakgrunnsfargen ligger.

    ; Nå ønsker vi å sende en farge til denne addressen. Da tar vi verdien
    ; inn i et register, og så lagrer vi registerverdien i PPU-addressen
    ; $2007. Verdien vil da lagres i PPU-minnet, i den addressen som ble
    ; satt i ovenfor ($3F00).

    lda #$11        ; A = #$11 (en blå farge)
    sta $2007       ; Lagre A-verdien i $2007

    ; Nå har vi sendt det vi ønsker til PPU-en, og vi er klare for å la den
```

```
    ; tegne skjermbildet. Da sender vi et sett med instillinger til
    ; de to konfigurasjonsregistrene til PPU-en; $2000 og $2001.

    lda #0          ; Bare lagre A i $2000.
    sta $2000
    lda #%00011110  ; Vi lagrer %00011110 i $2001.
    sta $2001       ; I guiden er det forklart hva det betyr.

    ; Nå er vi ferdige. PPU-en er nå gitt en bakgrunnsfarge, og den er stilt
    ; inn til å tegne et bilde. Dette vil den da gjenta 60 ganger i
    ; sekundet. Da er det ikke noe mer for prosessoren å gjøre, og den kan
    ; gå inn i en uendelig løkke:

Ferdig:
    jmp Ferdig
```

Her er det noen små ting som må gjøres rede for. Som du sikker ser brukes semikolon for å betegne kommentarer til koden. Alt som står bak semikolon ignoreres av ca65. Når det gjelder tall kan disse angis i forskjellige tallsystemer. De tallene det står $ foran er i sekstentallsystemet. Tallene med % foran er i totallsystemet. Tall uten noe slikt tegn foran er i titallsystemet. Tegnet # har forvirrende nok ikke noe med dette å gjøre. Det tegnet kan forekomme foran verdier gitt i alle tallsystemene og betydningen av det har med selve assemblykoden å gjøre. Uten at vi skal gå nærmere inn på det nå, betyr # at tallet skal tolkes som en *direkte verdi* mens tall uten # foran skal tolkes som adresser.

Strukturen til denne eksempelkoden er typisk for en NES-kodefil. Filen er delt opp i forskjellige *segmenter* (deler.) Segmentene er abstrakte inndelinger som hjelper oss i å holde orden på forskjellige deler av koden. Noen av segmentene representerer deler av den endelige `.nes`-filen, mens andre kun er en virtuell avgrensning (f.eks. RAM, som ikke er en del av koden.) Segmentene er definert i filen `linker_config.cfg`. I kapittel 5.1 skal vi se nærmere på hvordan segmentene er definert.

I `linker_config.cfg` til dette eksempelet er følgende segment definert:

- **INES**
  Hver **.nes**-fil har en såkalt iNES-header som forteller emulatoren informasjon om innholdet i filen. Husk at en **.nes**-fil representerer en komplett 'kopi' av en spillkassett, med grafikk, kode og så videre. Det finnes mange forskjellige typer kassetter med bl.a. varierende størrelse på ROM-chippene. Emulatorene leser altså iNES-headeren for å emulere riktig type spillkassett. Når vi programmerer NES-kode må vi oppgi hvilken type spillkassett vi ønsker at koden vår skal kjøre på.

- **VECTORS**
  Her ligger det tre verdier som CPU-en trenger, såkalte *adressevektorer*. Den første er en adresse til det stedet i PRG-ROM hvor CPU-en kan finne koden som skal kjøres dersom det skjer en såkalt *non-maskable interrupt* (NMI.) Den andre verdien er adressen til det stedet i PRG-ROM hvor koden vår starter. Dette er den adressen CPU-en begynner å kjøre fra når man slår på eller restarter NES-en. Den siste verdien er en adresse til koden som skal kjøres dersom det skjer en *interrupt request* (IRQ.) Dette kommer vi tilbake til i mer detalj senere. Helt i starten vil det være adresse nummer to – startadressen – vi vil være

mest interessert i.

- **DATA**
  Her lagrer vi data som skal benyttes av koden vår. Dette kan være alt fra tekststrenger, palettfargedata, brett-data (altså dataene som forteller hvordan brettene i spillet skal se ut), lyd og musikk, og så videre.

- **CODE**
  Her skriver vi selve koden, i assemblyspråket.

- **GFX**
  Her ligger alle grafikkdataene, tile for tile. Vanligvis vil det eneste vi legger i GFX-segmentet i kodefilen være eksterne grafikkfiler som er lagret i et av tegneprogrammene (se neste delkapittel.)

- **RAM**
  Her definerer vi variabelnavn, det vil si navn som representerer en adresse i NES-ens RAM.

La oss se nærmere på koden ovenfor. Det første vi kan merke oss er koden er at den, med unntak av noen få linjer, er indentert, det vil si at den er «tabbet in». Dette er ikke nødvendig, men det bidrar til å gjøre koden lettere å lese. I de linjene som ikke er tabulert inn, er det såkalte *labels*. En label kjennetegnes av et ord etterfulgt av et kolon. De fungerer som «merkelappe» i koden. Når koden kompileres byttes labelen ut med den *adressen* eller posisjonen i koden hvor labelen befinner seg.

Ovenfor her har vi f.eks. en label kalt `Start:`. Denne markerer at koden begynner på linjen under, og som vi skal se senere, benyttes så denne labelen til å fortelle prosessoren i NES-en hvor koden den skal kjøre begynner. Nederst er det også en label kalt `Ferdig:`. På linjen under er det så en instruksjon, `jmp Ferdig`. hvor dette navnet altså dukker opp igjen. Uten at vi skal feste oss for mye med det, er `jmp` (forkortelse for jump) en instruksjon som forteller prosessoren at den skal foreta et *hopp* i programflyten – den skal begynne å kjøre instruksjoner fra et annet sted. I dette tilfellet blir prosessoren bedt om å hoppe til første instruksjon under labelen `Ferdig`. I dette tilfellet vil den da havne på samme `jmp`-instruksjon om igjen. Effekten av dette blir at prosessoren kommer til å kjøre denne ene instruksjonen i en uendelig repetisjon. I eksempelet ovenfor er dette ønskelig, siden det ikke er noe mer vi vil at prosessoren skal gjøre etter at den har endret bakgrunnsfargen. Hvorfor det er nødvendig å holde prosessoren i aktivitet på denne måten skal vi se senere.

Det andre vi kan merke oss er de linjene som starter med et punktum. Disse er såkalte *direktiver* til assembleren. De er ikke i seg selv kode som prosessoren skal utføre – de er kode som ca65 skal utføre når den kompilerer koden. Ord som `.segment` og `.byte` er altså ord som dirigerer ca65 til å gjøre et eller annet. (For de leserne som har brukt C eller C++, er dette helt analogt med linjer som begynner med #, f.eks. `#include`.) Vi skal ta for oss hva forskjellige slike direktiv betyr etter hvert.

Vi ser også at koden er inndelt i forskjellige av de overnevnte segmentene. De første to linjene i koden er

```
.segment "INES"
```

```
    .byte "NES", $1A,1
```

Direktivet `.segment "INES"` forteller at de linjene som følger skal legges i **INES**-segmentet. Som forklart ovenfor, inneholder dette segmentet informasjon til NES-emulatorene om hvordan de skal emulere filen (hvilken type kasett det er, og så videre.) iNES-formatet er en standard som man har blitt enige om i NESdev-miljøet. Det består av 16 bytes, der de tre første er bokstavene NES, etterfulgt av tallet $1A. Deretter kommer selve informasjonen.[2] Vi skal ikke gå i detalj på dette, men heller ta ting som vi trenger etter hvert. Vi benytter direktivet `.byte` for å lagre bytene vi vil ha i iNES-headeren. Vi kan som vi ser enten skrive ord med hermetegn, slik som ovenfor, der hver bokstav i ordet lagres i en egen byte, eller vi kan skrive inn tallverdien til byten og skille de forskjellige bytene fra hverandre med komma.

Hva sier så iNES-headeren vi definerer i eksempelfilen? Som vi ser kommer de påkrevde fire bytene NES og $1A. Den neste byten antyder hvor stor PRG-ROM-chip kassetten vår har, der 1 er 16KiB, 2 er 32KiB, 3 er 48KiB og så videre. Vi trenger ikke mer enn 16KiB, så derfor legger vi inn verdien 1 her. De resterende 11 bytene er ikke oppgitt, så disse blir satt til 0. Dersom vi hadde hatt noe grafikk måtte vi oppgitt hvor stor CHR-ROM-chipen er ved å legge en tilsvarende verdi i den neste byten. Men her har vi ikke behov for noen CHR-ROM-chip, så det er greit at denne byten er 0.

De neste linjene i koden ser vi inneholder et nytt `.segment`-direktiv. Dette bytter segment til VECTORS-segmentet.

```
.segment "VECTORS"
    .word 0, Start, 0
```

Her settes de tre adressevektorene som ble nevnt i listen ovenfor. Koden i dette eksempelet er såpass enkel at den hverken trenger en NMI-adresse eller en IRQ-adresse. Derfor er disse satt til 0. Kun den midterste adressen trengs, for det er jo tross alt adressen til det stedet der prosessoren skal begynne å kjøre kode når den starter. I motsetning til i INES-segmentet så er verdiene her angitt med `.word`-direktivet i stedet for `.byte`. Det er fordi verdiene her må oppta 16 bits (altså to bytes) siden de er adresser. Akkurat som en 8-biters verdi kalles en byte så kalles en 16-bits verdi for et *word*.[3]

Når INES-headeren og adressevektorene er definert er det kun selve koden som gjenstår som helt nødvendig for å lage en komplett `.nes`-fil. Denne må ligge i CODE-segmentet. Selve kode-delen begynner derfor som vi ser med et `.segment "CODE"`-direktiv. Hva koden gjør skal vi forstå oss mer på i de neste kapitlene. Merk at vi ikke har lagt noen informasjon i DATA, GFX eller RAM-segmentene. Dette er rett og slett fordi koden vår verken har grafikk, data eller behov for å definere variabler i RAM. Dette er dog ganske uvanlig (vi får raskt bruk for alle tre), og det er altså derfor ld65 klager på dette når vi kompilerer eksempelet.

[2] `http://nesdev.parodius.com/neshdr20.txt`

[3] Strengt tatt varierer størrelsen til et *word* med hvilken prosessor vi har å gjøre med. Det har likevel stort sett blitt enighet om at et word vanligvis er 16 biter stort.

2.2 Grafikk og tegneprogrammer

Man kan lage grafikk til NES på flere måter. Hvis man har veldig god tid og liker å gjøre det vanskelig for seg selv, kan man til og med skrive inn grafikken i ren kode. For de litt siviliserte er det derimot bedre med et eget tegneprogram. Man *kan* bruke programmer som MS Paint eller GIMP, men problemet med disse er at de ikke er beregnet for de begrensningene som NES har. Når man arbeider med grafikk for NES må man jobbe med små tiles på 8x8 piksler. I tillegg er det begrenset hvor mange, og hvilke, farger man kan benytte seg av. Da er det greit med et tegneprogram som tar høyde for disse tingene. Det finnes flere slike programmer. Denne guiden kommer til å ta utgangspunkt i Java-programmet Tile Molester. Et alternativ for Windows-brukere er programmet Tile Layer Pro, som i grunn er et bedre program. Jeg har valgt Tile Molester siden det er mer portabelt.

2.2.1 Installasjon og bruk

Tile Molester lastes ned her: `http://nesdev.parodius.com/tilemolester-0.16.zip`.

Tile Molester er skrevet i Java, så det bør kjøre på de fleste platformer med en Java-runtime installert. Programmet kan virke noe buggy, i alle fall på Mac, men det gjør jobben sin og vel så det.

Lage en ny grafikkfil

Som vi så i kapittel 1 opererer NES-en med to former for grafiske elementer – bakgrunner og sprites. Det er lurt å holde disse to adskilt, også når man arbeider med grafikken. CHR-ROM er vanligvis delt opp slik at 4KiB (av 8KiB) er forbeholdt grafikk for bakgrunner, og 4KiB er forbeholdt grafikken til sprites. Vi ønsker å føye oss etter denne inndelingen, så når man lager en ny grafikkfil skal denne altså ha størrelse 4KiB, som er 4096 bytes.

For å lage en ny fil velger man menyen `File` og deretter valget `New...`. En dialogboks ber deg taste inn filstørrelsen. Dette skal være i bytes. Tast inn 4096 og trykk `OK`. Et nytt, svart vindu skal nå dukke opp, samt en ny meny til venstre med et utvalg av tegneverktøy.

Oppsett for NES

TIle Molester støtter mange forskjellige grafikkformater til en rekke forskjellige spillmaskiner. Grafikkfilene til NES er rene binærfiler uten noen form for informasjon om innholdet. Det betyr at hver gang man lager en ny eller åpner en eksisterende grafikkfil, må man velge riktig format i Tile Molester. Det er viktig å gjøre dette før man begynner å tegne.

For å gjøre dette må du først ha en fil åpen. Velg så menyen `View` og deretter `Codec`. Fra listen velger du så valget `2bpp, planar, composite`. Til slutt anbefales det sterkt at du velger `Tile grid` fra `View`-menyen. Da kan du se hver individuelle tile.

Selve bruken av Tile Molester bør være grei å komme inn i. Det fungerer i det store og det hele

som de fleste enklere tegneprogrammer som f.eks. MS Paint og bør ikke være noe problem å lære seg. Det du bør merke deg er at fargeutvalget nederst i vinduet **ikke** er representativt for fargene som vises på skjermen når grafikken kompileres i en `.nes`-fil. De fire fargene man kan velge mellom representerer forskjellige fargevalg i PPU-ens *palett*. Paletten er et fargeutvalg som man selv definerer i selve koden til programmet. Det er altså ikke nødvendig å endre på fargene i Tile Molester. Det har ingen som helst innvirkning!

Etter hvert som vi kommer til hvordan vi programmerer PPU-en, skal vi ta for oss mer i detalj hvordan vi går frem for å lage grafikken og hvordan denne samkjøres med koden. I mellomtiden kan du ta en kikk på eksempelfilene som følger med guiden. Se på hvordan grafikkfilene tar seg ut når du åpner dem i Tile Molester (husk å stille inn riktig format først) og hvordan grafikken ser ut når den tilhørende `.nes`-fila kjøres i en emulator.

2.3 Kjøring av programmer på en virkelig NES

Det aller kjekkeste er selvsagt å se sine egne kreasjoner kjøre på en vaskekte NES. For å gjøre det har man i grunn to valg:

- For de med lite elektronikkerfaring er det enklest å gå til anskaffelse av Retrousbs PowerPak. Dette er en spillkassett med en slot hvor man kan sette inn et flashkort. På flashkortet trenger man bare å legge inn .nes-filer, og når man slår på NES-en få man opp en liste over alle .nes-filer i filsystemet. Så og si alle mappere og RAM/ROM-konfigurasjoner støttes. Mer informasjon finnes her: `http://www.retrousb.com`.
- Elektronikkyndige som vil spare noen kroner kan ta en spillkassett og lodde av PRG-ROM- og CHR-ROM-chippene. Dette er EPROM-brikker av typen 27128 og 2764, som kan erstattes av pin-kompatible EEPROM-brikker. Innholdet i .chr- og .spr-filene legges på CHR-ROM-brikken, mens den assemblede koden legges på PRG-ROM-brikken. I `linker_config.cfg` må CHR-ROM og INES-headeren tas bort fra filen. Den blir da seende slik ut (med eventuelle ekstra tillegg, som vi skal se senere):

```
MEMORY {

# 24K PRG-ROM
    PRGROM: start = $8000, size = $4000, file = %O, fill = yes, define = yes;

# RAM
    RAM_SEG: start = $0000, size = $0800, define = yes;
}

SEGMENTS {
    CODE:       load = PRGROM, type = ro, define = yes;
    DATA:       load = PRGROM, type = ro, define = yes;
    VECTORS:    load = PRGROM, type = ro, define = yes, start = $bffa;
    RAM:    load = RAM_SEG, type = rw;
}
```

Kapittel 3

CPU-en

Det er nå på tide å komme skikkelig i gang. Det aller første vi må gjøre er å ta for oss CPU-en (prosessoren). Det er jo tross alt denne vi skal programmere, og da trenger vi en bedre forståelse av hvordan den fungerer.

3.1 Registrene

I forrige kapittel ble det kort forklart at *registrene* til prosessoren er helt sentrale når vi programmerer den. For å minne på hva de er; registrene er små lagringsplasser internt i selve prosessoren. Disse brukes til to formål: Av vi som programmerer for å blant annet gi prosessoren data som skal brukes i instruksjoner, og av prosessoren selv for å holde styr på *programflyten*. 6502-prosessoren har seks registre, der tre av dem kan brukes av oss direkte, mens de tre andre først og fremst kontrolleres av prosessoren. De seks registrene har fått navn **A**, **X**, **Y**, **PC**, **S** og **P**, og vi skal nå se nærmere på hvert av disse.

Akkumulatoren, A

A er et 8-bits register som kan brukes fritt av oss når vi programmerer. Det vil si at vi kan legge hvilken verdi vi ønsker i A, vi kan instruere prosessoren om å hente verdier fra et sted i adresseområdet og legge i A, og så videre. A-registeret har en spesiell rolle, for det er det eneste registeret som er koblet direkte sammen med den delen av prosessoren som gjør aritmetiske og logiske operasjoner (det vil si operasjoner som pluss og minus, og operasjoner som «and», «or», «xor» osv.) Hver gang prosessoren utfører en slik instruksjon vil resultatet legges i A.

Indeksregistrene, X og Y

X og Y er 8-bits registre som også kan brukes fritt når vi programmerer. Som A kan disse altså brukes for å hente og lagre verdier i adresseområdet, men de kan ikke brukes i aritmetiske eller logiske operasjoner. X og Y kan derimot brukes som *indeks* når vi skal hente en verdi fra et

sted i adresseområdet. De aller fleste er nok vant med at hvis vi har en liste, tabell (array) eller lignende med verdier der vi ønsker å utføre en operasjon på hvert element, kan dette gjøres ved å lage en såkalt *løkke* med en variabel som varierer fra start til slutt av listen. Dette kan vi bruke X- og Y-registrene til. I stedet for å instruere prosessoren om «hent verdien i adresse $200», kan vi instruere den til «hent verdien i adresse $200 + X».

Programtelleren, PC

PC («Program Counter») er et 16-bits register som angir i hvilken adresse den neste instruksjonen i programmet som står for tur befinner seg. Dette registeret kan vi naturligvis ikke brukes slik som A, X og Y, da prosessoren selv bruker dette registeret til å finne neste instruksjon hele tiden. Det eneste vi får lov til er å endre verdien. Det som skjer da er at vi forandrer programflyten – vi ber jo da prosessoren om å begynne å hente instruksjoner fra et annet sted i adresseområdet.

Stack-pekeren, S

S er et 8-bits register som har en litt spesiell rolle som vi skal gå mer i dybden på senere (se avsnitt 3.3.1). Kort sagt inneholder dette en 8-bits adresse til hvor neste ledige del av *stacken* er. En stack (stabel på norsk) er en spesiell datastruktur hvor vi kan «stable» verdier oppå hverandre. Når vi legger noe på stacken sier vi at vi *push*er det på stacken, og når vi tar noe av stacken så sier vi at *pop*er det av stacken. Når vi pusher noe på stacken så legges det øverst på stabelen. Når vi poper noe av, får vi tilbake det øverste på stacken. S-registeret brukes da til å holde på adressen til «toppen» av stacken, det vil si til neste ledige plass.

Statusregisteret, P

P («Processor Status») inneholder *statusflaggene* til prosessoren. Et flagg er en én-bits verdi som vi tenker på som «ja» og «nei». Det er 7 forskjellige statusflagg, som hver har sine roller (se under). Hvert individuelle flagg skal vi komme tilbake til, men for å ta et eksempel kan det for eksempel hende at prosessoren utfører pluss-instruksjonen (som legger et tall til verdien i A), men så blir resultatet større enn 8 bits, slik at ikke alt får plass i A. Dette kan eksempelvis skje når vi legger sammen tallet 255 og 1; resulatet er tallet 100000000_2, som tar 9 bits. Det som skjer da er at et av statusflaggene, nemlig carry-flagget («mente»-flagget), settes til 1.

Følgende statusflagg finnes i P:

Bit	Flagg	Navn	Beskrivelse
0	C	Carry	Holder en bit i mente ved addisjon og en lånebit ved subtraksjon. Indikerer om en addisjon har resultert i et tall over 255 eller en subtraksjon har resultert i et tall under 0 (se avsnitt 3.2.3)
1	Z	Zero	Indikerer om forrige instruksjon resulterte i 0
2	I	Interrupt disable	Brukes for å aktivere/deaktivere IRQ-avbrudd (se avsnitt 3.5)
3	D	Decimal	Ubrukt i NES-prosessoren
4	B	Break	Indikerer om det har skjedd et programvareavbrudd eller ikke (se avsnitt 3.5)
5	-	-	Ubrukt bit
6	V	Overflow	Indikerer om forrige regneoperasjon resulterte i en *overflyt* (se avsnitt 3.2.3)
7	N	Negative	Indikerer om forrige instruksjon resulterte i et negativt tall (se avsnitt 3.2.3)

De viktigste flaggene for oss vil være C, Z, N og en sjelden gang V. Dette er flagg som settes basert på resultatet fra forrige instruksjon. Ikke alle instruksjoner påvirker flaggene, men de som på en eller annen måte gir et resultat (for eksempel aritmetiske operasjoner, bitoperasjoner, henting fra minnet og så videre) gjør det. Poenget med flaggene er at noen av instruksjonene kan bruke informasjonen i disse. Noen instruksjoner kan for eksempel endre programflyten til CPU-en basert på om et flagg er satt eller ikke.

Flaggene I, D og B er mer spesielle. Disse har ikke noe med forgående instruksjon å gjøre slik som de andre har. D-flagget er litt spesielt i NES-sammenheng, da det faktisk ikke har noen som helst funksjon. En ordinær 6502-prosessor har mulighet for å regne med tall i et spesielt *binary coded decimal* (BCD)-format. I dette formatet er hver byte delt inn i to deler på 4 bits som representerer hver sine sifre. Tallet %10010111 er for eksempel tallet 97 på BCD-form, siden $1001_2 = 9$ og $0111_2 = 7$. Når CPU-en er i *decimal mode* kan den regne korrekt med slike tall (slik at $15 + $05 blir $20 i stedet for $1A og så videre). I NES-prosessoren er delen av kjernen som tar seg av dette rett og slett tatt bort [1], men flagget er fortsatt igjen.

[1] Det spekuleres i hvorfor. Mest sannsynlig ble decimal mode, som var en patentert teknologi, fjernet fra 6502 for at Nintendo ikke skulle bli saksøkt av Commodore (som på den tiden hadde kjøpt opp MOS Technology som designet 6502) når de lanserte NES i USA.

3.2 Instruksjoner

Nå skal vi se på de viktigste instruksjonene vi kan gi prosessoren. Det finnes flere instruksjoner en de som blir tatt opp her, men de skal vi ta etter hvert som vi får bruk for dem senere. For en komplett liste over instruksjoner kan jeg anbefale følgende nettside: `http://www.obelisk.demon.co.uk/6502/instructions.html`.

En instruksjon består av to deler: en *opcode* og et eventuelt *argument*. Opcoden er en byte som forteller prosessoren hvilken operasjon den skal utføre, og hvordan et eventuelt argument skal tolkes. Argumentent inneholder data som prosessoren trenger for å utføre operasjonen, og tar en eller to bytes. En instruksjon kan altså for eksempel være denne tallrekken: `$AE $30 $20`. Denne tallrekken vil alle 6502-prosessorer tolke som «ta verdien i minneadressen $2030 og legg den i register X». Problemet for oss mennesker er at denne tallrekken ikke gir like mye mening for oss. Av nettopp denne grunnen har man utviklet det som kalles *assemblyspråk*. Dette er en type programmeringsspråk der man skriver instruksjoner med ord, og så omsettes disse av en *assembler* til slike tallrekker som ovenfor. I assemblykode vil for eksempel tallrekken ovenfor skrives «LDX $2030». Alle prosessorfamilier har sine egne assemblyspråk, og vi skal arbeide med assemblyspråket for 6502-prosessoren når vi programmerer NES-spill.

I 6502-assembly representeres instruksjonene med korte ord på tre bokstaver, etterfulgt av eventuelle argumenter på en mer forståelig form. Ordene er forkortelser av det instruksjonene gjør. For eksempel har vi som vi snart skal se en instruksjon som legger en verdi inn i A-registeret. Denne heter `LDA`, kort for «`LoaD A`». Da kan du kanskje gjette hva de tilsvarende instruksjonene for å laste en verdi inn i X eller Y heter?

Prosessoren gjør ikke annet enn å hente instruksjoner fra adresseområdet og kjøre dem, i en endeløs løkke. Når prosessoren er ferdig med en instruksjon, øker den adressen i PC-registeret med lengden av instruksjonen («LDX $2030» tok for eksempel 3 bytes, som vi så ovenfor), og så henter den en ny instruksjon fra den nye adressen i PC-registeret. Derfor er det viktig at instruksjonene ligger sekvensielt etter hverandre. Uansett hva som ligger i adressen som PC-registeret peker til vil prosessoren tolke som en opcode og prøve å utføre instruksjonen som den representerer. Vi skal se senere hvordan vi kan endre adressen der prosessoren henter instruksjoner fra slik at vi unngår at den prøver å kjøre for eksempel data som kode.

Etter at en instruksjon er ferdig oppdaterer prosessoren flaggene i statusregisteret, P, basert på resultatet fra instruksjonen. Hvis instruksjonen resulterte i 0 vil for eksempel Z-flagget settes til 1 for å indikere dette.

3.2.1 Argumenttyper og adresseringsmåter

De fleste instruksjonene finnes i flere varianter (her med sin opcode), der forskjellen mellom disse variantene er hvordan prosessoren håndterer eventuelle argumenter. Et argument kan for eksempel brukes direkte i operasjonen, eller det kan referere til et sted i adresseområdet hvor dataene kan hentes, eller resultatet av instruksjonen skal lagres. 6502-prosessoren har i tillegg flere måter å *lage* adresser på. Noen instruksjoner kan eksempelvis benytte seg av såkalt *indeksert adressering*. Da lages den endelige adressen ved å legge verdien av et av indeksregistrene (X eller Y) til det oppgitte argumentet.

Følgende adresseringsmåter finnes:

Adresseringsmåte	**ca65-syntaks**	**Beskrivelse**
Implisitt	`SEC`	Det er ingen argument
Akkumulator	`LSR A`	Verdien i A-registeret (akkumulatoren) brukes i instruksjonen
Direkte verdi	`LDA #12`	Argumentet brukes direkte i instruksjonen
Absolutt adressering	`LDA $1234`	Argumentet peker til et sted i adresseområdet
Absolutt, indeksert med X	`LDA $1234, X`	Argumentet legges sammen med verdien av X-registeret for å gi den endelige adressen
Absolutt, indeksert med Y	`LDA $1234, Y`	Argumentet legges sammen med verdien av Y-registeret for å gi den endelige adressen
Indirekte	`JMP ($1234)`	Argumentet brukes som en adresse til å hente en *ny* adresse fra adresseområdet som da blir de endelige adressen. `JMP` er den eneste instruksjonen som har denne adresseringsmåten.
Indirekte, indeksert	`LDA ($12), Y`	Argumentet brukes som en indeks i *zero page* (se avsnitt 3.3) til å hente en *ny* 16-bits adresse, og verdien av Y legges til denne for å gi den endelige adressen
Indeksert, indirekte	`LDA ($12, X)`	X legges til argumentet, og resultatet brukes som en indeks i zero page (se avsnitt 3.3) til å hente en *ny* 16-bits adresse
Relativ	`BEQ $12`	Argumentet er et positivt eller negativt tall (fra -128 til 127) som angir hvor mange bytes fremover eller bakover den endelige adressen ligger i forhold til adressen der instruksjonen forekommer. Brukes kun av betingede hoppeinstruksjoner

De tre siste her kan nok virke noe kompliserte. Vi skal komme tilbake til dem senere. De tre øverste vil vi derimot få bruk for ganske fort.

Labels

Det er håpløst å prøve å huske adressene til hver bidige ting i programmet vårt. Som i andre språk er det nyttig å gi variabler og forskjellige deler av koden navn. Dette gjør vi ved å definere såkalte *labels*. Som vi så i forrige kapittel er disse på en måte som «merkelapper» i koden. Når vi definerer en label, forbindes navnet med adressen til det stedet der labelen er definert. Etter at labelen er definert kan vi benytte navnet dens når vi ønsker å referere til den aktuelle minneadressen. Vi definerer en label ved å skrive det ønskede navnet etterfulgt av `:`. La oss se på et eksempel:

```
.segment "RAM"
    variabel1: .byte 0
    variabel2: .byte 0
    variabel3: .byte 0

.segment "CODE"
    ; ...
    LDA variabel3
```

Her definerer vi tre labeler. Den første, `variabel1` vil korrespondere med adressen 0, den neste vil korrespondere med adressen 1, og så videre. For å repetere, direktivet `.byte 0` forteller at det skal settes av én byte. Når ca65 kommer over instruksjonen `LDA variabel3` vil den søke etter labeldefinisjonen og bytte labelen ut med adressen den svarer til. Fordelen med dette er at vi kan legge definisjonen av `variabel3` et helt annet sted (eller vi kan definere flere variabler foran `variabel3`), uten at vi må endre alle instruksjonene som involverer denne variabelen.

Vi skal også se at labeler brukes når vi utfører såkalte *hopp* i koden (se under.) Da oppgir vi labelen til det stedet i koden som vi ønsker at prosessoren skal hoppe til.

3.2.2 Henting og lagring

De viktigste instruksjonene er de som gjør oss i stand til å sette verdier inn i et register, flytte verdier mellom registrene, og lagre registerverdier i adresseområdet.

Instruksjon	Navn
LDA	LoaD A
LDX	LoaD X
LDY	LoaD Y
STA	STore A
STX	STore X
STY	STore Y
TAX	Transfer A to X
TAY	Transfer A to Y
TXA	Transfer X to A
TYA	Transfer Y to A
TXS	Transfer X to S
TSX	Transfer S to X

For å sette en verdi inn i et register bruker vi instruksjonene `LDA`, `LDX` og `LDY`. Disse setter en verdi inn i henholdsvis A, X og Y-registrene. La oss se på noen eksempler:

```
LDA #123        ; Sett verdien 123 inn i A
LDX $2000       ; Sett verdien som ligger i adresse $2000 inn i X
LDA $2000, X    ; Sett verdien som ligger i adresse $2000 + X inn i A
```

`LD`-instruksjonene kan benytte seg av de fleste adresseringsmåtene som er listet opp i tabellen ovenfor.

Tilsvarende har vi instruksjonene `STA`, `STX` og `STY` som lagrer verdiene i A, X og Y i adresseområdet.

Noen ganger kan det være gunstig å flytte verdier mellom registrene. Til dette har 6502-prosessoren fire instruksjoner: `TAX`, `TXA`, og de tilsvarende `TAY` og `TYA`. Disse fungerer som navnene tilsier ved at verdien i det første registeret i instruksjonsforkortelsen settes inn i det andre registeret. Verdien i det første registeret vil forbli uforandret. Noen eksempler:

```
LDA #2          ; Sett tallet 2 inn i A
TAX             ; Flytt verdien i A inn i X, dvs. at X får verdien 2
LDY #3          ; Sett tallet 3 inn i Y
TYA             ; Flytt verdien i Y inn i A
```

Merkelig nok finnes det ingen TXY- eller TYX-instruksjon. Det betyr at for å flytte verdier mellom X- og Y-registrene må man enten ta i bruk RAM ved å midlertidig lagre verdien i det ene registeret i RAM og deretter hente den inn til det andre registeret, eller benytte A-registeret som et mellomlager:

```
STY $100      ; Lagre verdien i Y-registeret i adressen $100 (RAM)
LDX $100      ; Hent verdien i adresse $100 inn til X

; Eller:
TYA           ; Sett verdien i Y-registeret inn i A-registeret
TAX           ; Sett verdien i A-registeret inn i X-registeret
```

Hvilken av disse som er mest hensiktsmessig kommer an på situasjonen. Noen ganger trenger vi å beholde verdien som allerede ligger i A-registeret. Da velger vi den øverste. Ellers vil den nederste være kjappest å utføre.

Vi har også noen instruksjoner for å manipulere noen av flaggene i P-registeret:

Instruksjon	Navn
SEC	SEt Carry
CLC	CLear Carry
SEI	SEt Interrupt
CLI	CLear Interrupt
CLV	CLear oVerflow

Disse brukes altså til å henholdvis sette (flagg-biten blir 1) eller fjerne (flagg-biten blir 0) C-, I- og V-flaggene (V kan bare fjernes).

3.2.3 Tallregning

6502-prosessoren har en nokså primitiv regneenhet innebygd. Denne benyttes kan utføre addisjon og subtraksjon, og vi kan be den gjøre forskjellige operasjoner med følgende instruksjoner:

Instruksjon	Navn
ADC	ADd with Carry
SBC	SuBtract with Carry
INX	INcrement X
DEX	DEcrement X
INY	INcrement Y
DEY	DEcrement Y
INC	INcrement
DEC	DEcrement

Addisjon (ADC)

Instruksjonen ADC legger sammen verdien som ligger i A og argumentverdien, i tillegg til carry-flagget. Resultatet legges i A. Carry er det vi på norsk kaller «mente». Som nevnt i forrige seksjon består statusregisteret til proessoren av en rekke biter som kalles flagg, som brukes til å holde styr på forskjellige ting. Et av disse er det såkalte carry-flagget, som altså kan holde én bit i mente. Formålet med dette er å gjøre oss i stand til å legge sammen tall som er større enn det A-registeret kan ta i én omgang. Hvis A for eksempel inneholder 255 og så legger vi til 3, så får vi problemer. Resultatet blir 258, et tall som tar opp 9 biter, og det er åpenbart ikke plass til hele tallet i A. Når dette skjer, kalles det en *overflow*. Det som vil skje da er at de 8 *laveste* bitene (de som er på de 8 laveste plassene) lagres i A, og så settes den niende biten inn i carry-flagget

i statusregisteret. Med eksempeltallet 258 har vi at 258 = %100000010. Den høyeste biten – biten til venstre – settes inn i carry-flagget, så det får nå verdien 1. De resterende 8 bitene lagres i A-registeret, så A får nå verdien %00000010 = 2.

La oss se på et eksempel som tar to 8-bits tall i fra RAM og legger disse sammen. La oss si at tallene ligger i variablene 'tall1' og 'tall2' (det vil si adressene med labelene 'tall1' og 'tall2') og at vi vil lagre resultatet i variabelen 'resultat'.

```
LDA tall1      ; hent inn det ene tallet
CLC            ; sett carry-flagget til 0
ADC tall2      ; legg til det andre tallet
STA resultat   ; lagre i 'resultat'
```

Denne koden har en svakhet. Dersom summen av tall1 og tall2 er større enn 255 vil resultatet bli galt – det vil bare inneholde de 8 laveste bitene. For å lagre resultatet trenger vi *to* bytes. Den korrekte koden ser slik ut:

```
LDA tall1         ; hent inn det ene tallet
CLC               ; sett carry-flagget til 0
ADC tall2         ; legg til det andre tallet
STA resultat      ; lagre i 'resultat'
LDA #0            ; fjern talleti A
ADC #0            ; legg sammen A med 0 (og carry-flagget!)
STA resultat+1    ; lagre i byten som kommer etter 'resultat' i minnet
```

Denne koden må vi gå nærmere i sømmene. Først gjør vi det samme som sist. Det som lagres i 'resultat' vil være de 8 laveste bitene i resultatet. Det koden under gjør er å ta hånd om et eventuelt carry-flagg, altså en bit i mente. Dette blir da den niende biten i resultatet. Denne ønsker vi å lagre i byten som kommer etter 'resultat' i minnet. Adressen til den byten er da `resultat + 1`. For å gjøre dette må vi få carry-biten over i A. Det gjør vi ved å først sette A til 0, og deretter å legge 0 + carry-flagget til A. Da vil A ha enten verdien 0 eller 1, basert på hva carry-flagget er.

Til slutt kan vi se på hvordan vi kan legge sammen to 16-bits tall og lagre resultatet i et nytt 16-bits tall. Det er ikke mye vi trenger å forandre koden for å gjøre det:

```
LDA tall1         ; hent inn de første 8 bitene av det ene tallet
CLC               ; sett carry-flagget til 0
ADC tall2         ; legg til de første 8 bitene av det andre tallet
STA resultat      ; lagre A i de første 8 bitene av resultatet
LDA tall1+1       ; hent inn de neste 8 bitene av det første tallet
ADC tall2+1       ; legg sammen A med de neste 8 bitene av det andre tallet
                  ; (og carry-flagget!)
STA resultat+1    ; lagre i byten som kommer etter 'resultat' i minnet
```

Vi tar altså og legger sammen de to første bytene av hvert tall. Resultatet er den første byten av summen. Deretter legges den neste byten av hvert tall sammen, i tillegg til en eventuell carry-verdi fra den første addisjonen. Resultatet blir da den andre byten av summen. Hvis vi vil kan vi benytte akkurat samme prinsipp til å legge sammen så store tall vi egentlig vil!

Subtraksjon (SBC)

Subtraksjon foregår rimelig likt som addisjon. Den eneste forskjellen i bruk er at vi må sette carry-flagget til 1 i stedet for 0 før vi utfører en `SBC` (når vi subtraherer fungerer carry-biten som lånetall i stedet for minnetall):

```
LDA tall1
SEC             ; sett C til 1
SBC tall2
```

En 16-bits subtraksjon utfører vi på samme måte som addisjonen ovenfor, men da med `SEC` i stedet for `CLC` før første subtraksjon.

Inkrementering og dekrementering (INX, DEX)

Instruksjonene `INX`, `INY` og `INC` *inkrementerer* (øker med 1) verdien i henholdsvis X, Y og den oppgitte adressen. Instruksjonene `DEX`, `DEY` og `DEC` *dekrementerer* (senker med 1) verdien i henholdsvis X, Y og den oppgitte adressen. Når verdien er $FF og den inkrementeres, vil den nye verdien bli 0. Når verdien er 0 og den dekrementeres, blir den nye verdien $FF.

Negative tall

I utgangspunktet kan de 8 bitene i registrene lagre verdier mellom 0 og 255, men i noen tilfeller kan det være greit å ha negative tall. 6502-prosesoren skiller *ikke* mellom positive og negative tall når den utfører instruksjoner, så det er opp til koden vår å *tolke* tall som positive eller negative.

Den beste måten å representere negative tall på er på *toerkomplementsform*. På denne formen representerer tallene fra 0 til $7F (127) de positive tallene fra 0 til $7F, mens tallene fra $FF og ned til $80 representerer tallene fra -1 og ned til -128. Alle tallene fra 0 til $7F har 0 i den høyste biten, mens tallene fra $80 til $FF har 1 i den høyeste biten. Det gjør det veldig enkelt å skille mellom positive og negative tall. Det kan virke litt merkelig at de negative tallene går «motsatt vei», men la oss se litt nærmere på dette: Hvis vi legger sammen 1 og $FF, og ser bort fra carry-flagget, får vi 0 i de 8 laveste bitene. Det betyr at det gir mening å tenke på tallet $FF som -1, siden vi får 0 når vi legger det sammen med 1. På samme måte kan vi tenke på $FE som -2, siden $FE + $02 gir 0, og så videre.

En fordel med toerkomplementsform er at instruksjonene `ADC` og `SBC` fungerer slik vi ønsker. Hvis A inneholder $F0 (-16 på toerkomplementsform) og vi utfører `ADC $F1` (-15 på toerkomplementsform) blir resultatet `E1` (med carry). Det er -31 på toerkomplementsform.

6502-prosessoren legger opp til denne måten å representere negative tall på. Hver gang en instruksjon resulterer i et tall der bit 7 er satt, vil N-flagget settes til 1 for å indikere dette. Da gir det kanskje mening at dette flagget heter «Negative flag», og at de tilhørende branch-instruksjonene heter `BMI` («Branch if Minus») og `BPL` («Branch if Plus»). Vi kan se et eksempel hvor N-flagget brukes:

```
LDA tall        ; LDA setter N til 1 dersom tallet er negativt

BMI negativt    ; BMI hopper hvis N = 1
    ; kode som kjører hvis tallet er positivt
JMP slutt
negativt:
    ; kode som kjører hvis tallet er negativt
slutt:
```

Et annet flagg som er relatert til tall på toerkomplementsform er V («Overflow»)-flagget. Dette settes til 1 dersom en addisjon eller subtraksjon resulterer i en såkalt overflyt (overflow). Hvis vi for eksempel har $7F (127) i A-registeret og legger til 1 så får vi plutselig det negative tallet $80 (-128), det har da skjedd en overflyt, og resultatet har blitt et tall med feil fortegn. Da settes V til 1 for å indikere dette. Det samme skjer for eksempel om vi har $90 (-112) og trekker fra $14 (20). Resultatet blir $7C, som er 125, i stedet for -132 som det skulle ha blitt om vi hadde flere bits til rådighet.

Hvis vi vil ha ut selve tallverdien til et negativt tall i «normal» form (dvs. mellom 0 og $7F i stedet for mellom $80 og $FF), kan vi enkelt og greit bare trekke tallet fra 0:

```
LDA #0
SEC
SBC tall      ; variabelen tall holder tallet vi skal konvertere
```

3.2.4 Bitoperasjoner

I tillegg til de aritmetiske instruksjonene har 6502-prosessoren også noen instruksjoner for å manipulere de individuelle bitene i et tall. Som med de aritmetiske instruksjonene er det A-registeret som er involvert i disse.

Logiske operasjoner (AND, ORA, EOR, BIT)

Den kan utføre de tre mest brukte logiske operasjonene; **AND**, **OR** og **XOR**. For å minne på det, så virker disse slik:

AND		
A	B	Resultat
0	0	0
1	0	0
0	1	0
1	1	1

OR		
A	B	Resultat
0	0	0
1	0	1
0	1	1
1	1	1

XOR		
A	B	Resultat
0	0	0
1	0	1
0	1	1
1	1	0

Prosessoren har følgende instruksjoner for å gjøre dette:

Instruksjon	Navn
AND	AND with A
ORA	OR with A
EOR	Exclusive OR with A
BIT	BIT-test with A

De tre første instruksjonene utfører henholdsvis AND, OR og XOR mellom tallet i A og argumentet, og lagrer resultatet i A. Et eksempel:

```
LDA #27    ; 27 = %00011011
AND #231   ; 231 = %11100111

; Nå vil A inneholde %00000011 = 3 siden bare de to siste bitene er 1 i 27 og 231.
```

Den siste instruksjonen, BIT, utfører en AND uten å lagre resultatet i A. Det kan være nyttig når vi bare er interessert i flaggendringene i P-registeret.

Bitshifting (ASL, LSR, ROL, ROR)

Prosessoren kan *skyve* og *rotere* bitene i A-registeret. Det vil si at alle bitene flyttes enten mot høyre eller mot venstre. Det som skiller skyving fra rotering er hva som skjer med biten som flyttes ut, og hva som fylles i den ledige biten som oppstår.

Instruksjon	Navn
ASL	Arithmetic Shift Left
LSR	Logical Shift Right
ROL	ROtate Left
ROR	ROtate Right

Ved skyving (ASL og LSR) får den «ledige» biten lengst til henholdsvis venstre eller høyre verdien 0. Dette har flere bruksområder, men først og fremst brukes det til å gange eller dele med 2. Hvis vi har et binært tall og skyver alle bitene et hakk mot venstre ser vi at resultatet blir at vi ganger med 2, siden hvert binære siffer flyttes til en plass med dobbelt så høy verdi. For eksempel er %101 = 5, mens %1010 = 10. På samme måte vil forskyving mot høyre føre til at vi deler tallet på 2.

Ved rotering (ROL og ROR) skyves verdien i carry-flagget inn i den ledige biten, mens biten i motsatt ende skyves inn i carry-flagget. Ved gjentatte roteringer vil da bitene som forsvinner i den ene enden dukke opp i motsatt ende. Et eksempel viser best hvordan dette fungerer:

```
LDA %10001101
CLC       ; fjern carry-biten
ROL A     ; %00011010
ROL A     ; %00110101
```

```
ROL A     ; %01101010
ROL A     ; %11010100
```

3.2.5 Sammenligninger og programflyt

En sentral del av alle program er å ta *avgjørelser*. Når vi programmerer for 6502-prosessoren skjer dette ved at vi (som regel) utfører en *sammenligning*. Deretter utfører vi en intruksjon som *endrer programflyten* til programmet. Vi kan f.eks. sammenligne et register med et tall. Hvis registerets verdi er større enn dette tallet, ønsker vi kanskje å gjøre én ting, mens vi vil gjøre noe annet hvis verdien er mindre. Prosessoren har instruksjoner for å utføre sammenligninger, og instruksjoner som endrer programflyten ved at den starter å utføre instruksjoner fra et annet sted i adresseområdet.

Til sammen har vi følgende instruksjoner:

Instruksjon	Navn
CMP	CoMPare with A
CPX	ComPare with X
CPY	ComPare with Y
BEQ	Branch if EQual
BNE	Branch if Not Equal
BCS	Branch if Carry Set
BCC	Branch if Carry Clear
BMI	Branch if Negative
BPL	Branch if Plus / Positive
BVS	Branch if OVerflow Set
BVC	Branch if OVerflow Clear

De viktigste instruksjonene for oss vil være `CMP`, `CPX`, `CPY`, `BEQ`, `BNE`, `BCS` og `BMI`. De andre instruksjonene kan være hendige, men det er veldig sjeldent de brukes. La oss se nærmere på de viktigste.

Sammenligningsinstruksjoner (CMP, CPX, CPY)

Disse tre instruksjonene er *sammenligningsinstruksjoner*. De sammenligner henholdsvis verdien i register A, X eller Y med argumentverdien til instruksjonen. En hver sammenligning av to tall har to utfall: enten er det ene tallet større enn det andre, eller så er de like. Etter hver sammenligningsintruksjon lagrer prosessoren resultatet av sammenligningen i statusregisteret P. Vi skal snart se at dette er for at andre instruksjoner skal kunne benytte seg av denne informasjonen.

```
CPY #12       ; Sammenlign verdien i Y med 12
CPX $1234     ; Sammenlign verdien i X med verdien i addresse $1234
CMP ($1234)   ; Sammenlign verdien i A med verdien i den indirekte addressen $1234
```

Dersom sammenligningen gir at de to verdiene er like settes Z-flagget til 1. Dersom den gir at registerverdien er *større enn eller lik* argumentverdien, settes C-flagget til 1.

Betingede hopp (BEQ, BNE, BCS, BCC, BMI, BPL, BVS, BVC)

Alle disse instruksjonene er såkalte «hoppeinstruksjoner» («branch instructions»). De får prosessoren til å foreta et hopp i programkoden – den begynner å hente intruksjoner fra et annet sted i addresseområdet. Argumentet til en slik instruksjon er hvor langt fremover eller hvor langt bakover prosessoren skal hoppe. Argumentet er et 8-bits tall, så det kan maksimalt ha verdier mellom -128 og 127. Det vil si at en betinget hoppeinstruksjon bare kan hoppe maksimalt 128 bytes bakover og 127 bytes fremover. Dette er det viktig å huske på: Vi kan ikke hoppe ubegrenset langt i koden med en betinget hoppeinstruksjon.

I praksis er det ikke så tungvint at vi telle og huske på hvor mange bytes prosessoren skal hoppe over – vi kan få assembleren ca65 til å ta seg av det ved at vi oppgir *labelen* til det stedet i koden vi ønsker at den prosessoren skal hoppe til. [2] Det som skiller de forskjellige hoppeinstruksjonene er på hvilket grunnlag hoppet skjer eller ikke. Instruksjonene BEQ og BNE har med statusen til Z (Zero)-flagget å gjøre. BEQ foretar programhoppet dersom flagget er 1. BNE foretar hoppet dersom det er 0. Husk fra forrige avsnitt at Z-flagget settes til 0 dersom en sammenligning resulterer i at de to verdiene er like. Da gir det kanskje mening at BEQ («Branch if Equal») foretar hoppet dersom Z er 1, og at BNE («Branch if Not Equal») foretar hoppet dersom Z er 0?

I første omgang vil det være de nevnte BEQ og BNE-instruksjonene som er av interesse for oss. De bruker vi som sett til å hoppe i koden basert på om de sammenlignede verdiene er like eller ikke. Som et eksempel kan vi tenke oss at vi har en figur på skjermen, og vi ønsker å finne ut om denne befinner seg helt til høyre (x-koordinaten er 248) eller helt til venstre (x-koordinaten er 0). Basert på om den befinner seg til høyre eller venstre ønsker vi å gjøre forskjellige ting. Det kan vi gjøre slik:

```
    LDA x_koordinat  ; Hent en verdi til A, fra stedet i adresseområdet med
                     ; labelen 'x_koordinat'
    CMP #0           ; Sammenlign A med 0
    BEQ venstre      ; Hvis de er like, hopp til addressen med labelen 'venstre'
    CMP #248         ; Sammenlign A med 248
    BEQ hoyre        ; Hvis de er like, hopp til adressen med labelen 'hoyre'

    ; Her er kode som kjøres dersom verdien i A ikke er 0 eller 248

venstre:
    ; Her er kode som kjøres dersom verdien i A er 0

hoyre:
    ; Her er kode som kjøres dersom verdien i A er 248
```

De forskjellige navnene x_koordinat, venstre og hoyre er som du kanskje husker fra tidligere, såkalte *labels* – merkelapper i koden. Disse bruker vi som sagt for å slippe å huske på adresser eller hvor mange bytes prosessoren må hoppe frem eller tilbake. Når ca65 assembler koden, vil

[2] Dette er også en av grunnene til at vi benytter en assembler og ikke skriver instruksjonene for hånd.

den selv telle hvor mange bytes det f.eks. er mellom instruksjonen `BEQ hoyre` og labelen `hoyre`.

Ubetinget hopp (JMP)

Noen ganger er det nødvendig å foreta et hopp i koden uten at det er noen betingelse som skal være oppfylt. Til det har vi den enkle instruksjonen `JMP` («jump»). Argumentet til denne instruksjonen er den adressen som prosessoren skal fortsette å hente instruksjoner fra. Det betyr at en JMP, i motsetning til de betingede hoppeinstruksjonene, kan hoppe til hvor som helst i adresseområdet.

Det er et problem med det forrige eksempelet. Hva skjer dersom x-koordinaten verken er 0 eller 248? Etter at instruksjonene der det står «Her er kode som kjøres dersom verdien i A ikke er 0 eller 248» er kjørt, vil prosessoren havne der labelen `venstre` er, og kjøre koden som står der (husk at prosessoren slavisk henter instruksjoner og kjører dem!) For å forhindre dette er `JMP`-instruksjonen akkurat det vi trenger. Et korrekt eksempel vil være slik:

```
    LDA x_koordinat  ; Hent en verdi til A, fra stedet i adresseområdet med
                     ; labelen 'x_koordinat'
    CMP #0           ; Sammenlign A med 0
    BEQ venstre      ; Hvis de er like, hopp til addressen med labelen 'venstre'
    CMP #248         ; Sammenlign A med 248
    BEQ hoyre        ; Hvis de er like, hopp til adressen med labelen 'hoyre'

    ; Her er kode som kjøres dersom verdien i A ikke er 0 eller 248
    JMP fortsett     ; hopp ned til labelen 'fortsett'

venstre:
    ; Her er kode som kjøres dersom verdien i A er 0
    JMP fortsett     ; hopp ned til labelen 'fortsett'

hoyre:
    ; Her er kode som kjøres dersom verdien i A er 248
    ; Her trengs ingen JMP, siden 'fortsett' kommer rett under

fortsett:
    ; Her fortsetter eventuelt koden
```

3.3 Adresseområdet

Som nevnt består adresseområdet av alle adresser mellom $0000 og $FFFF. Alle komponenter som CPU-en skal ha kommunikasjon med må tilordnes adresser mellom disse verdiene. I NES er adresseområdet lagt opp slik:

Adresser	Beskrivelse
$0000 - $07FF	RAM
$0800 - $1FFF	Tre speilinger av RAM
$2000 - $2007	PPU-registre
$2008 - $3FFF	Gjentatte kopier av PPU-registrene
$4000 - $4017	APU-, kontroller-, og DMA-registre
$4018 - $7FFF	Frie adresser som kan benyttes av spillkassetten
$8000 - $FFFF	PRG-ROM

For å lagre noe i RAM skriver vi altså til en adresse mellom $0000 og $07FF. [3]

Skal vi kommunisere med PPU-en eller APU-en må vi skrive til adressene $2000 til $2007 og $4000 til $4017. Disse er eksempler på det som kalles *adresseregistre*. I motsetning til RAM og ROM der vi enten lagrer data eller leser data, representerer disse adressene interne registre i komponenten de tilhører. Noen adresseregistre kan f.eks. være såkalte statusregistre som er slik at når vi leser fra dem, får vi en oversikt over hvilken tilstand komponenten er i. PPU-en har f.eks. et statusregister (adresse $2002). Ved å lese fra dette får vi ut informasjon om hvorvidt PPU-en er i vblank, om det har skjedd en såkalt *sprite 0-kollisjon*, og mer. Andre adresseregistre kan være konfigurasjonsregistre som benyttes til å konfigurerer komponenten.

Området fra $4018 til $7FFF er fritt. Her kan hver spillkassett koble disse adressene fritt til nær sagt hva det måtte være. Spillkassetter bruker dette typisk til batteridrevet RAM til å lagre saves i, eller til å kommunisere med såkalte *mappere*. En mapper er en brikke som utvider funksjonaliteten til NES-en, først og fremst for å få mer lagringsplass. Disse ble oppfunnet når spillene begynte å bli så avanserte at de krevde mer plass enn de 32KiB-ene som er avsatt til PRG-ROM i adresseområdet. For å få mer plass har man for eksempel fire ROM-chipper på 16KiB: Én med kode og tre andre med data (brett-designene for eksempel). ROM-chippen med koden blir typisk koblet til fra $C000 til $FFFF, og så kan det andre området fra $8000 til $BFFF kobles til en av de tre andre ROM-chippene, etter behov. Dette er det mapperen utfører. Når koden for eksempel trenger data som ligger på ROM-brikke nummer 3, sender den en instruks til mapperen (vanligvis i et adresseregister) om at ROM-brikke 3 skal switches inn. Deretter kan den lese fra denne ROM-brikken på vanlig måte gjennom adressene fra $8000 til $BFFF.

Igjen er det viktig å merke seg at for CPU-en er det ikke noen forskjell på adressene. Hvis den på en eller annen måte kommer over en `JMP $2002`-instruksjon vil den starte å hente instruksjoner fra $2002, som er statusregisteret til PPU-en. Det den får tilbake fra PPU-en vil den da tolke som en instruksjon (husk på at instruksjoner er tall). Dette kan tilfeldigvis være en gyldig instruksjon, eller det kan være en ugyldig instruksjon som får prosessoren til å gå i lås. Poenget er at vi må passe på at vi ikke lar CPU-en få gå fritt utover i adresseområdet. Når vi er ferdig å gi CPU-en ting den skal gjøre er det vanlig praksis å la den gå inn i en uendelig løkke. På den måten holder vi den i sjakk, for å si det slik.

[3] Vi *kan* også skrive til adressene mellom $0800 og $1FFF. De samme RAM-bytene dukker opp i adressene fra $0800 til $0FFF, $1000 til $17FF og $1800 til $1FFF. Å skrive til adressen $0803 vil ha akkurat samme effekt som å skrive til $0003 eller $1003.

Litt terminologi (page, offset, lave og høye deler av adresser)

I 6502-sammenheng deles adresseområdet ofte inn i *pages* på 256 (= $100) bytes. Pagene numereres fra 0 og oppover. Adressene fra $0000 til $00FF utgjør altså første page – page 0, $0100 til $01FF utgjør page 1, og så videre. Hvis vi ser på en generell adresse så kan vi lett identifisere hvilken page den befinner seg i ved å se på de to første heksadesimale sifrene; $312E er f.eks. en adresse som befinner seg i page $31 (= 49). Denne delen av adressen kalles også den *høye delen* av adressen, siden den består av de 8 høyeste bitene av adressen. De resterende to sifrene til høyre i adressen utgjør den *lave* delen av adressen. Denne forteller oss hvor mange bytes adressen peker inn i den aktuelle pagen. Adressen $312E er altså til en byte som befinner seg $2E (= 46) bytes inn i page $31. Vi kaller også den lave delen for adressens *offset* eller *indeks* inn i den aktuelle pagen.

Å tenke på adresser som bestående av et pagenummer og en offset (eventuelt høy og lav del) kan være veldig nyttig. Når vi skal begynne å håndtere større datamengder må vi ofte forholde oss til data som strekker seg over 256 bytes. Da må vi manipulere adresser som er 16 bits i biter på 8 bits om gangen (husk at registrene bare er 8 bit store!)

Zero page

En noe spesiell page for 6502-prosessoren er den første, den såkalte *zero page*. Det som er spesielt med zero page er at adressene til bytene i denne pagen bare er 8 bit store (den høye delen er 0). Dette utnytter CPU-en ved at den har en egen adresseringsmåte (som ikke ble tatt med ovenfor) – absolutt zero page-adressering. I stedet for at argumentet til en slik instruksjon tar opp to bytes (som det vanligvis gjør ved absolutt adressering), tar det bare opp én byte, og så er det implisitt gitt at argumentet er en offset inn i zero page. Fordelen med dette er at slike instruksjoner tar mindre plass og ikke minst mindre tid å utføre.

Når vi programmerer bør vi i så stor grad som mulig definere alle variabler i zero page. Da utføres koden vår så raskt som mulig. Merk at ca65 selv finner ut om variablene er definert i zero page eller ikke, og velger korrekt adresseringsmåte for instruksjonen når den assembler koden. Vi trenger bare å sørge for at variablene er definert i zero page.

3.3.1 Stacken

En *stack* («stabel») er en spesiell datastruktur som fungerer som navnet tilsier, som en stabel. Man har to operasjoner for å manipulere stacken: *push* (legg en ny verdi øverst på stacken) og *pop* (ta øverste verdi av stacken). Dette her analogt med f.eks. en stabel med tallerkener. Det eneste man kan gjøre er å legge til en ny tallerken, eller ta av den øverste. Det er ikke så lett å ta ut en tallerken midt i stabelen.

De aller fleste prosessorer har en eller annen form for stack implementert. Stacken muliggjør å ha såkalte *subrutiner*, som vi snart skal komme tilbake til, og kan være kjekk å ha når vi skal skrive mer kompleks kode der alle registrene er i bruk.

I 6502-prosessoren er stacken 256 bytes stor og dekker adressene fra $0100 til $01FF (page 1).

Bunnen av stacken ligger merkelig nok til slutt i denne pagen, i $01FF, og så vokser stacken *nedover*. S-registeret fungerer som nevnt som prosessorens indeksregister i stacken. S-registeret holder til en hver tid offseten / indeksen til den neste ledige byten i stacken. Når en verdi pushes på stacken, senkes S med 1, og når en verdi popes av stacken, øker S med 1. For å pushe og pope ting på stacken, og for å manipulere S-registeret har prosessoren følgende instruksjoner:

Instruksjon	**Navn**
PHA	PusH A
PLA	PulL (pop) A
PHP	PusH P
PLP	PulL P
TXS	Transfer X to S
TSX	Transfer S to X

Merk at det kun er A- og P-registrene som kan pushes og popes. For å pushe X- og Y-registrene på stacken må vi først overføre verdiene deres til A og så pushe A. Det finnes heller ingen «LDS»-instruksjon. Hvis vi vil gi S-registeret en verdi må vi først legge verdien i X-registeret og deretter benytte TXS-instruksjonen. Et kort eksempel på bruk av stacken:

```
LDX #$FF    ; Stacken starter i offset $FF
TXS         ; Overfør til S

; Nå er stacken satt opp, og vi kan pushe og poppe verdier på den:

LDA #4
PHA         ; Nå inneholder stacken verdien 4
LDA #2
PHA         ; Nå inneholder stacken verdiene 4 og 2
LDA $1234   ; Legg verdien i adresse $1234 i A
PLA         ; Nå inneholder A 2 igjen og stacken inneholder verdien 4
PLA         ; Nå inneholder A verdien 4
```

3.4 Subrutiner

I høynivåspråk som C, Java, Python osv. kan vi definere det som kalles *funksjoner* eller *subrutiner*. Disse består av kode som vi kan *kalle* fra andre deler av koden. 6502-prosessoren gjør det også mulig å definere subrutiner. I et NES-spill vil det f.eks. ofte være aktuelt å sjekke om en figur befinner seg innenfor skjermbildet eller ikke. Hvis vi legger dette i en subrutine, kan vi kalle subrutinen hver gang vi ønsker å finne ut dette, i stedet for å gjenta koden hver eneste gang. La oss se på hvordan subrutiner fungerer for 6502-prosessoren.

En subrutine er (selvfølgelig) ikke noe annet enn en rekke instruksjoner. En subrutine kalles med instruksjonen JSR («Jump to SubRoutine»). Denne instruksjonen fungerer som JMP, med en viktig forskjell: Adressen til den neste instruksjonen (denne adressen kalles *returadressen*) blir pushet på stacken før hoppet skjer. På den måten kan prosessoren huske hvor den skal

fortsette når subrutinen er ferdig. En subrutine signaliserer at den er ferdig med instruksjonen `RTS` («ReTurn from Subroutine»). Når prosessoren kommer over en slik instruksjon, poper den returadressen av stacken og hopper til denne adressen. Da fortsetter altså programmet der det var før subrutinen ble kalt. Merk at prosessoren ikke «vet» at den utfører kode i en subrutine. Dersom prosessoren havner i en subrutine uten at den ble kalt med `JSR` (vi kan f.eks. ha hoppet med `JMP`, eller prosessoren kan ha kommet dit fordi subrutinen ligger rett etter den ordinære koden), og den så kommer til instruksjonen `RTS`, vil den pope de to øverste bytene på stacken og hoppe til adressen de utgjør. Følgende eksempel viser hvordan vi definerer og kaller subrutiner:

```
kode:
    ; her er den 'ordinære' koden
    JSR subrutine
    ; her fortsetter koden etter kallet
    ...

subrutine:
    ; her er koden som subrutinen skal utføre

    RTS     ; her avslutter subrutinen
```

En annen viktig ting å passe på i en subrutine er at man har kontroll på stacken. Hvis subrutinen benytter stacken til å for eksempel mellomlagre registerverdier og disse ikke er popet av igjen før RTS-instruksjonen så vil de lagrede verdiene popes av og benyttes av prosessoren som returadresse.

En subrutine kan godt kalle andre subrutiner selv. Stacken tar opp opp til en hel page, så det er som regel nok av plass til å lagre returadresser (opp til 128 stykker).

3.5 Avbrudd

Prosessoren kjører som sagt et kontinuerlig løp der den henter og utfører instruksjoner. Men noen ganger skjer det ting utenfor prosessoren som krever dens oppmerksomhet. I NES vil f.eks. lydprosessoren trenge å få tilført nye data nå og da, og PPU-en må få oppdatert grafikkdataene før neste frame tegnes. I stedet for at koden vår hele tiden må sjekke om det er noe nytt fra hver enkelt komponent i systemet, har man i stedet såkalte *avbrudd* (interrupts). En komponent som trenger prosessorens oppmerksomhet sender den et avbruddssignal. Disse finnes i to varianter: IRQ («Interrupt Request») som prosessoren kan velge å ignorere, og NMI («Non-Maskable Interrupt») som prosessoren må ta hånd om med en gang. Et IRQ-avbrudd ignoreres når I-flagget («Interrupt Disable») i P-registeret er 1.

Det som skjer når prosessoren mottar et avbruddssignal er at den begynner å kjøre en spesiell subrutine som kalles en *interrupt-handler*. Når denne er fullført går prosessoren tilbake og fortsetter å kjøre koden der den stoppet når avbruddsignalet kom. Vi kan definere forskjellige interrupt-handlere for IRQ- og NMI-avbruddene. Dette er nyttig da IRQ-avbrudd vil komme fra APU-en, mens NMI-avbrudd kommer fra PPU-en. Vi skal snart se hvordan vi setter opp en interrupt-handler, men først må vi se litt mer detaljert på hva som skjer.

Når prosessoren får et avbruddssignal (og det ikke ignoreres i tilfelle det er et IRQ-signal), kan

den ikke uten videre avbryte det den holder på med. Det må være mulig for den å gå tilbake til det stedet der den ble avbrutt etter at interrupt-handleren har kjørt, og fortsette i den samme *tilstanden* som den hadde før avbruddet. Det vil si at flaggene i P-registeret må være de samme som før. Hvis ikke kan det være at koden kjører annerledes etter avbruddet fordi et flagg har endret verdi (avbruddet kan for eksempel skje mellom en `CMP`-instruksjon og en betinget hoppeinstruksjon). Det som i korte trekk skjer er at prosessoren pusher P-registeret på stacken. Deretter finner prosessoren frem adressen til den riktige interrupt-handleren og hopper til subrutinen. Adressen til interrupt-handleren finner den i en av de tre *adressevektorene* som er nevnt tidligere. Interrupt-handleren returnerer ved å benytte den spesielle instruksjonen `RTI` («ReTurn from Interrupt»). Da popes P-registeret og adressen til den neste instruksjonen som skulle utføres da avbruddet inntraff fra stacken, og CPU-en fortsetter som før.

Adressevektorer

Hver avbruddstype har som sagt sin egen adressevektor. Adressevektoren inneholder adressen til det stedet som prosessoren skal begynne å kjøre kode fra når avbruddet av den aktuelle typen skjer. Følgende tabell viser oversikten over adressevektorene:

Adresser	Avbrudd
$FFFA - $FFFB	NMI
$FFFC - $FFFD	Reset
$FFFE - $FFFF	IRQ

I linker-filen `linker_config.cfg` som vi benytter i denne guiden, er det definert et eget *segment* for adressevektorene. Dette heter `VECTORS`. De seks bytene som puttes i dette segmentet vil plasseres i adressene fra $FFFA til $FFFF i PRG-ROM-delen av .nes-filen.

Vi kan tenke oss at vi har skrevet en interrupt-handler for NMI som starter ved labelen kalt `NMI`, og en for IRQ som starter ved labelen `IRQ`. For å fylle inn de tre adressevektorene gjør vi da følgende:

```
.segment "VECTORS"
    .word NMI
    .word Start
    .word IRQ
```

`.word`-direktivet fyller som vi så i forrige kapittel inn en 16-bits verdi (et «word»). Den første linja, `.word NMI` fyller altså inn to bytes som består av adressen som labelen `NMI` svarer til, og så videre. ca65 tillater oss også å skrive dette slik:

```
.segment "VECTORS"
    .word NMI, Start, IRQ
```

Dersom vi ikke har noen NMI- eller IRQ-handler kan vi fint sette adressevektorene til 0. Merk at *hvis* det da skjer et IRQ- eller NMI-avbrudd, vil prosessoren hoppe til adresse 0 (dvs. starten

av RAM) og kjøre koden som er der! Den eneste adressevektoren vi er nødt til å ha med er den midterste, som tilhører Reset-avbruddet. Som sagt er det denne adressevektoren prosessoren henter når den slås på (og resettes) for å finne ut hvor den skal starte å kjøre kode fra.

Programavbrudd

I tillegg til eksterne komponenter tilkoblet CPU-en kan også programmet selv avbryte prosessoren. Dette kalles programvareavbrudd («software interrupts»). Et slikt avbrudd er et IRQ-avbrudd (så det er IRQ-handleren som vil kjøres), og settes i gang med instruksjonen `BRK` («BReaK»). CPU-en gjør det samme som ved et fysisk avbrudd, men med en viktig forskjell: Når P pushes på stacken *settes B-flagget til 1 i den kopien av registeret*. Det indikerer at det er et programvareavbrudd og ikke et fysisk avbrudd som har skjedd. Den eneste måten for IRQ-handleren å sjekke dette på er faktisk å pope av stacken (og på igjen) og sjekke om B er satt [4].

3.6 Klokkesykluser

Som sagt i kapittel 1 har CPU-en en klokkefrekvens på 1.79MHz for NTSC-modeller, og 1.66MHz for PAL-modeller. Klokkefrekvensen angir hvor ofte CPU-en mottar et *klokkesignal* (man sier da at den *klokkes*). Hver gang CPU-en klokkes av et klokkesignal, utfører den et nytt *steg* i prosessen. Typiske steg vil være å hente en opcode, hente et argument, utføre en logisk eller aritmetisk operasjon, og så videre. Stegene er altså mindre operasjoner enn instruksjonene. En instruksjon tar minst to klokkesykluser, og noen kan ta helt opp til 6. Det er adresseringsmåten som avgjør dette; desto mer komplisert den er, desto lengre tid bruker prosessoren på å utføre instruksjonen. Vi kan for eksempel se på instruksjonen `LDA #20`. Denne tar to klokkesykluser å utføre. I den første klokkesyklusen hentes opcoden fra adresseområdet. I den andre klokkesyklusen hentes tallet 20 og legges i A. Instruksjonen `LDA ($12,X)` tar i kontrast 6 klokkesykluser. Det er ikke urimelig, CPU-en må jo da utføre langt flere steg for å finne frem til verdien som skal legges i A: Først må opcoden hentes, som tar én syklus. Deretter hentes zero-page-adressen $12 og legges sammen med X, som tar to sykluser. Så settes denne på adressebussen og de to nye adressebytene hentes inn, som tar to sykluser til. Til slutt skal verdien hentes fra adresseområdet og legges i A, som tar én syklus.

Vi skal snart se at vi ikke har ubegrenset tid til å oppdatere grafikken før hvert skjermbilde tegnes. Da er det viktig at vi ikke sløser for mange klokkesykluser, og derfor er det viktig å være klar over hvor mange klokkesykluser instruksjonene tar (hvertfall sånn omtrentlig). En god tommelfingerregel er at instruksjonen bruker én klokkesyklus for hver gang den gjør noe i adresseområdet.

En komplett oversikt finnes på den nevnte nettsiden `http://www.obelisk.demon.co.uk/6502/reference.html`. Her står det hvor mange klokkesykluser hver instruksjon tar med de forskjellige adresseringsmåtene.

[4] B-flagget eksisterer egentlig ikke fysisk i P-registeret. Det er kun en bit som settes i kopien av P som pushes på stacken når en BRK utføres.

Kapittel 4

PPU-en

Nå har vi vært gjennom alt vi trenger å vite om CPU-en på en god stund, så nå er det på tide å vende blikket over på PPU-en. I motsetning til CPU-en, som utfører instruksjoner fra et program, er PPU-en *hardkodet* til å gjøre det samme hele tiden. Dette ligger i selve kretsdesignet. I den grad PPU-ens atferd kan forandres, er det snakk om noen enkle instillinger som CPU-en kan gi den. Alle forandringer av skjermbildet er det CPU-en som må utføre ved å legge data i minnet til PPU-en.

4.1 Grunnleggende om PPU-en

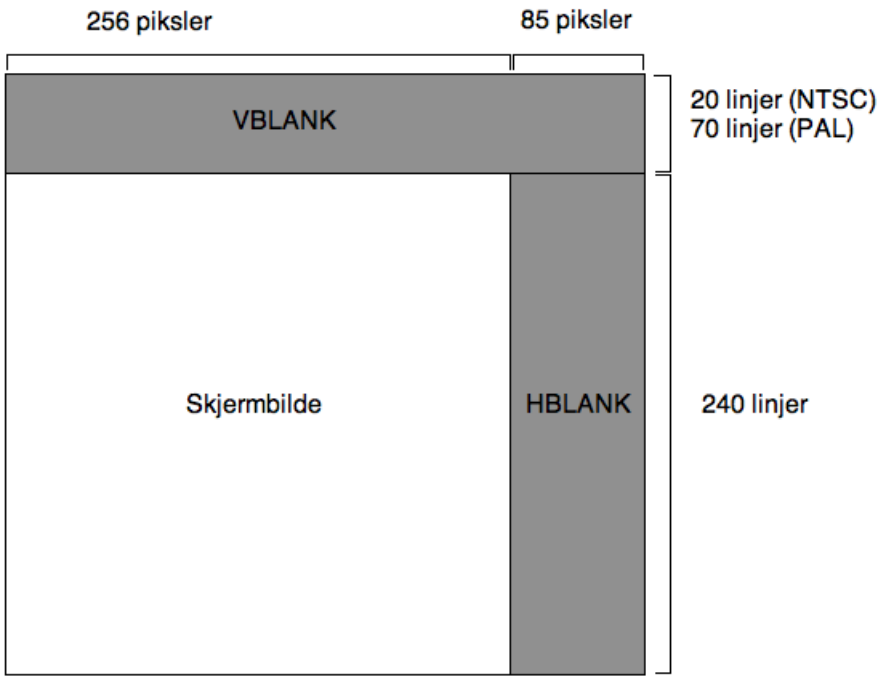
Før vi ser på detaljene rundt hvordan bakgrunner og sprites er bygd opp, må vi se litt nærmere på de grunnleggende delene av PPU-en og hvordan den fungerer.

4.1.1 Renderingssyklusen

Her blir det litt repetisjon fra kapittel 1, men det skader ikke.

Som CPU-en er PPU-en en *klokket krets*. Hver gang PPU-en mottar et nytt klokkesignal, skjer det et nytt steg internt, og det kommer et TV-signal til enten en piksel eller eventuelt et vblank-signal ut). Klokkefrekvensen til PPU-en er henholdsvis 3 ganger høyere (NTSC) eller 3.2 ganger høyere (PAL) enn CPU-ens klokkefrekvens.

Når PPU-en tegner skjermbildet bruker den data- og adressebussen sin konstant for å hente dataene den trenger. Da har vi rett og slett ikke mulighet for å endre på noe i adresseområdet dens. Når PPU-en derimot er i vblank-modus, skjer det så og si ingen aktivitet innvendig. Det er kun da CPU-en har tilgang til PPU-en og dens adresseområde. Som vi ser fra figur 4.1 varer denne perioden i $(256+85)\cdot 20 = 6820$ PPU-klokker for NTSC, og $(256+85)\cdot 70 = 23870$ for PAL. Dette svarer til henholdsvis 2273 CPU-klokker for NTSC og 7956 CPU-klokker for PAL. Hvis vi tenker oss at en gjennomsnittelig instruksjon tar 4 klokkesykluser, vil det si at vi i snitt klarer å presse inn omtrent 568 instruksjoner (NTSC) og 1989 instruksjoner (PAL) i løpet av én vblank-periode. Dette er ikke ubegrenset mange, så det er viktig at vblank-koden er effektiv.



Figur 4.1: *Signalene som sendes av PPU-en og varigheten av disse. 85 piksler for HBLANK vil si at grafikkprosessoren genererer et HBLANK-signal i løpet av samme tid som den ville rukket å tegne 85 piksler.*

Hvordan kan vi vite når PPU-en er i vblank? CPU-en kan selvfølgelig vente til det har gått så og så mange klokkesykluser, men det er et håpløst prosjekt. For det første blir det vanskelig å holde en slik synkronisering gående over lengre tid, og det blir vanskelig å få gjort andre ting som må gjøres i et spill. Heldigvis kan PPU-en si i fra til oss om dette, faktisk på flere måter. Enten ved at vi leser av *statusregisteret* dens, eller ved at PPU-en sender et avbruddssignal til CPU-en.

4.1.2 Tiles og pattern tables

Bare for å minne om det igjen: Alt som befinner seg på skjermen er enten del av *bakgrunnen* eller bestående av bevegelige objekter kalt *sprites*. Både bakgrunnen og spritene er bygd opp av en felles «grunnenhet» kalt *tiles*.

Tilene er blokker på 8x8 piksler, der hver piksel representeres av et 2-bits tall. Det vil si at en tile tar opp $8 \cdot 8 \cdot 2 = 128$ bits, eller 16 bytes. Vi kan tenke på tilene som «byggeklosser» som settes sammen for å danne større bilder. Hvordan tilene settes sammen til bakgrunner eller brukes i sprites, bestemmes av hva som ligger i VRAM og OAM («Object Attribute Memory»). Det er det koden vår som avgjør. Selve grafikken til tilene ligger i en egen ROM-chip, CHR-ROM, på spillkassetten. CHR-ROM er 8KiB stort [1], og har altså plass til 8KiB / 16 bytes = 512 tiles.

I CHR-ROM er tilene ordnet i to såkalte *pattern tables* på 4KiB hver. De 256 første tilene ligger

[1] Det finnes mange spill med større CHR-ROM, men disse må bruke en såkalt mapper for å dra nytte av den ekstra plassen.

i pattern table 1, mens de 256 neste ligger i pattern table 2. Det er ikke noe hokus pokus over dette – pattern table er bare det offisielle navnet på disse områdene. Grunnen til at vi har denne oppdelingen er at PPU-en ikke kan bruke alle de 512 tilene samtidig. Den kan bare velge blant de 256 første (pattern table 1) eller de 256 siste (pattern table 2) tilene til bakgrunner og sprites. I adresseområdet til PPU-en ligger pattern tablene helt først. Fra adresse $0000 til $0FFF har vi pattern table 1, og fra $1000 til $1FFF har vi pattern table 2.

4.1.3 Registrene

For at CPU-en skal kunne kommunisere med PPU-en, dukker den som vi så opp i adresseområdet til CPU-en i adressene fra $2000 til $2007. Hver av disse 8 adressene går til et register i PPU-en:

Adresse	**Register**	**Tilgang**
$2000	Kontroll 1	Skrive
$2001	Kontroll 2	Skrive
$2002	Status	Lese
$2003	OAM-indeks	Skrive
$2004	OAM-data	Lese/skrive
$2005	Scroll	Skrive
$2006	VRAM-adresse	Skrive
$2007	VRAM-data	Lese/skrive

Etter hvert som vi tar for oss forskjellige aspekter ved PPU-en, skal vi ta for oss de relevante registrene.

Tilgang ved oppstart

Når PPU-en starter opp tar det omtrent 30000 klokkesykluser før den er klar til å operere normalt. Før den tid er det flere av registrene som ignorerer dataene som CPU-en sender til den. Det første programmet vårt bør gjøre er altså å vente en stund før det begynner å skrive til PPU-en. Den enkleste måten er å vente til det har gått to frames (to vblank-perioder). Senere i kapittelet skal vi se hvordan det gjøres.

4.1.4 Adresseområdet

PPU-en har sin egen data- og adressebuss, separat fra CPU-en. Dette er nødvendig for å unngå at begge prøver å lese og skrive til adresseområdet samtidig. PPU-ens adressebuss er 14 bits bred, det vil si at adressene er fra $0000 til $3FFF. I adresseområdet finner vi følgende:

Adresser	**Beskrivelse**	**Fysisk lagringsplass**
$0000 - $0FFF	Pattern table 1	CHR-ROM
$1000 - $1FFF	Pattern table 2	CHR-ROM
$2000 - $23BF	Nametable 1	VRAM
$23C0 - $23FF	Attribute table 1	VRAM
$2400 - $27BF	Nametable 2	VRAM
$27C0 - $27FF	Attribute table 2	VRAM
$2800 - $2BBF	Nametable 3	VRAM
$2BC0 - $2BFF	Attribute table 3	VRAM
$2C00 - $2FBF	Nametable 4	VRAM
$2FC0 - $2FFF	Attribute table 4	VRAM
$3F00 - $3F0F	Bakgrunnspalett	Palett-RAM
$3F10 - $3F1F	Sprite-palett	Palett-RAM

Vi skal snart se på nametables, attribute tables og palettene.

Lesing og skriving fra/til adresseområdet

Alt i adresseområdet, utenom de to pattern tablene, må vi fylle med data som forteller hvordan skjermbildet skal se ut. Det er altså koden vår som må gjøre dette. PPU-en har to registre, $2006 (VRAM-adresse) og $2007 (VRAM-data). Disse to registrene gir CPU-en tilgang til adresseområdet til PPU-en. $2006 bruker vi til å fylle opp et internt 14-bits *adresseregister* i PPU-en. Her skriver vi den adressen i PPU-ens adresseområde som vi ønsker å skrive til eller lese fra. Første gang vi skriver, fylles de 6 høyeste bitene av registeret opp. Neste gang vi skriver, fylles de 8 lave bitene opp. Da er adressen satt, og vi kan skrive til eller lese fra $2007. Da vil PPU-en sette den gitte adressen på adressebussen, og enten skrive verdien vi la i $2007, eller lese inn en verdi og legge den i $2007 (og da videre til et register i CPU-en). Etter at vi har skrevet en verdi til $2007 eller lest en verdi fra $2007 vil adressen i adresseregisteret automatisk øke med enten 1 eller 32. Det velger vi ved å sette flagget $2000.2:

Kontroll 1 ($2000)		
Bit	**Beskrivelse**	**Valg**
2	Adresseøkning etter skriving til / lesing fra $2007	0: 1, 1: 32

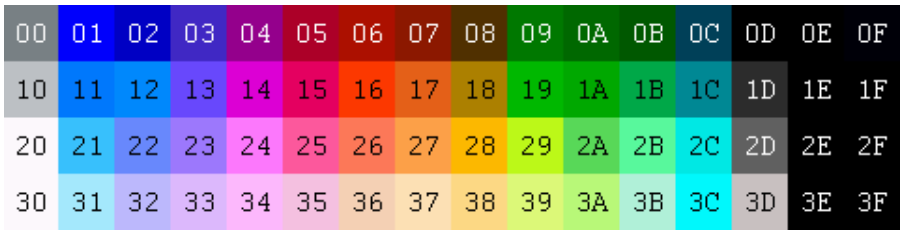
Bruken av $2006 og $2007 er best vist med et kort eksempel. La oss tenke oss at vi vil skrive 42 til den første byten i nametable 1. Fra oversikten ovenfor ser vi at adressen dit er $2000.

Den høye delen av denne adressen er $20, og den lave delen er $00. Da skal vi først skrive $20 til $2006 og deretter $00 til $2006. Så skal vi skrive 42 til $2007:

```
LDA #$20
STA $2006
LDA #$00
STA $2006
LDA #42
STA $2007
```

4.2 Farger

Dagens datamaskiner lagrer hver individuelle piksel som for eksempel en 24-bits verdi – 8-bits for hver fargekomponent (rød, grønn, blå). PPU-en gjør det litt vanskeligere. Den har et fast utvalg av farger; den såkalte *PPU-paletten*:



Figur 4.2: *Dette er alle fargene som er mulig å vise.*

Dette er fargene vi kan velge blant. PPU-en kan ikke generere noen flere farger. Som vi husker fra forrige seksjon så har tilene bare to bits per piksel, som bare er nok til å velge mellom fire farger. For å løse dette har PPU-en et internt minne hvor vi kan lagre til sammen åtte paletter (fargeutvalg): fire bakgrunnspaletter og fire sprite-paletter. Hver slik palett består av tre farger som vi velger ut i fra PPU-paletten. Når en tile plasseres i bakgrunnen, eller brukes som grafikken til en sprite, velger vi en av disse palettene og sier at den skal gjelde for tilen. Da er det fargene innenfor denne paletten tilen må velge blant. Hvis en piksel for eksempel har verdien 1, velges den første fargen i den valgte paletten, og så videre. En piksel som har verdien 0 vil være transparent.

Dette systemet er noe innviklet, og det setter klare begrensninger på hvilke farger tilene kan ha, siden de bare kan bare ha farger innenfor én palett. Samtidig gjør det at vi lett kan bruke samme tile til forskjellige ting, bare ved å endre fargene i paletten. Det er noe som gjøres ofte i de fleste NES-spill. Se for eksempel på Super Mario Bros:

I bildet til venstre ser vi at både skyene og buskene deler samme tiles (se på den øvre delen av skyene), bare med forskjellige palettvalg. I bildet til høyre ser vi at gulvet i undergrunnsverdenene er akkurat det samme som på overflaten, bare med en annen palett.

4.2.1 Palett-RAM

Palettene lagrer vi i et internt minne i PPU-en. Dette minnet er tilkoblet adressebussen, på adressene fra $3F00 til $3F1F:

Adresser	**Beskrivelse**
$3F00	Bakgrunnsfarge
$3F01 - $3F03	Bakgrunnspalett 1
$3F05 - $3F07	Bakgrunnspalett 2
$3F09 - $3F0B	Bakgrunnspalett 3
$3F0D - $3F0F	Bakgrunnspalett 4
$3F10	Speiling av bakgrunnsfargen
$3F11 - $3F13	Sprite-palett 1
$3F15 - $3F17	Sprite-palett 2
$3F19 - $3F1B	Sprite-palett 3
$3F1D - $3F1F	Sprite-palett 4

For å skrive til palett-RAM benytter vi den metoden som ble skissert i seksjonen om adresseområdet: Vi skriver først adressen til $2006 og deretter verdien til $2007. Vi kan for eksempel være interessert i å velge den globale bakgrunnsfargen. Den ligger i adresse $3F00. La oss si at vi ønsker en blå farge. Da må vi se på PPU-paletten. $12 ser ut som en fin blåfarge, så vi prøver den:

```
LDA #$3F      ; Høy del av adressen
STA $2006
LDA #$00      ; Lav del av adressen
STA $2006
LDA #$12      ; Fargekoden til blåfargen
STA $2007
```

Som regel vil vi få bruk for alle de åtte palettene. I stedet for å skrive en haug med LDA- og STA-instruksjoner for å lagre fargene i palett-RAM-en, kan vi i stedet gjøre det smartere på denne måten:

```
.segment "DATA"
palett:
    ; Bakgrunnspaletter:
    .byte $00, $10, $11, $12    ; 1
    .byte $00, $10, $11, $12    ; 2
    .byte $00, $10, $11, $12    ; 3
    .byte $00, $10, $11, $12    ; 4

    ; Sprite-paletter:
    .byte $12, $10, $11, $12    ; 1
    .byte $00, $10, $11, $12    ; 2
    .byte $00, $10, $11, $12    ; 3
    .byte $00, $10, $11, $12    ; 4
```

```
.segment "CODE"
    ; ...
    LDA $3F          ; Sett start-adressen
    STA $2006
    LDA $00
    STA $2006
    LDX #0
last_palett:
    LDA palett, x    ; Hent byte nr. x i paletten
    STA $2007        ; Send til PPU-en
    CPX #32          ; Har vi tatt alle 32 bytene?
    BNE last_palett  ; Nei, hopp opp igjen
    ; ...
```

Når vi gjør det slik som her, vil PPU-en riktignok skrive til adressene $3F04, $3F08, ..., som ikke er nevnt ovenfor. Det gjør ikke noe [2]. Det eneste vi må være obs på er at $3F10 er en speiling av $3F00. Det betyr at når loopen kommer til byte nr. $10 og vi skriver til $3F10, vil bakgrunnsfargen overskrives. Derfor må bakgrunnsfargen alltid oppgis i denne byten (første i sprite-delen), i stedet for i den første byten.

4.3 Bakgrunner

Bakgrunner er kanskje det som er mest vrient å forstå fullt ut når det kommer til hvordan PPU-en fungerer. Det som er spesielt vanskelig er hvordan PPU-en fargelegger bakgrunnen, og hvordan den scroller mellom flere bakgrunner. De tingene tar vi etter hvert. Vi begynner med noe enklere, nemlig hvordan en bakgrunnen bygges opp av tiles.

4.3.1 Nametables

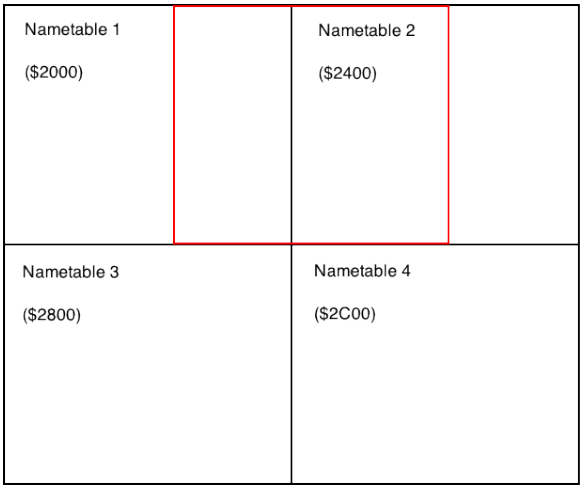
Som vi så i kapittel 1, kunne skjermbildet deles opp i et rutenett på 32 x 30 tiles. Det som forteller PPU-en hvilken tile som skal plasseres hvor i rutenettet er en datastruktur i VRAM som kalles et *nametable*. Hver byte i nametablet korresponderer med en tile i rutenettet. De 32 første bytene forteller hvilke tiles som skal plasseres i den øverste raden (fra venstre mot høyre), de 32 neste forteller hvilke tiles som skal plasseres i den neste raden, og så videre. Verdien til hver byte er en indeks inn i pattern table, som altså gir nummeret til tilen. Hvis en byte i nametablet er 5, vil den 6. tilen (vi teller fra 0) i pattern table vises. Nå er det kanskje forståelig hvorfor det er to pattern tables – én byte er rett og slett ikke nok til å velge blant alle de 512 tilene i CHR-ROM. Vi forteller PPU-en hvilken pattern table vi vil at den skal hente tiles til bakgrunnen fra ved å skrive til $2000.4:

Kontroll 1 ($2000)		
Bit	**Beskrivelse**	**Valg**
4	Pattern table for bakgrunnen	0: $0000, 1: $1000

[2] Adressene i regionen $3F00 - $3F0F går til gyldige lagringsplasser i palett-RAM som aldri brukes, mens adressene i regionen $3F10 - $3F1F er speilinger av disse. Å skrive til dem gjør ikke noe.

4.3.2 Scrolling

I utgangspunktet dekker et nametable akkurat et helt skjermbilde. Grunnen til at vi har flere er for å gjøre det mulig å *scrolle* («rulle»). PPU-en setter sammen nametable-dataene slik som det er vist i Figur 4.3. Den røde firkanten representerer det synlige skjermbildet som PPU-en renderer. Vi kan flytte på dette «vinduet» ved å skrive til $2005 to ganger: Første gang vi skriver bestemmer vi hvor mye vinduet skal flyttes mot høyre, og andre gang vi skriver bestemmer vi hvor mye det skal flyttes nedover.



Figur 4.3: *Nametablene kan tenkes på som «virtuelle skjermer» som PPU-en kan scrolle mellom. Det røde vinduet representerer det synlige skjermbildet.*

Forflytningen som vi bestemmer ved å skrive til $2005 er *relativ til et valgt nametable*. Det nametablet kaller vi *base-nametablet*. Vi velger base-nametable ved å skrive til de to nedre bitene i $2000:

Kontroll 1 ($2000)		
Bits	**Beskrivelse**	**Valg**
0-1	Base- nametable	0: $2000, 1: $2400, 2: $2800, 3: $2C00

Å skrive her bestemmer hvilket nametable som vises på skjermen når det ikke er satt noen scroll. Hvis eksempelvis nametable 2 velges som base-nametable og vi så scroller mot høyre, vil venstre del av nametable 1 dukke opp. Dette er det litt viktig å få med seg. Alle sidescroller-spill (platformspill) utnytter dette til å vise områder som er mye større enn det to nametables alene kan dekke.

4.3.3 Nametable-konfigurasjoner og speiling

PPU-en har satt av adresser til fire nametables, men i en NES-konsoll er det bare 2KiB VRAM, som er nok til å lagre to nametables. Årsaken til dette er nok at Nintendo ville spare på RAM (som var kostbart på den tiden). De så nok heller ikke for seg at fire nametables ville være

nødvendig med det første (ingen av de første spillene til NES drar nytte av dette). Dersom et spill trenger flere nametables, kan det ha en egen RAM-chip på spillkassetten, og koble denne inn på de ledige nametable-adressene.

Spill som ikke har egen RAM til flere nametables kan velge å konfigurere de to interne nametablene som enten nametable 1 og 2 eller 1 og 3 (se Figur 4.3). Førstnevnte konfigurasjon brukes i spill der man scroller mot venstre, mens sistnevnte brukes i spill der man scroller nedover. Hvis førstnevnte konfigurasjon velges, vil nametable 3 og 4 *speile* nametable 1 og 2. Det vil si at disse adressene går til henholdsvis nametable 1 og nametable 2. På samme måte; i vertikal konfigurasjon vil nametable 2 speile nametable 1 og nametable 4 vil speile nametable 3.

Disse konfigurasjonene er noe vi må oppgi i INES-headeren for at emulatoren skal vite hvordan den skal sette opp nametablene.

4.3.4 Attribute tables

Nametablene sier altså hvilke tiler som skal plasseres hvor i bakgrunnen. Det som ikke ligger her er informasjon om hvilken av de fire bakgrunnspalettene tilene skal hente farger fra. Det oppgir vi i et såkalt *attribute table*. Hvert nametable har et attribute table som ligger rett etter nametablet i VRAM (se oversikten ovenfor).

Hver byte i attribute table dekker et område av nametablet på 4 x 4 tiles. Den første byten dekker området øverst til venstre i nametablet. Videre er hver byte delt inn i fire grupper på to bits. Hver slik gruppe dekker en blokk *inni* området på 2 x 2 tiles, og forteller hvilken av de fire palettene disse tilene skal bruke. Dette er ganske innviklet, så her vil et bilde si mer enn tusen ord:

La oss ta utgangspunkt i Figur 4.4. Den første byten i attribute table dekker det første området på 4 x 4 tiles (den første røde firkanten øverst fra venstre), den andre byten dekker det neste til høyre for denne, og så videre. Bytene i attribute table er inndelt i grupper på to bits slik: %DDCCBBAA, hvor disse gruppene korresponderer til blokkene A, B, C og D innenfor området som byten dekker. Altså: De første to høyeste bitene dekker blokken øverst til venstre, de to neste dekker blokken øverst til høyre, de to neste dekker blokken under til venstre, og de to laveste bitene dekker blokken under til høyre. Dette er et veldig innviklet system, men det var en smart måte å spare plass på. Å manipulere attribute tablene krever mye jobb, men det skal vi takle når den tid kommer!

Figur 4.4: *Attribute table. Hver røde inndeling er én byte, som videre er inndelt i gruppene A, B, C og D.*

4.4 Sprites

Sprites er en helt annen sak enn bakgrunner, og på mange måter er de enklere å forholde seg til enn bakgrunnene. PPU-en opererer med to typer sprites: 8x8 piksler eller 8x16 piksler. Dette er noe vi velger når vi setter opp PPU-en, og det gjelder hele tiden (vi kan ikke ha noen på 8x8 og noen på 8x16 piksler). Det vanligste er 8x8-sprites, men i noen tilfeller kan 8x16 være nyttig. Vi velger mellom 8x8 og 8x16 med flagget $2000.5:

Kontroll 1 ($2000)		
Bit	**Beskrivelse**	**Valg**
5	Sprite-størrelse	0: 8x8, 1: 8x16

For å plassere en sprite på skjermen trenger vi ikke å styre med name- og attribute tables. All informasjon om spritene ligger i en spesiell datastruktur inni PPU-en som kalles OAM («Object Attribute Memory»). Dette er en lang liste med plass til informasjon om opp til 64 sprites. Denne informasjonen består av fire bytes i følgende format:

Byte	Beskrivelse
1	Y-koordinat (0 - 239)
2	Tile-nummer (0 - 255)
3	Instillinger (flagg)
4	X-koordinat (0 - 255)

X- og Y-koordinatene er ganske selvforklarende. Disse er koordinatene til *øvre venstre* hjørne av spriten. Tile-nummeret tolkes forskjellig av PPU-en avhengig av om den er i 8x8- eller 8x16-modus.

I 8x8-modus vil byten tolkes likt som bytene i nametable – den vil være en indeks inn i et av de to pattern tablene. Vi kan velge hvilken av dem som skal brukes for sprites ved å skrive til flagget $2000.3:

Kontroll 1 ($2000)		
Bit	**Beskrivelse**	**Valg**
3	Pattern table for sprites	0: $0000, 1: $1000

I 8x16-modus ignorerer PPU-en instillingen i $2000.3, og bruker i stedet bit 0 av tile-byten til å velge pattern table (0 er pattern table 1 og 1 er pattern table 2), mens de resterende bitene 1 - 7 utgjør en (partallig, siden bit 0 ikke er med) indeks inn i dette pattern tablet. Den resulterende tilen vil da brukes til den øvre delen av spriten, mens den neste (odde) tilen i pattern tablet vil brukes som den nederste.

Instillingsbyten består av følgende flagg:

Bit	Beskrivelse
0-1	Palettvalg
2-4	Ubrukt
5	Prioritet (0: foran bakgrunn, 1: bak bakgrunn)
6	Flipp horisontalt (0: nei, 1: ja)
7	Flipp vertikalt (0: nei, 1: ja)

Bitene 0 og 1 bruker vi til å oppgi hvilken av de fire sprite-palettene tilen skal få farger fra. Her står vi altså friere enn for bakgrunner, der klynger på 2x2 tiles må dele samme palett.

4.4.1 OAM

Spritene ligger i likhet med palett-RAM, i et internt minne i PPU-en. Men i motsetning til palett-RAM, er dette minnet *ikke* tilkoblet det ordinære adresseområdet, men har heller sin egen data-

og adressebuss [3]. CPU-en har tilgang til OAM via registrene $2003 og $2004, som fungerer på samme måte som $2006 og $2007 som vi bruker til å skrive til det ordinære adresseområdet. OAM er akkurat 256 bytes stort ($64 \cdot 4$ bytes). Det vil si at én byte akkurat er nok til å adressere hver byte Vi trenger altså bare skrive én gang til $2003 for å velge hvilken adresse i OAM vi vil skrive til. Når vi har skrevet en verdi til $2004 vil adressen i $2003 øke automatisk med 1. (**Merk**: Å lese fra $2004 fungerer ikke i alle tilfeller og bør unngås. Noen PPU-er støtter ikke dette i det hele tatt, mens andre ikke gir riktig verdi hver gang.)

La oss se på et eksempel der vi plasserer en sprite på skjermen. Det vi må gjøre da er å fylle inn fire bytes i OAM. Vi kan like greit velge de fire første:

```
LDA #0            ; Vi starter med den første byten i OAM
STA $2003
LDA #116          ; Y-verdi ca. midt på skjermen
STA $2004
LDA #0            ; Ta den første tilen i pattern tablet
STA $2004
LDA #%11000011    ; Flipp horisontalt og vertikalt, og velg den fjerde sprite-paletten
STA $2004
LDA #124          ; X-verdi ca. midt på skjermen
STA $2004
```

4.4.2 Sprite DMA

Vanligvis må OAM oppdateres hver frame, siden noen sprites må flyttes på, noen skal kanskje fjernes, og noen skal legges til. Husk på at den eneste tiden vi har tilgang til OAM er når PPU-en er i vblank-modus. Da har vi ikke ubegrenset med klokkesykluser, vi må gjøre oppdateringene raskt og effektivt. Bare for å lagre én eneste sprite ser vi at det trengs en serie med totalt åtte `LDA`- og `STA`-instruksjoner. Med bare 10 sprites har vi da svidd av 80 instruksjoner (og 480 klokkesykluser) allerede. I værste tilfelle bruker vi alle de 64 spritene. Da vil det ta 512 slike instruksjoner (3072 klokkesykluser) å oppdatere hele OAM. Det er mer enn vi har til rådighet i NTSC-maskiner (og omtrent halvparten av vblank-tiden i PAL-maskiner.)

NES-ingeniørene løste dette ved å inkludere en *DMA-kontroller* («Direct Memory Access») i CPU-kjernen. Det denne gjør er å kopiere en hel page fra RAM til OAM (via $2004). Dette skjer i løpet av bare 512/513 klokkesykluser, som gir oss mye vblank-tid til overs! Vi setter DMA-kontrolleren i sving ved å skrive til adresseregisteret $4014. Verdien vi skriver her er page-nummeret som vi ønsker å kopiere. Skriver vi for eksempel 5 her, vil pagen fra $0500 til $05FF kopieres til OAM. Før vi setter i gang kopieringen må vi skrive 0 til $2003, slik at kopieringen starter ved starten av OAM.

```
LDA #0            ; Vi starter med den første byten i OAM
STA $2003
LDA #$03          ; Kopier page 3 inn i OAM
STA $4014
```

[3] Dette er fordi PPU-en er helt avhengig av å hente bakgrunnsdata fra name, attribute og pattern tablene hele tiden, samtidig som den må gå gjennom OAM-listen på hver scanline. For å kunne gjøre disse tingene samtidig er det nødvendig at OAM har et eget separat adresseområde.

Det er vanlig og nødvendig praksis å bruke DMA i stedet for å skrive til OAM manuelt. Det tar riktignok opp en hel page i RAM som *kunne* vært brukt til annet, men i større spill vil man fort komme opp i mange sprites på skjermen samtidig. PPU-en vil heller ikke alltid motta dataene hvis vi skriver dem direkte via $2003 og $2004.

4.4.3 Rendering av sprites

I OAM er det plass til 64 sprites, men bare 8 av disse kan vises horisontalt ved siden av hverandre [4]. Dersom det er flere enn 8 ved siden av hverandre vil bare de 8 første (med lavest indeks i OAM) vises. PPU-en vil si i fra om dette har skjedd ved å sette et flagg i statusregisteret.

Når flere sprites er plassert slik at de overlapper hverandre vil PPU-en tegne den spriten med lavest indeks øverst.

4.5 Statusregisteret og kontrollregistrene

Kontroll 1 ($2000)

Dette registeret bruker vi til å kontrollere hvordan PPU-en skal rendere bildet. Hver bit i registeret er et flagg som aktiverer eller deaktiverer en eller annen funksjon. Vi har vært innom de fleste flaggene ovenfor, men her er en repetisjon:

Kontroll 1 ($2000)		
Bit	**Funksjon**	**Valg**
0-1	Aktivt nametable	0: $2000, 1: $2400, 2: $2800, 3: $2C00
2	Adresse-økning etter skriving til $2007	0: 1 byte, 1: 32 bytes
3	Pattern table for sprites:	0: $0000, 1: $1000
4	Pattern table for bakgrunnen	0: $0000, 1: $1000
5	Sprite-størrelse	0: 8x8, 1: 8x16
6	Ubrukt	-
7	Send NMI-avbrudd for hver nye frame	0: av, 1: på

Bit 7 er ny. Hvis vi setter denne til 1 vil PPU-en sende et NMI-avbrudd til CPU-en hver gang den kommer inn i vblank. Da vet vi at hver gang NMI-handleren blir kjørt, er PPU-en i vblank-modus, og vi kan oppdatere grafikken. Det eneste NMI-handleren bør gjøre er å oppdatere alt som har med grafikken og PPU-en å gjøre. Så snart det er gjort, kan den returnere, og den ordinære koden kan fortsette som normalt.

[4] Grunnen til dette ligger i kretsdesignet til PPU-en. Siden en ny piksel skal sendes ut for hver klokkesyklus, samtidig som én klokkesyklus bare er nok til å gjøre ett steg (f.eks. hente en byte fra OAM), er det begrenset hvor mange sprites PPU-en kan hente inn i løpet av en scanline.

Kontroll 2 ($2001)

Dette registeret holder resten av instillingene vi kan gi PPU-en:

Kontroll 2 ($2001)		
Bit	**Funksjon**	**Valg**
0	Farger eller gråtoner	0: farge, 1: gråtoner
1	Bakgrunnsklipping	0: på, 1: av
2	Sprite-klipping	0: på, 1: av
3	Bakgrunn	0: ikke vis, 1: vis
4	Sprites	0: ikke vis, 1: vis
5-7	Vektlegging av fargekomponenter («color emphasis»)	-

Bit 0 er ikke så veldig aktuell å diskutere i dag. Settes denne til 1 vil paletten bestå av forskjellige gråtoner i stedet for farger.

Bit 1 og 2 velger hva som skal skje i området lengst til venstre og høyre på skjermen. Hvis bit 1 er 0 vil de 8 første pikslene (fra venstre) og de 8 siste pikslene av bakgrunnen skjules, og det samme gjeler sprites når bit 2 er 0. Hvorfor i alle dager dette er et valg kommer vi tilbake til når vi ser på scrolling.

Dersom både bit 3 og 4 er 0, det vil si at verken bakgrunn eller sprites skal vises, vil PPU-en stoppe den vanlige renderingssyklusen. Det vil da ikke genereres noe TV-signal, og adresseområdet vil være tilgjengelig for CPU-en hele tiden. Dette er vanlig å bruke i oppstartsdelen av spillet, når det er mye som skal legges inn i VRAM (ny brettgrafikk, nye paletter, og så videre).

Bitene fra 5 til 7 brukes til å vektlegge henholdsvis den røde, grønne eller blå fargekomponenten i TV-signalet. Hvis for eksempel bit 5 er 1, vil hele paletten få et slags rødskjær, mens komplementærfargen til rød; blå, vil gjøres svakere. Det er ikke så ofte disse bitene brukes. Det beste er egentlig prøve seg frem for å se hva virkningen av de forskjellige instillingene er.

Status ($2002)

Dette registeret kan vi kun lese fra. Også dette er et register bestående flagg-biter, men her er det PPU-en som setter flaggene for å gi oss informasjon.

Status ($2002)	
Bit	**Funksjon**
0-4	Ubrukt
5	Sprite overflow
6	Sprite 0-kollisjon
7	Vblank

Bit 5 blir satt når PPU-en har registrert flere enn åtte sprites på én scanline.

Bit 6 forteller oss om sprite 0 (den første spriten i OAM) har kollidert med bakgrunnen. At den har kollidert med bakgrunnen vil si at både spriten og bakgrunnen har tegnet en piksel (forskjellig fra 0) på samme sted. Dette kalles «kollisjon», men det har ingenting med kollisjonstesting (i spillsammenheng) å gjøre. Grunnen til at PPU-en har denne noe merkelige oppførselen er at vi skal kunne gjøre såkalt *split-scrolling* der én del av skjermen står i ro mens en annen del av skjermen scroller. Statusbaren i Super Mario Bros. er et eksempel på det. Split-scrollingen skjer kort sagt ved at sprite 0 plasseres ved det stedet på skjermen hvor vi ønsker å splitte. CPU-en må så kjøre en loop som hele tiden sjekker statusregisteret og ser om bit 6 har blitt satt. Når den er det, vet vi at PPU-en har kommet til den scanlinen som vi ønsker å splitte ved. Ved neste hblank-periode får vi da akkurat tid til å skrive til de nødvendige registrene for å scrolle skjermbildet. Disse endringene vil da tre i kraft fra og med den neste scanlinen, og bildet som tegnes vil da være scrollet. I Super Mario Bros. er sprite 0 gjemt bak mynten som står ved siden av tallene som viser hvor mange mynter man har samlet.

Bit 7 forteller oss når PPU-en er i vblank-modus. Dersom vi har satt PPU-en til å trigge en NMI hver gang, så er ikke den informasjonen så interessant. Men under oppstarten av programmet vårt (før vi får satt opp instillingene) så er det den eneste måten å vite om PPU-en er i vblank eller ikke. Den letteste måten å vente på vblank på er slik:

```
vent:
    LDA $2002       ; Hent status fra statusregisteret
    BPL vent        ; Hvis bit 7 er satt, hopp opp igjen

    ; her er kode som kjører når PPU-en er i vblank
```

Hvis vi husker fra forrige kapittel så settes N-flagget i P-registeret hver gang en instruksjon resulterer i et negativt tall. Men negative tall vil jo egentlig bare si alle tall som har 1 i den syvende biten. Det vil si at når vi utfører `LDA $2002` så vil N-biten settes dersom vblank-flagget (bit 7) er satt, og være 0 så lenge det ikke er satt. Da kan vi enkelt utføre et betinget hopp, basert på om vblank-flagget er satt eller ikke. Instruksjonen `BPL` hopper som vi husker så lenge N-flagget er 0. Det vil altså si at så lenge vblank-flagget *ikke* er satt, vil koden over kjøre løkken om igjen og om igjen, helt til vblank-flagget settes. Da vil koden fortsette under `BPL`-instruksjonen.

Statusregisteret har en annen viktig funksjon. PPU-en har to interne flagg som må resettes før hver frame. Det ene er et internt vblank-flagg. Når dette flagget er 1, vil PPU-en sende et NMI-signal til CPU-en. Det vil si at det kan sendes en ny NMI rett etter at den forrige rutinen avsluttet hvis ikke det interne vblank-flagget blir satt til 0. Det andre flagget holder styr på hvor det som skrives til $2006 skal havne (høy eller lav del av adresseregisteret). For å resette disse flaggene må vi lese fra statusregisteret. Det beste stedet å gjøre dette på er i NMI-handleren.

Kapittel 5

Praktisk: Noen enkle program

De forgående kapitlene har gitt oss en god del teoretisk kunnskap. Nå skal vi sette teorien ut i praksis. De eksemplene vi skal gå gjennom i dette kapittelet vil være ting som kommer igjen flere ganger senere. Først og fremst kan det være en god idé å gå tilbake til kapittel 2 og se på eksempelet der. Dette finner du også her: `http://home.no/johan-af/nesguide/eksempel1`. Det eksempelet gir en viss idé om hvordan en typisk kodefil er bygd opp.

5.1 Litt mer om ca65

Før vi går løs på det første «skikkelige» eksempelet skal vi se litt nærmere på ca65.

5.1.1 Labels

ca65 har flere hjelpemidler som hjelper oss å holde styr på koden. Det viktigste er kanskje labels. Som vi så i kapittel 3 er en label et navn som representerer adressen til det stedet i koden der den forekommer. På den måten kan vi referere til steder i koden med navn i stedet for rene adresser.

Høy og lav del av label-adressene

Noen ganger kan det være interessert i å få fatt i bare den høye eller lave delen av adressen som labelen representerer. Til dette har ca65 to *operatorer*: > og <. Førstnevnte velger ut den høye delen av argumentet, mens sistnevnte velger ut den lave delen. Uttrykket `>Test` resulterer altså i den høye delen (pagen) til `Test` mens `<Test` resulterer i den lave delen. Dette kan være hendig når vi ønsker å for eksempel lagre adressen til en label i RAM. Følgende eksempel viser dette:

```
LDA #<Test      ; Gi A den lave delen av adressen til Test
STA $00         ; Lagre i RAM
LDA #>Test      ; Gi A den høye delen av adressen til Test
STA $01         ; Lagre i neste byte i RAM
```

```
; Nå vil bytene $01 og $00 til sammen utgjøre adressen som Test representerer.

JSR ($00)       ; Hopp til den indirekte adressen som starter i $00
```

Denne koden vil i bunn og grunn gjøre akkurat det samme som en enkel `JMP Test`-instruksjon, men bruker indirekte adressering i stedet for absolutt.

Lokale og anonyme labels

Noen labeler er ikke interessante andre steder enn et veldig begrenset område av koden. I en løkke kan det for eksempel være aktuelt å kalle labelen som angir starten av løkken for `loop`. Hvis man så har flere løkker blir det et problem at navnet `loop` allerede er brukt. For å løse dette problemet har ca65 *lokale* labeler. Dette er labeler som kun er gyldige mellom de to nærmeste vanlige labelene. En label defineres som lokal ved å skrive en alfakrøll foran. Følgende eksempel bør vise hvordan dette fungerer:

```
label1:
    ; ...
@loop:
    ; en loop her
    jmp @loop
    ; ...

label2:
    ; ...
@loop:
    ; en annen loop her
    jmp @loop
    ; ...
```

Den første `@loop`-labelen gjelder mellom `label1` og `label2`, mens den andre `@loop`-labelen gjelder etter `label2` og frem til en eventuell neste ikke-lokal label.

Noen ganger har vi med så triviell kode å gjøre at det føles unødvendig å i det hele tatt definere en label. Da kan vi definere såkalte *anonyme* labeler. Disse markeres med et enkelt kolon. For å referere til en anonym label skriver vi et kolon etterfulgt av en serie med + eller --tegn. Dette fungerer slik at `:-` refererer bakover i koden til nærmeste anonyme label, mens `:+` refererer fremover i koden til nærmeste anonyme label. Dette eksempelet viser prinsippet:

```
:    LDA test1      ; # 1
     BNE :-         ; Hopp opp til #1
:    LDA test2      ; # 2
     BEQ :+         ; Hopp ned til #3
     JMP :-         ; Hopp opp til #2
:                   ; # 3
```

Det kan oppgis flere enn bare ett +- eller --tegn. `:---` vil for eksempel referere bakover til den tredje labelen fra det aktuelle stedet.

Koden kan fort bli rotete og vanskelig å lese når man bruker anonyme labels, siden den som leser koden må telle seg fremover og bakover for å finne labelen som det refereres til. Jeg anbefaler å kun bruke anonyme labels i de mest trivielle og korte løkkene der det er helt tydelig hva koden gjør. Det kan for eksempel være nyttig i løkkene der man venter på at PPU-en skal komme i vblank, eller løkken som kopierer en palett over i palett-RAM. I de neste eksemplene vil koden som gjør disse to tingene benytte seg av anonyme labeler.

5.1.2 Segmenter og tilordning av adresser

Hele poenget med å ha labels er at vi slipper å tenke på alt som har med adresser å gjøre, men noen ganger er det nødvendig å diktere hvilken adresse en label skal referere til. For å finne ut hvilken adresse en label representerer tar ca65 utgangspunkt i det *segmentet* den befinner seg i. Deretter teller den hvor mange bytes (instruksjoner, data, eller hva enn det er) som ligger foran labelen og kommer frem til riktig adresse. Segmentene er som vi husker abstrakte inndelinger av adresseområdet. Segmentene er definert i en fil som jeg har valgt å kalle `linker_config.cfg` i denne guiden. Nå skal vi se litt nærmere på den konfigurasjonsfilen som brukes av eksempel 1:

```
MEMORY {

# INES header
    INES_HEADER: start = $0, size = $10, file = %O, fill = yes;

# 16K PRG-ROM
    PRGROM: start = $8000, size = $4000, file = %O, fill = yes;

# 16K CHR-ROM
    CHRROM: start = $0000, size = $2000, file = %O, fill = yes;

# RAM
    RAM_MEM: start = $0000, size = $0800;
}

SEGMENTS {
    INES:       load = INES_HEADER, type = ro;
    CODE:       load = PRGROM, type = ro;
    DATA:       load = PRGROM, type = ro;
    GFX:        load = CHRROM, type = ro;
    VECTORS:    load = PRGROM, type = ro, start = $bffa;
    RAM:        load = RAM_MEM, type = rw;
}
```

Det er to hovedseksjoner, `MEMORY` og `SEGMENTS`. I `MEMORY`-seksjonen deler vi .nes-filen inn i *minneområder*. Hvert minneområde kan så inndeles i segmenter. Som vi ser lages det tre minneområder: `INES`, `PRGROM`, `CHRROM` og `RAM_MEM`. Dette er de fire typene «minne» vi vil ha med å gjøre. Utenom INES-headeren så representerer disse minneområdene de forskjellige minnene i NES-en. Vi skal ikke gå nærmere inn på dette, for vi kommer ikke til å få behov for å endre på noe som helst i `MEMORY`-delen. Vi skal konsentrere oss om `SEGMENTS`. Her defineres segmentene som vi ser med en slik linje som dette:

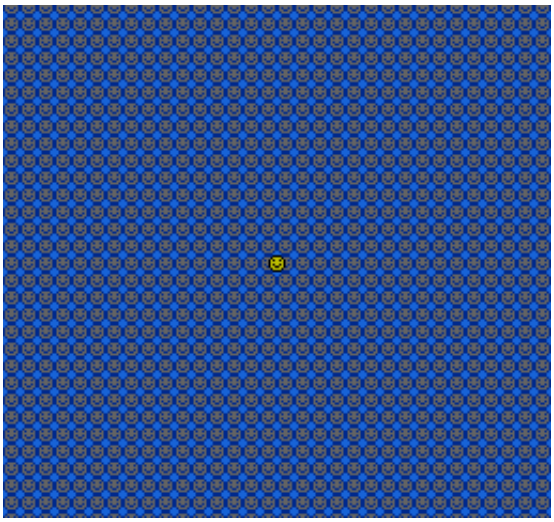
```
NAVN:    load = minneområde, start = startadresse, type = rw/ro;
```

Navnet på segmentet kan ikke være brukt fra før som navnet på et minneområde. I feltet `load` oppgir vi hvilket minneområde segmentet skal plasseres i. I feltet `start` oppgir vi en eventuell startadresse for segmentet, og i feltet `type` oppgir vi entet `rw` (read/write) eller `ro` (read only) (en spesiell type `zp` er forbeholdt zero page).

5.2 Eksempel 2: En sprite

Nå skal vi gjøre en liten utvidelse av eksempelet som vi så i kapittel 2. Vi skal vise noe annet enn bare en farget bakgrunn, og i tillegg skal vi ha en sprite som beveger seg over skjermen.

Eksempelet finner du her: `http://home.no/johan-af/nesguide/eksempel2`.



Figur 5.1: *Eksempel 2*

I dette eksempelet får vi bruk for følgende ting som vi var innom i forrige kapittel:

- Kopiere en 32-bytes palett til palett-RAM
- Bruke DMA til å kopiere en page fra RAM til OAM
- Vente på vblank

Page 2 skal inneholde sprite-dataene som DMA-kontrolleren skal kopiere over til OAM. For å lagre dataene i denne pagen på en sivilisert måte med labels i stedet for adresser må vi gjøre det vi så på i forrige seksjon; lage et eget segment for OAM. Dette gjør vi ved å føye til en ny linje i den gamle `linker_config.cfg` som ble brukt i eksempel 1:

```
OAM:        load = RAM_MEM, start = $0200, type = rw;
```

Her definerer vi altså et nytt segment kalt `OAM` som vi sier skal ligge i minneområdet `RAM_MEM` (RAM), starte i adresse $0200 (page 2) og være både lesbart og skrivbart.

I tillegg må vi fjerne segmentet som før het `RAM`. Nå har vi ikke lenger et segment som dekker hele RAM, så det må bort. I stedet kan vi lage et segment som vi kan kalle `ZEROPAGE`, som vi kan bruke til å lagre eventuelle variabler senere:

```
ZEROPAGE:   load = RAM_MEM, type = zp;
```

Nå kan vi ta for oss selve koden. Koden starter ved at vi venter to frames til PPU-en har fått varmet seg opp. Deretter slår vi av renderingen, slik at vi har fri tilgang til adresseområdene til PPU-en. Paletten lastes inn, og startkoordinatene og tile-nummeret til spriten legges i variablene som er definert i page 2 (som dekkes av segmentet kalt `OAM`). Det er dataene i denne pagen som skal kopieres over i OAM før hver frame. Til slutt legger vi instillinger i $2000 og $2001.

Etter at instillingene er satt, entrer så hovedloopen (labelen «Hovedloop»), som kjøres om og om igjen. Det første hovedloopen gjør er å vente til vi er i vblank. Deretter kopieres page 2 over til OAM med DMA. Til slutt flyttes spriten ved at x-koordinaten inkrementeres med 1. Den endringen vil da fraktes over til OAM i neste frame. Når vi kommer til slutten av koden, hopper vi opp igjen til `Hovedloop`. Da vil CPU-en vente til neste frame, og så kjører loopen på nytt.

I GFX-segmentet legger vi bare én fil, `eks2.spr`. Denne vil da fylle pattern table 1. Pattern table 2 vil bare fylles med 0. I koden har vi satt både sprites og bakgrunner til å hente tiles fra pattern table 1. Dersom ingenting er skrevet til VRAM, vil hver byte være fylt med 0. Det vil si at alle nametablene vil inneholde 0. Da plasseres tile 0 over hele bakgrunnen. Det er derfor smilefjesene vil dekke hele bakgrunnen. Dette er noe å være obs på: Hvis tile 0 ikke er tom, og ingenting er lagt i nametablene, vil bakgrunnen fylles av den første tilen. Det er en vanlig feil å glemme dette. Dersom spillet kommer til å fylle nametablene fullstendig, er ikke dette noe problem, men hvis ikke bør den første tilen i pattern tablet være tom.

5.2.1 Oppgaver

1. Få spriten til å snu retning når den treffer kantene på skjermen
2. Legg inn flere sprites. Hva skjer når det er flere enn 8 én linje?

5.3 Eksempel 3: Input fra kontrollene

En ting vi ikke har sett på enda er hvordan vi kan hente inn tastetrykk fra kontrollene. Det passer ikke så godt inn med hverken CPU-en eller PPU-en, så det kan vi ta nå. Tanken her er å modifisere forrige eksempel slik at vi kan styre spriten med piltastene.

Å kommunisere med kontrollene gjør vi med to adresseregistre: $4016 (kontroll 1) og $4017 (kontroll 2). Disse fungerer på en merkelig måte. Før vi kan lese fra en kontroll må vi først skrive 1 til det tilhørende registeret, og deretter 0 [1]. Deretter kan vi lese hvert tastetrykk ved å lese fra registeret. Hver gang vi leser fra registeret får vi statusen til en ny knapp. Rekkefølgen er som følger: A, B, Start, Select, Opp, Ned, Venstre, Høyre. Den laveste biten av verdien vi leser

[1] Dette forteller en seriell 8-bits latch i kontrollen til å lagre statusen til hver knapp.

forteller om knappen er trykket inn eller ikke. De andre bitene kan være både 1 og 0, så det er viktig at vi fjerner disse fra registeret ved å ANDe med 1. La oss se på hvordan vi leser fra kontroll 1 og reagerer på tastetrykkene:

```
    ; Klargjør kontrollen for å hente data:
    LDA #1
    STA $4016
    LDA #0
    STA $4016

    ; Les tastetrykk:
    LDA $4016          ; A
    LDA $4016          ; B
    LDA $4016          ; Start
    LDA $4016          ; Select
    LDA $4016          ; Opp
    AND #1             ; Fjern alt utenom den laveste biten
    BEQ @SjekkNed              ; Hvis biten er 0, gå til neste knapp
    ; ... her er kode som kjører når Opp er trykket inn ...
@SjekkNed:
    LDA $4016
    AND #1
    BEQ @SjekkVenstre
    ; ... her er kode som kjører når Ned er trykket inn ...
@SjekkVenstre:
    LDA $4016
    AND #1
    BEQ @SjekkHoyre
    ; ... her er kode som kjører når Venstre er trykket inn ...
@SjekkHoyre:
    LDA $4016
    AND #1
    BEQ @Slutt
    ; ... her er kode som kjører når Høyre er trykket inn ...
@Slutt:
```

Merk at vi ikke trenger å sammenligne med 0 før hver `BNE`-instruksjon. Z-flagget settes etter hver instruksjon der resultatet er 0. Det vil si at `BNE` vil hoppe dersom forrige instruksjon var forskjellig fra 0 (Z-flagget er 0).

Eksempel 3 finner du her: `http://home.no/johan-af/nesguide/eksempel3`.

Den eneste forandringen vi har gjort her er å bytte ut den konstante økningen av `x_posisjon` med en kodesnutt som sjekker piltastene og flytter spriten basert på det.

5.3.1 Oppgaver

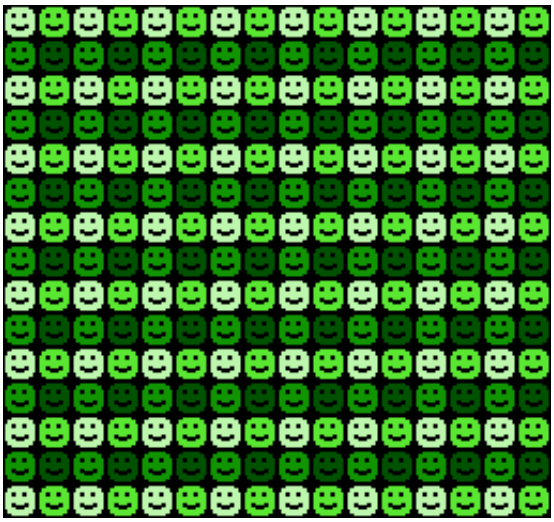
1. Få spriten til å stoppe når den kommer til en av kantene av skjermen
2. Legg inn en ny sprite som kan styres av kontroll 2
3. Gjør det mulig å bytte bakgrunnsfarge ved å trykke på A (neste) eller B (forrige)

5.4 Eksempel 4: En bakgrunn

Nå skal vi ta for oss et eksempel som tegner en enkel bakgrunn. Å tegne en bakgrunn vil egentlig si å fylle et nametable og det tilhørende attribute tablet med data om hvordan bakgrunnen skal se ut. Planen her er å fylle bakgrunnen med blokker på 2 x 2 tiles, siden det er den minste enheten vi kan fargelegge i attribute table. Her skal blokkene bestå av følgende fire tiles (de to første øverst og de to siste nederst):

Figur 5.2: *Tile 0 til 3 i pattern table 1*

Som du sikkert ser fra Figur 5.4 er de to første tilene (0 og 1) den øvre delen av et smilefjes, mens de to neste tilene (2 og 3) er den nedre delen. Vi ønsker å fylle nametablene med smilefjes slik at resultatet blir seende slik ut:



Figur 5.3: *Sluttresultatet*

For å gjøre det må altså første rad i nametablet bestå av `0,1,0,1,0,1,...,0,1`, andre rad av `2,3,2,3,2,3,...,2,3`, tredje rad av `0,1,0,1,0,1,...,0,1`, og så videre. Å fylle nametablene slik kan vi gjøre på flere måter. Det enkleste her er kanskje å lage en *dobbelt løkke*: Vi har en *ytre* løkke som repeterer én gang for hver rad i nametable og en *indre* løkke som fyller raden med tiles. I den ytre løkken må det avgjøres om raden er en odde- eller partallsrad. Hvis det er en partallsrad (rad 0, 2, 4, osv.) skal raden fylles med tile 0 og 1 (øvre del av figuren), mens hvis det er en oddetallsrad skal den fylles med tile 2 og 3.

Koden på neste side viser hvordan dette kan løses. Vi bruker X til å telle hvor mange rader som er igjen, og Y til å telle hvor mange smilefjes som er igjen i den aktuelle raden. Vi har en ytre løkke som starter ved labelen `lagre_rad`. Denne setter Y til 16, og sjekker så om raden er en odde- eller partallsrad. Basert på resultatet av den sjekken lagres enten 0 og 1 eller 2 og 3 i variablene `tile_a` og `tile_b`. Den indre løkken starter ved labelen `lagre_tiles`. Oppgaven til denne løkken er å skrive verdien som ligger i `tile_a` og deretter verdien i `tile_b` til $2007, og

gjenta dette 16 ganger. Y minker med 1 hver gang to tile-numre (dvs. ett smilefjes) er skrevet til nametablet, og så gjentar løkken seg så lenge Y ikke er 0. Når Y er 0 minkes X, og så gjentas den ytre løkken hvis X ikke er 0. Når X er 0 er vi ferdige.

For å oppsummere så gjør koden følgende:

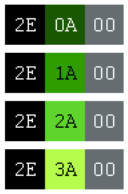
- Sett $2006 til adressen der nametable 1 starter ($2000)
- Sett X til 30 (antall rader)
- Kjør den ytre løkken `lagre_rad`:
  - Sett Y til 16 (antall metatiles på en rad)
  - Sjekk om X er et partall:
    - Hvis X er partall: `tile_a = 0`, `tile_b = 1`
    - Hvis X er oddetall: `tile_a = 2`, `tile_b = 3`
  - Kjør den indre løkken `lagre_tiles`:
    - Skriv `tile_a` til $2007
    - Skriv `tile_b` til $2007
    - Senk Y med 1
    - Gjenta så lenge $Y \neq 0$
  - Senk X med 1
  - Gjenta så lenge $X \neq 0$

Her er selve koden:

```
    lda #$20            ; Nametable 1 starter i $2000
    sta $2006           ; Send adressen til PPU.
    lda #0
    sta $2006

    ldx #30             ; X holder antall rader som er igjen

@lagre_rad:
    ldy #16             ; Y holder antall smilefjes (to tiles) igjen i hver rad
    txa                 ; Flytt X (antall rader) over i A
    and #1              ; Er tallet et partall eller oddetall?
    beq @partall

    ; Hvis raden er en oddetallsrad skal vi skrive tile 2 og 3 (nedre del av figuren)
    ; bortover raden:
    lda #2
    sta tile_a
    lda #3
    sta tile_b
    jmp @lagre_tiles

    ; Hvis raden er en partallsrad skal vi skrive tile 0 og 1 (øvre del av figuren)
    ; bortover raden:
```

```
@partall:
    lda #0
    sta tile_a
    lda #1
    sta tile_b

    ; Her er loopen som lagrer tilene i raden:
@lagre_tiles:
    lda tile_a          ; Hent den første tilen (venstre del av figuren)
    sta $2007           ; Send til PPU-en
    lda tile_b          ; Hent den andre tilen (høyre del av figuren)
    sta $2007           ; Send til PPU-en
    dey                 ; Nå har vi lagret én figur, så vi har ett mindre å ta oss av
    bne @lagre_tiles    ; Hvis Y ikke er 0 så er vi ikke ferdige; gjør det samme igjen.
    dex                 ; Nå har vi unnagjort en rad, så vi senker X med 1
    bne @lagre_rad      ; Hvis X ikke er 0 så har vi flere rader igjen. Hopp opp igjen til
                        ; partall/oddetallssjekken.
```

Dette er som sagt én av flere måter å løse problemet på. Det vi har igjen nå er å fylle attribute table med data om hvilken palett hver blokk på 2 x 2 tiles (smilefjes) skal bruke. Bakgrunnspaletten som er definert i eksempelet er som følger:



Hvis vi ser tilbake på Figur 4.4 og på Figur 5.3 så ser vi at her skal smilefjeset i blokk A ha palett 3 (lysest grønnfarge), blokk B skal ha palett 2 (nest lysest), og så videre. Det betyr at hver byte i attribute table må se slik ut: `00 01 10 11` (husk at attributt-byten er bygd opp slik: `DD CC BB AA`. Den kjappeste måten å lagre denne byten i hele attribute tablet på er ved en enkel loop, tilsvarende den vi bruker for å kopiere en palett til palett-RAM:

```
    lda #%00011011      ; Attributt-byten (00 = 0, 01 = 1, 10 = 2, 11 = 3)
    ldx #64             ; Vi har 64 bytes igjen
@loop:
    sta $2007           ; Send attributt-byten til PPU-en
    dex                 ; Vi har én mindre byte igjen
    bne @loop           ; Hvis X ikke er 0 så hopper vi opp igjen
```

Det er ikke nødvendig å sette en PPU-adresse ved å skrive til $2006 først, for etter at koden som fyller nametable har kjørt, vil riktig adresse allerede ligge i adresseregisteret, siden attribute table starter rett etter nametablet.

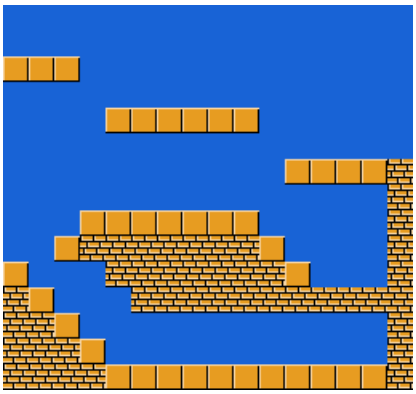
Når disse tingene er gjort, er bakgrunnen lastet inn. Da kan vi aktivere renderingen igjen ved å skrive til $2000 og $2001. En siste ting som må gjøres er å sikre at det ikke er satt noen scroll. Det gjør vi ved å skrive 0 til $2005 to ganger (X og så Y). Den endelige koden ligger her: `http://home.no/johan-af/nesguide/eksempel4`.

5.5 Eksempel 5: Et metatile-system

Siden attribute tablene bare kan fargelegge individuelle blokker på 2 x 2 tiles eller større, er det hensiktsmessig å dele inn grafikken etter denne begrensningen, slik vi gjorde i forrige eksempel. Da kan vi gi hver blokk en egen palett, og samtidig tar informasjonen om hvilke blokker som skal plasseres hvor på skjermen mye mindre plass. Et helt nametable tar 960 bytes, mens en oversikt over hvilke metatiles som skal hvor på skjermen tar til sammenligning $15 \cdot 16 = 240$ bytes (det er 16 blokker på én rekke og 15 rekker). Blokker på 2 x 2 tiles kaller man ofte for *metatiles*. Svært mange NES-spill, da spesielt platformspill, bygger grafikken opp slik.

I dette et eksempelet skal vi implementere et system for å bygge opp bakgrunnsgrafikken med metatiles. Det programmet skal gjøre er å ta et *brett* som forteller hvor metatilene skal plasseres på skjermen, og fylle nametable 1 med de riktige tilene:

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,1,1,1,1,2
0,0,0,0,0,0,0,0,0,0,0,0,0,0,2
0,0,0,1,1,1,1,1,1,1,0,0,0,0,2
0,0,1,2,2,2,2,2,2,2,1,0,0,0,2
1,0,0,0,2,2,2,2,2,2,2,1,0,0,2
2,1,0,0,0,2,2,2,2,2,2,2,2,2,2
2,2,1,0,0,0,0,0,0,0,0,0,0,0,2
2,2,2,1,0,0,0,0,0,0,0,0,0,0,2
2,2,2,2,1,1,1,1,1,1,1,1,1,1,2
```



Sammenhengen mellom tabellen til venstre og figuren til høyre bør være grei å se. Vi ser da for eksempel at metatile 0 her helt blank, mens metatile 2 er mursteiner. I koden vår er vi nødt til å definere hvilke individuelle tiles hver metatile skal bestå av. De ni første tilene i pattern table ser slik ut i dette eksempelet:

Figur 5.4: *Tile 0 til 8 i pattern table 1*

Vi ser da med andre ord at alle de fire tilene i metatile 0 må bestå av tile 8 (den er blank), mens metatile 1 må bestå av de fire første tilene, og metatile 2 må bestå av de fire neste. Det er flere måter å oppgi hvilke tiles metatilene skal bestå av. Én måte kunne for eksempel være at vi gjør noe slikt:

```
metatiles:
    ; metatile 0
    .byte 8, 8, 8, 8

    ; metatile 1
    .byte 0, 1, 2, 3

    ; metatile 2
```

```
    .byte 4, 5, 6, 7
```

Hver rekke definerer da hvilke tiles metatilen består av i rekkefølgen øverst til venstre, øverst til høyre, nederst til venstre og nederst til høyre. Dette systemet gjør noen ting enkle å beregne, men det har den ulempen at det krever noe mer å finne de riktige tilene. Hvis vi for eksempel vil ha tak i tilene til metatile 2 må vi multiplisere metatile-nummeret med fire for å komme til riktig rad. En litt enklere måte, som kanskje ikke virker like elegant, er å gjøre det slik:

```
metatiles_A:
    .byte 8       ; metatile 0
    .byte 0       ; metatile 1
    .byte 4       ; metatile 2

metatiles_B:
    .byte 8       ; metatile 0
    .byte 1       ; metatile 1
    .byte 5       ; metatile 2

metatiles_C:
    .byte 8       ; metatile 0
    .byte 2       ; metatile 1
    .byte 6       ; metatile 2

metatiles_D:
    .byte 8       ; metatile 0
    .byte 3       ; metatile 1
    .byte 7       ; metatile 2
```

Her grupperer vi alle tilene som skal øverst til venstre for seg (A), tilene som skal øverst til høyre for seg (B), og så videre. Fordelen med dette er at om X eksempelvis holder et metatile-nummer og vi er interessert i å hente ut tilen nederst til høyre, gjør vi bare `LDA metatiles_D, x`. Av denne grunnen er det slik det er gjort i dette eksempelet.

La oss nå se på koden som leser brett-dataene og fyller brettet i nametablet. Hovedprinsippet er at vi har et *buffer* på 64 bytes i RAM. Dette bufferet har nok plass til å lagre to linjer med tiles, altså én rekke med metatiles. Koden består av en loop som går sekvensielt gjennom brettet. Den leser inn en byte (metatile-nummeret), og bruker så dette som en indeks inn i metatile-definisjonene (som er som vist ovenfor) til å finne de fire tilene. Disse fire tilene blir så skrevet til bufferet. Når 16 metatiles har blitt skrevet blir bufferet kopiert over til VRAM på samme måte som paletten lastes inn i tidligere eksempler. Deretter fortsetter loopen med neste rad med metatiles, og slik fortsetter det til alle de 240 bytene av brettet er ferdige. Koden ser slik ut:

```
FyllNametable:
    lda #$20                ; Nametable 1 starter i $2000
    sta $2006               ; Sett adressen
    lda #$00
    sta $2006

    ldy #0                  ; Y brukes som indeks i bufferet
    sty brett_indeks        ; brett_indeks er indeksen inn i brettet,
                            ; sett denne til 0.
```

```
@loop:
    ldx brett_indeks        ; X skal være indeks inn i brettet
    lda brett_data, x       ; Hent neste byte fra brettet
    tax                     ; Putt denne i X
    lda metatile_A, x       ; Hent tile nummer X fra listen med A-tiles
    sta buffer, y           ; Skriv til bufferet, indeksert med Y
    lda metatile_B, x       ; Hent tile nummer X fra listen med B-tiles
    sta buffer + 1, y       ; Skriv til neste plass i bufferet
    lda metatile_C, x       ; Hent tile nummer X fra listen med C-tiles
    sta buffer + 32, y      ; Skriv til neste plass i andre linje av bufferet
                            ; (32 bytes bak de andre)
    lda metatile_D, x       ; Hent tile nummer X fra listen med D-tiles
    sta buffer + 33, y      ; Skriv til neste plass i andre linje av bufferet

    iny                     ; Øk Y med 2 (neste metatile i bufferet)
    iny
    inc brett_indeks        ; Gå til neste byte i brettet
    lda brett_indeks        ; Har vi skrevet 16 metatiles
    and #15                 ; til bufferet?
    bne @loop               ; Nei, gjenta prosedyren

    ; Når vi kommer her er vi ferdige med 16 metatiles (en rad)
    ; Da kopierer vi bufferet over til nametablet i VRAM:

    ldy #0
:   lda buffer, y
    sta $2007               ; Samme type loop som vi bruker for å laste
    iny                     ; paletten
    cpy #64
    bne :-

    ldy #0                  ; Sett Y til 0 (første posisjon i bufferet)
    lda brett_indeks
    cmp #240                ; Har vi skrevet ut alle de 240 bytene i brettet?
    bne @loop               ; Nei, begynn på en ny rad

    rts                     ; Ja, returner
```

Her er det noen ting vi må se nærmere på. For å sjekke om det har blitt fylt opp 16 metatiles sjekker vi her om `brett_indeks` er delelig på 16. Det bør være klart at det kun er tilfelle hver 16. gang `brett_indeks` økes. Her sjekker vi det ved å ANDe `brett_indeks` med 15 og se om resulatet er 0. Den kan jo kun være delelig på 16 dersom alle de fire lave bitene er 0!

En annen sak som kan være verdt å merke seg er hvordan vi, i stedet for å øke y mellom hver gang vi skriver til bufferet, heller forandrer på *baseadressen* i instruksjonen. Vi *vet* jo for eksempel at B-tilen alltid kommer rett etter A-tilen, så det er mye enklere å skrive `sta buffer + 1, y` enn det er å først øke Y og deretter gjøre en ny `sta buffer, y`.

Det dette eksempelet ikke tar høyde for er metatiles med forskjellige paletter. Her må alle metatilene benytte seg av den første bakgrunnspaletten, siden det ikke skrives noe til attribute table.

Hele koden til eksempel 5 ligger ute her: `http://home.no/johan-af/nesguide/eksempel5`.

5.5.1 Oppgaver

- Programmer en `FyllNametable`-funksjon som ikke bruker et buffer. Dette er fullt mulig, men krever litt mer jobb.
- Implementer støtte for flere paletter. Hvilken palett metatilene skal bruke kan for eksempel ligge i en liste som indekseres med metatile-nummeret. Koden må essensielt gå gjennom brettet og konstruere de riktige attribute table-bytene (her vil bitshiftingen beskrevet i kapittel 3.2.4 komme til nytte).

Kapittel 6

APU-en

I dette kapittelet skal vi se nærmere på APU-en. Noe av dette stoffet kan være vanskelig å få en oversikt over. I først omgang kan det derfor være lurt å fokusere på å *programmere* APU-en, og deretter gå dypere inn og lese om hvorfor ting er som de er.

6.1 Grunnleggende virkemåte

APU-en har som oppgave å produsere forskjellige analoge lydsignal som kan sendes til TV-apparatet. Lydsignalet er som vi husker fra seksjon **??** en spenning som varierer med tiden. Lydsignalene som APU-en genererer varierer mellom 0 og 5V. Internt er APU-en helt digital. Spenningene som skal genereres er her *kvantisert* til tall mellom 0 og 15 (square-, triangle- og noise-kanalene) eller mellom 0 og 127 (DMC-kanalen). Lydkanalene har hver sin DAC (Digital-to-Analog Converter) som omgjør tallene til korresponderende spenninger mellom 0 og 5V.

På samme måte som CPU-en og PPU-en er APU-en en klokket krets. Hver gang den mottar et klokkesignal (med samme frekvens som CPU-en), skjer det noe innvendig. APU-en er en svært komplisert krets med mange prosesser som skjer parallelt. Mange av disse prosessene er styrt av såkalte *tellekretser*, som normalt kalles *tellere* («counters»). En teller gjør ikke noe annet enn å telle ned fra en startverdi (fiksert eller stillbar) hver gang den mottar en klokkepuls. Når den når 0 sender den ut en puls. Noen tellere i APU-en resetter seg automatisk når de når 0, mens andre forblir 0 til de resettes utenifra.

Tellere er viktige fordi de fungerer som *frekvensdelere*. Hvis en teller gis initialverdien 1000 og den klokkes av en frekvens på 1.66 MHz vil den sende ut en puls hver 1000. gang den klokkes, det vil si at den sender ut et signal med en frekvens på 1.66 kHz. De to square-kanalene og triangle-kanalen har eksempelvis hver sine tellere som brukes til å få en lydbølge med ønsket frekvens. Ved å gi disse tellerene riktig verdi kan vi dele CPU-frekvensen ned til ønsket bølgefrekvens (tone), og lydkanalen vil da generere en tone med denne frekvensen. Noen av lydkanalene har også lengdetellere som teller ned fra en gitt verdi og så demper kanalen, mens noen lydkanaler har såkalte *envelope-generatorer* som demper lyden fra kanalen gradvis over en lengre tid, som også styres av tellere.

Lydkanalene i APU-en har som oppgave å produsere lydbølger basert på data som vi gir dem (volum, lengde, periode, blant annet). Hver lydkanal opererer uavhengig av de andre. Det som kommer ut fra hver lydkanal er som sagt en verdi mellom 0 og 127 for DMC-kanalen og mellom 0 og 15 for de andre kanalene. For å gi data til kanalene har APU-en et sett med 16 registre i adressene fra $4000 til $4018:

Adresser	**Beskrivelse**
$4000 - $4003	Square 1
$4004 - $4007	Square 2
$4008 - $400B	Triangle
$400C - $400F	Noise
$4010 - $4013	DMC
$4015	Kontroll og status
$4017	Frameteller

Registrene som tilhører de forskjellige kanalene skal vi ta for oss når vi ser på hver kanal, og $4017 ser vi på senere.

6.1.1 Kontroll og status

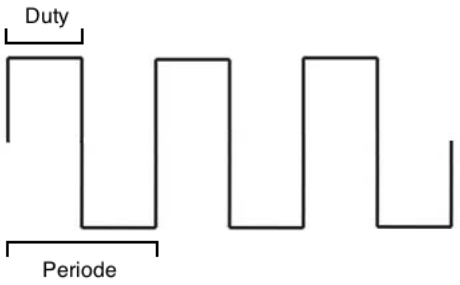
$4015 er det eneste registeret som både kan skrives til og leses fra. Når dette skrives til fungerer det som et kontrollregister, og når det leses fra fungerer det som et statusregister. $4015 har følgende format:

Kontroll ($4015)		
Bit	**Beskrivelse**	**Valg**
0	Square 1	0: av, 1: på
1	Square 2	0: av, 1: på
2	Triangle	0: av, 1: på
3	Noise	0: av, 1: på
4	DMC	0: av, 1: på
6	Frameteller-avbrudd (kun status)	1: avbrudd inntraff
7	DMC-avbrudd (kun status)	1: avbrudd inntraff

De to høyeste bitene eksisterer kun når registeret leses fra. De indikerer da om to typer avbrudd som APU-en kan generere, DMC- og frameteller-avbrudd, har skjedd (mer om dem senere). De laveste bitene brukes enkelt og greit til å aktivere eller deaktivere hver lydkanal når vi skriver til registeret, og de gir oss tilbake hvorvidt hver kanal *genererer lyd* når vi leser fra registeret.

6.2 Square-kanalene

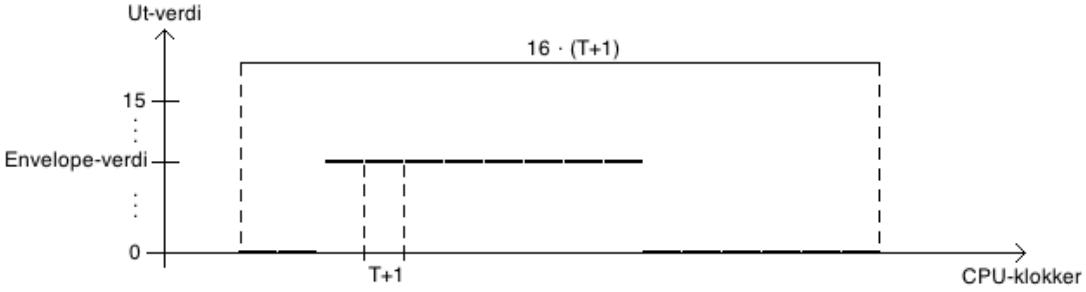
Square-kanalene genererer firkantbølger. Dette er bølger der utsvinget veksler mellom en konstant verdi og 0 (se figur under). Vi kan variere bølgens *duty cycle* mellom fire typer. Det gir fire forskjellige klanger, fra en ganske hul lyd til en mer stikkende og skarp lyd. Duty cycle vil kort sagt angi hvor stor del av bølgeperioden som skal være «på» (forskjellig fra 0). Dette er best illustrert med en illustrasjon:



Duty cycle oppgis i prosent. 25% duty cycle vil eksempelvis si at bølgen er på i 25% av perioden. I illustrasjonen ovenfor er duty cycle 50%. APU-en kan produsere firkantbølger med 12.5%, 25%, 50% og 75% duty cycle.

6.2.1 Bølgefrekvens

Bølgegenereringen styres av en teller som kalles kanalens *timer*. Denne telleren klokkes av CPU-klokken. Signalet som kommer ut fra denne telleren klokker så selve bølgegenereringskretsen. Denne kretsen genererer 1/16 del av én bølgeperiode hver gang den klokkes, og etter 16 ganger er altså bølgeperioden ferdig. Den resulterende bølgen vil altså ha en frekvens som er lik CPU-frekvensen delt på timerens verdi, T, som videre er delt på 16. For å gjøre det hele enda litt mer komplisert har det seg slik at timeren teller ned fra verdien $T + 1$ i stedet for T. Dette gjøres kanskje litt klarere av følgende figur:



Ut fra dette er det klart at frekvensen til bølgen da vil være gitt ved følgende uttrykk:

$$f = \frac{f_{CPU}}{(T+1) \cdot 16}.$$

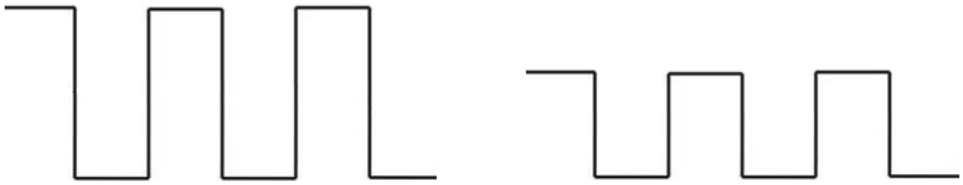
Det er mer interessant med et uttrykk som gir T hvis vi kjenner frekvensen f. Litt algebra på uttrykket ovenfor gir:

$$T = \frac{f_{CPU}}{f \cdot 16} - 1.$$

Forklaringen bak formlene kan nok virke litt innviklet, men det er ikke så viktig for selve programmeringen. Det eneste som er viktig er den siste formelen her, siden den gir oss hvilken verdi vi må sette timeren til for å få en bølge med ønsket frekvens.

6.2.2 Envelope-generatoren

Envelope-generatoren har som oppgave å bestemme lydstyrken til firkantbølgen som kanalen produserer. Den kan enten generere et konstant volum, eller «fade» (dempe gradvis) lydstyrken over en gitt tidsperiode. Envelope-generatoren består av to komponenter: en *envelope-timer* og en *volumteller*. Volumtelleren holder en verdi mellom 0 og 15 som representerer kanalens volum (0 = av, 15 = maks volum). Hvis envelope-generatoren er i konstant volum-modus, kan vi oppgi volumtellerens verdi direkte, og på den måten bestemme kanalens volum:



Hvis envelope generatoren er i dempemodus har volumtelleren alltid en startverdi på 15, og så brukes envelope-timeren til å dekrementere volumtelleren over en gitt tidsperiode. Det resulterer da i en slik bølge:

Envelope-timeren tar en 4-bits startverdi og klokkes med 240 Hz. Hver gang den når 0 blir volumtelleren senket med 1, envelope-timeren settes til startverdien igjen, og nedtellingen gjentar seg til volumtelleren når 0. Sånn for å gjøre det hele mer innviklet starter også envelope-timeren å telle ned fra $T+1$ i stedet for den gitte startverdien T. Sammenhengen mellom envelope-timerens periode T og hvor lang tid dempingen tar, t (i sekunder) blir slik:

$$t_{demping} = \frac{16 \cdot (T+1)}{240} = \frac{2}{15} \cdot (T+1), \quad T = \frac{15}{2} \cdot t - 1.$$

Igjen; forklaringen bak formlene kan helt sikkert virke noe komplisert, men den er heller ikke så viktig i første omgang. Når vi programmerer trenger vi uansett bare disse formlene. Hvis vi for

eksempel ønsker å dempe lyden over en tid på 0.5 sekunder må vi gi envelope-timeren verdien $T = 15/2 \cdot 0.5 - 1 = 2.75 \approx 3$.

6.2.3 Lengdetelleren

Lengdetelleren har en noe lignende virkemåte som envelope-generatoren, men denne demper ikke volumet gradvis. Den kutter i stedet kanalen av når den har telt ned til 0. Dette kan være hendig dersom kanalen brukes til musikk, men i praksis er det ofte like greit å dempe kanalen manuelt. Lengdetelleren klokkes med 120 Hz, og holder en 7-bits initialverdi. Denne initialverdien kan i midlertid ikke oppgis direkte, slik som det er tilfellet med envelope-telleren. Vi må i stedet oppgi en 5-bits verdi, som brukes som indeks i følgende tabell:

10	254	20	2	40	4	80	6	160	8	60	10	14	12	26	14
12	16	24	18	48	20	96	22	192	24	72	26	16	28	32	30

Hvis vi for eksempel skriver 18 til lengdetelleren vil den altså få 24 som startverdi. Da vil tonen vare i $24/120 = 0.2$ sekunder.

Det er en litt merkverdig sammenheng mellom lengdetelleren og envelope-generatoren. Dersom lengdetelleren er deaktivert og envelope-generatoren er i dempemodus, vil dempingen gjenta seg automatisk. Det vil si at så snart én demping er ferdig, settes volumtelleren til 15 igjen, og dempingen starter på nytt.

6.2.4 Registrene

De fire registrene fra $4000 til $4003 og fra $4004 til $4007 kontrollerer henholdsvis square 1 og square 2. I $4000/$4004 velger vi duty cycle, om lengdetelleren skal være aktivert, hvilken modus envelope-generatoren skal operere i, og en 4-bits verdi som enten brukes som volum eller periode for envelope-timeren:

$4000 og $4004		
Bit	**Funksjon**	**Valg**
0-3	Envelope-lengde / volum	0 - 15
4	Envelope-modus	0: konstant volum, 1: demping
5	Lengdeteller/envelope-looping	0: aktiver/ikke loop, 1: deaktiver/loop
6-7	Duty cycle	0: 12.5%, 1: 25%, 2: 50%, 3: 25% (omvendt)

I $4001 og $4005 skriver vi instillinger som har med sweep-enheten å gjøre. Denne skal vi se på senere.

I $4002 og $4006 gir vi de lave 8 bitene av timerens periode:

I $4003 og $4007 gir vi de høye 3 bitene av timerens periode, samt hvor lenge lengdetelleren skal telle ned før den kutter av lyden (hvis den er aktivert):

$4003 og $4007	
Bit	**Funksjon**
0-2	Høy del av periode
3-7	Indeks i lengdetabell

Dette er mye teori. La oss se på et eksempel:

```
LDA #%00000011      ; aktiver Square 1 og Square 2
STA $4015

LDA #%10101111      ; duty cycle 50%, deaktivert lengdeteller
                    ; deaktivert envelope-generator, fullt volum (15)
STA $4000

LDA #235            ; t = 235 gir f = ca. 440 Hz (en A) for PAL-systemer
STA $4002           ; lagre lav del

LDA #0              ; 235 er under 255, så høy del er bare 0
STA $4003           ; lagre høy del (lengde-bitene blir 0, dvs. indeks 0 i tabellen,
                    ; men lengdetelleren er deaktivert uansett)
```

Her anbefaler jeg å leke seg litt med de forskjellige instillingene vi har sett på. Prøv for eksempel å aktivere envelope-generatoren. Hvordan høres det ut? Prøv også å sette Square 2 til å generere en bølge med *nesten* samme frekvens. Hva skjer med lyden? [1]

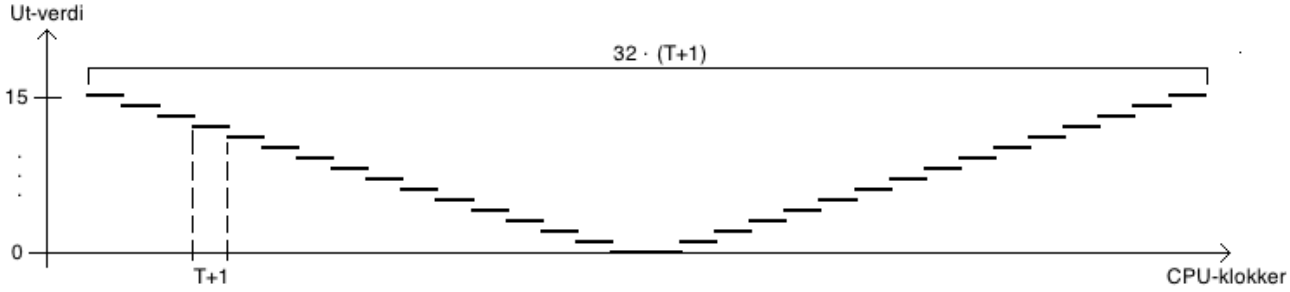
6.3 Triangle-kanalen

Triangle-kanalen lager såkalte triangelbølger, som vi så et bilde av i kapittel 1. Disse har en karakteristisk «hul» lyd, og passer godt til bass (den velkjente NES-bassen), eller fløytelignende instrumenter. I motsetning til square-kanalene har vi ikke mulighet for å endre på utseendet til triangelbølgene, slik vi kunne med firkantbølgenes duty cycle. Triangelkanalen har heller ingen envelope-generator, og ikke mulighet for å sette et ønsket volum.

6.3.1 Bølgefrekvens

Som i square-kanalene drives bølgegenereringen i denne kanalen også av en 11-bits timer som klokkes av CPU-frekvensen, men verdiene vi til denne vil ikke resultere i bølger med samme frekvens som i square-kanalene. Dette skyldes at bølgegenereringen her foregår på en annen måte som tar 32 trinn. Hver gang timeren teller ned lages et nytt trinn i triangelbølgen. For hvert 32. klokkesignal ut fra timeren er det altså generert én bølgeperiode. Følgende figur kan hjelpe:

[1] Dette kalles detuning og er en mye brukt teknikk i synthesizere for å få en «fetere» lyd.



Relasjonene mellom bølgefrekvensen og timer-verdien T blir nå:

$$f = \frac{f_{CPU}}{(T+1)\cdot 32}, \quad T = \frac{f_{CPU}}{32\cdot f} - 1.$$

I praksis betyr dette at om vi gir triangle-kanalen og en square-kanal samme timer-verdi, vil triangle-kanalen være én oktav lavere enn square-kanalen.

6.3.2 Lengdetelleren og lineærtelleren

I likhet med square-kanalene har triangle en lengdeteller. Denne fungerer på akkurat samme måte: Vi skriver en 5-bits verdi til registeret, som brukes som en indeks inn i periode-tabellen ovenfor til å finne en 7-bits startverdi. Lengdetelleren klokkes også her med 120 Hz.

I tillegg har triangle-kanalen en *lineærteller* («linear counter»). Denne har akkurat samme oppgave som lengdetelleren, å telle ned fra startverdien og så kutte av lyden, så det virker noe rart at triangle-kanalen har begge deler. Lineærtelleren kan gis en 7-bits verdi direkte, og den klokkes dobbelt så raskt (240 Hz) som lengdetelleren.

Begge tellerne er enten aktive (teller nedover når de klokkes) eller inaktive *samtidig*. Kanalen blir stille med en gang en av tellerne er 0. Dette har to viktige implikasjoner:

- Den telleren som bruker lengst tid er overflødig, for kanalen vil allerede være stoppet av den andre telleren. En vanlig praksis i trianglekanalen er å sette lengdetelleren til 254 (indeks 1). Da vil lineærtelleren alltid bruke minst tid, uansett hvilken verdi den får.

- Dersom vi ønsker å styre tonenes lengde manuelt, må vi gi lineærtelleren en verdi, selv om den ikke er aktiv (lengdetelleren har alltid en verdi forskjellig fra 0 (se på tabellen ovenfor), så den trenger vi ikke å tenke på.)

6.3.3 Registrene

De tre registrene $4008, $400A og $400B kontrollerer triangle-kanalen. $4009 er ubrukt. I $4008 angir vi hvorvidt både lengdetelleren og lineærtelleren skal aktiveres, og en 7-bits startverdi for lineærtelleren:

$4008		
Bit	**Funksjon**	**Valg**
0-6	Startverdi for lineærtelleren	0-127
7	Lengdeteller/lineærteller	0: aktiver, 1: deaktiver

I $400A skriver vi de 8 lave bitene av timerens periode.

I $400B skriver vi de 3 lave bitene av timerens periode, samt de 5 indeks-bitene til lengdetellerens tabell:

$400B		
Bit	**Funksjon**	**Valg**
0-2	Høy del av periode	-
3-7	Indeks i lengdetabell	0 - 31

La oss se på et kort eksempel der vi bruker triangel-kanalen:

```
LDA #%00000100      ; aktiver triangle
STA $4015

LDA #%10000001      ; deaktiver tellerne, sett lineærtelleren
                    ; til noe annet enn 0
STA $4008

LDA #235            ; t = 235 gir f = ca. 220 Hz (PAL)
STA $400A           ; lagre lav del av perioden

LDA #0              ; vi har ingen høy del
STA $400B
```

Her er det ikke så mye mer å se på. En liten oppgave: Få triangelkanalen til å spille av en tone på 1234 Hz som varer i akkurat et halvt sekund.

6.4 Noise-kanalen

Noise-kanalen kan generere forskjellige former for *støy*. Dette er lydsignaler der det ikke er et enkelt, gjentakende mønster som repeterer. Noise-kanalen har en lengdeteller og en envelope-generator, der begge fungerer helt likt som i square-kanalene. Disse er forklart i seksjon 6.2.4 og 6.2.3.

6.4.1 Støytyper og frekvens

APU-en lager støy ved å sende ut en strøm av tilnærmet tildfeldige biter fra en spesiell *tilfeldighetsgenerator*. Når biten er 1 sendes envelope-verdien ut fra kanalen, ellers er kanalens verdi

0. Dette resulterer på sett og vis i en firkantbølge der duty cycle varierer tilsynelatende tilfeldig. Tilfeldighetsgeneratoren kan generere en tilfeldig sekvens på 93 biter, eller en på 32767 biter. Velger vi en 93-bits sekvens vil lyden få en mer tone-aktig karakter, siden sekvensen gjentar seg relativt ofte. Velger vi 32767 vil lyden høres ut som hvit støy.

APU-en har en 11-bits timer slik de andre kanalene har. I stedet for å klokke en bølgegenereringskrets, klokker denne i stedet tilfeldighetsgeneratoren. Det vil si at hver gang den teller ned sendes en ny bit ut fra tilfeldighetsgeneratoren. Denne timeren kan *ikke* gis en direkte verdi av oss. Den fungerer mer likt lengdetelleren som vi har sett på i de andre kanalene: Vi oppgir en 4-bits verdi som så brukes som en indeks inn i følgende tabell:

4	8	14	30	60	88	118	148	188	236	354	472	708	944	1890	3778

Det gir ikke mening å snakke om bølgefrekvens for noise-kanalen på samme måte som for de andre kanalene. De forskjellige periodene man kan gi timeren i noise-kanalen vil heller endre på støyens *klang*. Her er det best å prøve seg litt frem med de forskjellige instillingene for å høre hvordan de 16 forskjellige valgene påvirker lyden.

6.4.2 Registrene

De tre registrene $400C, $400E og $400F kontrollerer noise-kanalen. $400D er ubrukt. I $400C angir vi akkurat det samme som i de tilsvarende registrene $4000 og $4004 for square-kanalene, utenom duty cycle:

$400C		
Bit	**Funksjon**	**Valg**
0-3	Envelope-lengde / volum	0 - 15
4	Envelope-modus	0: konstant volum, 1: demping
5	Lengdeteller/envelope-looping	0: aktiver/ikke loop, 1: deaktiver/loop

I $400E oppgir vi hvilken sekvenslengde tilfeldighetsgeneratoren skal bruke (som da vil gi forskjellig lydkarakter), samt en 4-bits indeks inn i periodetabellen som vi så ovenfor:

$400E		
Bit	**Funksjon**	**Valg**
0-3	Periode-indeks	0 - 15
7	Sekvenslengde	0: 32767 bits, 1: 93 bits

I $400F angr de 5 høyeste bitene lengdetellerens verdi, akkurat slik som i de tilsvarende registrene $4003, $4007 og $400B til square- og triangle-kanalene:

$400F		
Bit	**Funksjon**	**Valg**
3-7	Indeks i lengdetabell	0 - 31

Igjen kan vi ta et kort eksempel:

```
LDA #%00001000      ; aktiver noise
STA $4015

LDA #%00100111      ; deaktiver lengdeteller, sett envelope
                    ; til demping, nest kortest hastighet
STA $400C

LDA #$0C            ; 32767-bits sekvens, periode nr. C i tabellen
STA $400E

LDA #0              ; sett lengde til indeks 0 (bare for å sette den
                    ; til noe)
STA $400F
```

6.5 DMC

DMC-kanalen er uten tvil den mest avanserte lydkanalen. DMC er kort for *Delta Modulation Channel*. Som navnet tilsier kan denne kanalen generere lydbølger ved såkalt *deltamodulasjon*. Men den kan også generere lydbølger ved at prosessoren overtar bølgegeneratorens rolle og direkte gir kanalverdien som sendes til DAC-en. Vi skal se på begge måtene å bruke kanalen på.

6.5.1 Pulskodemodulasjon og deltamodulasjon

Den vanligste måten å representere lyd digitalt på er å ta «stillbilder», kalt *samples*, av lydbølgen med jevne (og veldig små) mellomrom, og konvertere bølgens utsving (amplitude) på det tidspunktet til en korresponderende verdi. Denne metoden kalles pulskodemodulasjon eller PCM (Pulse Code Modulation) og er den mest vanlige måten å lagre lyd digitalt på. Det krever i midlertid relativt mye plass for å gjøre dette. Hvis vi for eksempel lagrer utsvinget som en 8-bits verdi og gjør dette 8000 ganger i sekundet (som omtrent er det laveste man kan gå for å få brukbar lyd), vil ett sekund med lyd altså ta 8 KiB. Det er her deltamodulasjon kommer inn.

Deltamodulsjon går ut på at vi har en et register eller en teller som til en hver tid representerer bølgens utsving. Lyddataene består så av en strøm av biter der 1 betyr at telleren skal økes med 1, og 0 betyr at den skal senkes med 1. Hvis dette skjer raskt nok kan en brukbar tilnærming av det virkelige lydsignalet lages.

Ordliste

CHR-ROM Character-ROM, minnet der tilenes grafikk er lagret. 7

CPU Central Processing Unit, prosessoren i maskinen. 6, 8, 25

IRQ Interrupt Request, et avbrudd som kan ignoreres. 43

NMI Non-Maskable Interrupt, et avbrudd som ikke kan ignoreres. 43

NTSC National Television System Committee, en TV-standard. 6

OAM Object Attribute Memory, et minne der figurinformasjon lagres. 13, 59

PAL Phase Alternating Line, en TV-standard. 6

PPU Picture Processing Unit, grafikkprosessoren i maskinen. 6

PRG-ROM Program-ROM, minnet der programkoden er lagret. 7

RAM Random Access Memory, en lagringsplass. 6

VRAM Video RAM, PPU-ens RAM der nametable og attribute table lagres. 6