**x**

**Chip8M8**

**Våren 2016**

**DAT111**

**Are Jevne**

**Amar Duranovic**

**Daniel Krogstad**

**Sindre Gjelsten**



# Obligatorisk egenerklæring/gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

|  |  |  |
| --- | --- | --- |
| 1. | **Jeg/vi erklærer herved at min/vår besvarelse er mitt/vårt eget arbeid, og at jeg/vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.** | ☑ |
| 2. | **Jeg/vi erklærer videre at denne besvarelsen:**   * ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. * ikke refererer til andres arbeid uten at det er oppgitt. * ikke refererer til eget tidligere arbeid uten at det er oppgitt. * har alle referansene oppgitt i litteraturlisten. * ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse. | ☑ |
| 3. | **Jeg/vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. [Universitets- og høgskoleloven](https://lovdata.no/dokument/NL/lov/2005-04-01-15) §§4-7 og 4-8 og [Forskrift om eksamen](https://lovdata.no/dokument/SF/forskrift/2005-06-22-833)** §§ 31. | ☑ |
| 4. | **Jeg/vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.** | ☑ |
| 5. | **Jeg/vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens [retningslinjer for behandling av saker om fusk](http://www.uia.no/student/eksamen/linkeblokksamling/under-eksamen/fusk).** | ☑ |
| 6. | **Jeg/vi har satt oss inn i regler og retningslinjer i bruk av [kilder og referanser på biblioteket sine nettsider](http://www.uia.no/bibliotek/hjelp-og-veiledning/riktig-kildebruk-ved-uia).** | ☑ |
| 7. | **Vi har i plenum blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt.**  Ordinært vurderes alle deltakere i prosjektet samlet. | ☐Ja  ☑Nei |

# Publiseringsavtale

**Fullmakt til elektronisk publisering av oppgaven**

Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).

Oppgaver som er unntatt offentlighet eller tausehetsbelagt/konfidensiell vil ikke bli publisert.

|  |  |
| --- | --- |
| **Jeg/vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:** | ☑ Ja  ☐ Nei |
| **Er oppgaven båndlagt (konfidensiell)?**  **(**Båndleggingsavtale må fylles ut) | ☐ Ja  ☑ Nei |
| **Er oppgaven unntatt offentlighet?**  (inneholder taushetsbelagt informasjon. Jfr. Offl. §13/Fvl. §13) | ☐ Ja  ☑ Nei |

**Forord**

Hensikten med følgende rapport er å gi et innblikk i hvordan en CHIP 8 Emulator fungerer, samt gi en generell forståelse på prosessen bak koding og testing. Gruppen ble formet av Amar Duranovic, Are Jevne, Daniel Krogstad og Sindre Gjelsten. Vi retter en takk til Christian Auby som har hjulpet oss gjennom prosjektet fra starten av.

**Sammendrag**

Hovedmålet med prosjektet var å lage en chip8 emulator til PC. Den skulle kunne kjøre spill som Pong, Tetris og Space invaders. Vi jobbet individuelt med operasjonskodene, men vi satt som oftest samlet enten via Skype eller på skolen. Det funket veldig bra, men det ble ikke lagt altfor mange timer inn i prosjektet ved starten.

Da vi hadde fått operasjonskodene ferdige for første gang, testa vi og fant en del feil. Blant annet at Tetris stoppa når første brikke kom til bunnen. Random funksjon produserte bare samme loopen hver gang. Alt dette blir bedre beskrevet i testing lengere nede. Vi kan fort konkludere med at dette var et meget suksessfylt prosjekt til slutt. Det tok bare litt tid å starte.

Innholdsfortegnelse

[Innledning 3](#_Toc452567667)

[**Chip 8 Emulator** 3](#_Toc452567668)

[**Tastatur og grafikk** 3](#_Toc452567669)

[**8XY1-Sekvensdiagram** 4](#_Toc452567670)

[Bakgrunn og Teori 5](#_Toc452567671)

[Prosess 5](#_Toc452567672)

[**Fase I** 5](#_Toc452567673)

[**Fase II** 5](#_Toc452567674)

[**Fase III** 5](#_Toc452567675)

[**Fase IV** 5](#_Toc452567676)

[**Fase V** 6](#_Toc452567677)

[**Fase VI** 6](#_Toc452567678)

[**Fase VII** 6](#_Toc452567679)

[**Fase VIII** 6](#_Toc452567680)

[Testing 7](#_Toc452567681)

[**Testing av opkoder** 7](#_Toc452567682)

[**Feilsøking** 9](#_Toc452567683)

[Timeliste 12](#_Toc452567684)

[Drøfting 13](#_Toc452567685)

[**Hva gikk galt?** 13](#_Toc452567686)

[**Gruppen** 13](#_Toc452567687)

[**Forandringer til neste prosjekt** 14](#_Toc452567688)

[Konklusjon 14](#_Toc452567689)

[Ordliste 14](#_Toc452567690)

[Kilder 14](#_Toc452567691)

# **Innledning**

Chip 8 ble først tatt i bruk av COSMAC VIP og TELMAC 1800 8-bits Mikro-komputere på midten av 1970-tallet. Chip 8 programmer blir kjørt på en chip 8 virtuell maskin. Grunnen til at den ble laget var for å gjøre spill enklere å programmere på tvers av datamaskiner. Chip 8 kjører mange klassiske spill som for eksempel Pong, Space Invaders, Tetris og Pac-man. Vi arbeidet sammen for å få frem produktet og møtte opp til avtalte timer for å få gang i sakene. Vi hadde mange spørsmål innad i gruppen slik som, hva er en chip 8 emulator egentlig? Og hva kan vi lære fra det? Dette er spørsmål som ble svart relativt kjapt i arbeidsprosessen og det bringte oss videre til å utvide egenskapene til Chip-8.

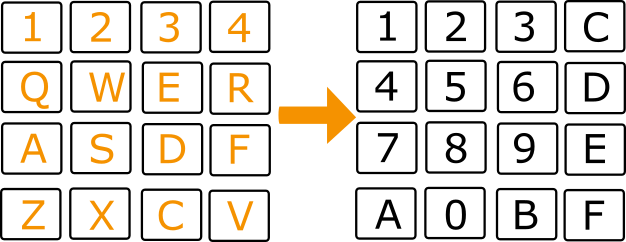
## **Chip 8 Emulator**

En emulator er et dataprogram som lar oss etterligne designet og funksjonaliteten til et datasystem (System X). Emulatoren lar oss kjøre software ment for System X på et helt anderledes datasystem (System Y). For eksempel så kan vi kjøre en gameboy emulator (System X) på vår lokale PC (System Y) for å spille gameboy spill.

Chip 8 emulatoren vi lager etterligner maskinen som ble laget på midten av 1970-tallet og lar oss kjøre Chip 8 spill på pcen vår via emulatoren. En chip 8 er bygget opp av 16 8-bits registere ved navn V0 til VF. Chip 8 har også to timere som teller ned på en felles 60 hertz klokke, helt til de når 0. De forskjellige timerene er:  
  
- Delay timer: Denne brukes til timing-relaterte hendelser i spill. Verdien kan settes og leses.  
- Sound timer: Denne brukes for lydeffekter. Når verdien ikke er 0 så lager den en pipelyd.

## **Tastatur og grafikk**

Chip 8 bruker et heksadesimalt tastatur med 16 taster som går fra 0 til F. Tastene ‘8’, ‘4’, ‘6’ og ‘2’ blir typisk brukt for bevegelse. Tre operasjonskoder blir brukt til å detektere knappetrykk. Vi går mer inn på operasjonskoder i neste del av rapporten. I emulatoren vår brukte vi keymappet under:



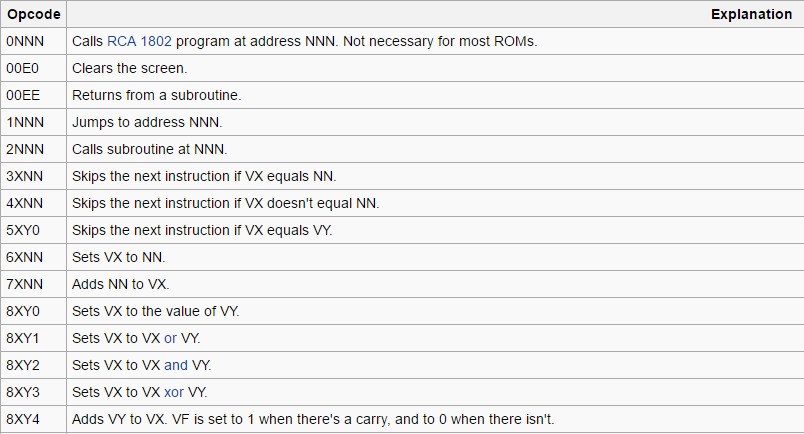
*(Figur 1.1) Utsikt over knappene vi bruker på tastaturet i forhold til hva de er på Chip-8 sitt Hex Tastatur.*

Et originalt chip 8 display er 64x32 piksler og har monokromt fargepalett. I vår chip 8 la vi til ekstra farger via biblioteket OpenGL. Grafikken i chip 8 blir tegnet med noe som heter ’sprites’. Sprites er to-dimensjonale figurer som blir satt sammen til å vise forskjellige objekter. I chip 8 tegner vi inn på skjermen for å få sprites.

## 

**Operasjonskoder**

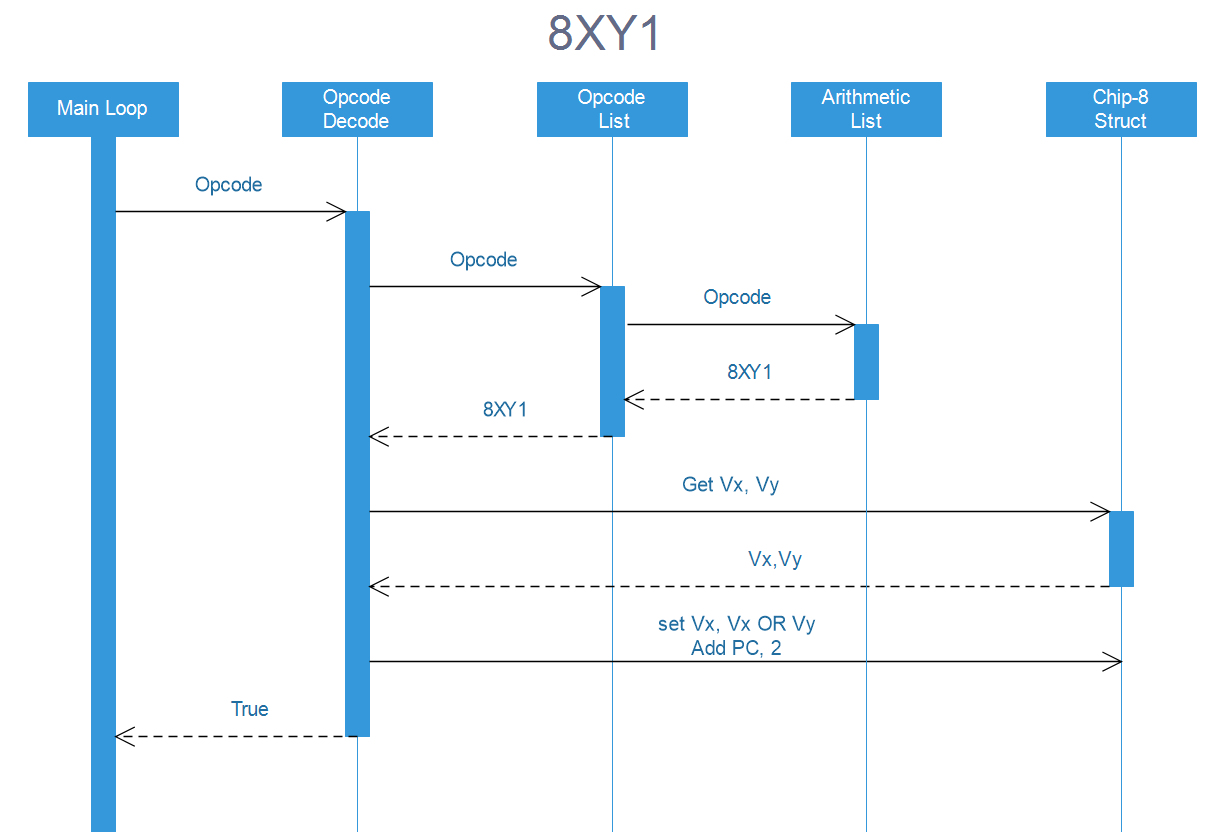
En operasjonskode er delen av en maskinkode-instruksjon som spesifiserer hvordan en operasjon skal utføres. Instruksjonene spesifiserer vanligvis dataen de vil arbeide med, i form av matematiske operasjoner. Chip 8 har 35 forskjellige operasjonskoder som styrer alle funksjonene til maskinen.

*(Figur 1.2) En liste over de 15 første operasjonskodene i Chip 8.*

Operasjonskoder er delt opp i forskjellige hovedgrupper. Disse gruppene er sortert etter funksjonaliteten. I figur 1.2 så ser vi for eksempel at operasjonskodene 3XNN til 5XY0 omhandler samme type operasjoner, men med forskjellige parametere. For eksempel 5XY0, når kjørt, hopper over neste instruksjon hvis register VX er det samme som register VY.

## **8XY1-Sekvensdiagram**

Nå som vi har litt forståelse for hvordan en Chip 8 emulator er bygget opp, så ser vi på et sekvensdiagram for en operasjonskode. Ut ifra figur 1.2 ser vi at 8XY1 har beskrivelsen: Sets VX to VX or VY.

*(Figur 1.3) Sekvensdiagram for operasjonskoden 8XY1*

Vi ser for oss at VX = 10110011 og VY = 00111010 før denne operasjonskoden tar sted. Main loopen sender operasjonskode-signalet videre til dekoderen. Dekoderen sender så videre hvor da listen detekterer det første tallet. Ut ifra hva det første tallet er (I dette tilfelle 8) så sendes den videre til hvor den hører til. 8XY1 utfører en aritmetisk operasjon, så den havner i ‘aritmetic’ gruppa, som er en av de samlede gruppene operasjonskoder i vårt program.

Det er her programmet detekterer det siste tallet for å signalisere hvilken operasjon som skal utføres, før operasjonskoden sendes tilbake til dekoderen. Dekoderen tar da og sender ut X og Y verdiene (bitverdiene) som skal inn på registrene. Her detekteres hvilke registre som verdiene skal puttes inn i. Til slutt utfører programmet, operasjonen og setter VX til å være VX or VY. I vårt tilfelle blir det 10110011 OR 00111010 som gir resultatet 10111011. Når verdien er på plass så sender den ‘true’ tilbake til main, som signaliserer at operasjonen er utført.

# **Bakgrunn og Teori**

En emulator bruker operasjonskoder for å fungere. Det er egentlig bare en instruksjon som datamaskinen (CPU) skal gjøre. Chip 8 har 35 operasjonskoder, som deles inn i noen forskjellige grupper utfra hva de gjør. SDL2 er et biblotek for å få grafikk til programmer. SDL2 må ha intilialize og setup, og bindes opp med openGL. Det er skrevet i C, og er derfor midt i blinken for prosjektet vårt.

C er et lavnivå programmeringsspråk. Det ble introdusert i 1972, og er det mest brukte språket gjennom tidene. Det finnes flere standarder innen C, men vi brukte C99. Git bruker også C. Git er et versjonskontrollsystem som ble utviklet i 2005 av Linux kernel utviklere.

# **Prosess**

## **Fase I**

Programmet I seg selv begynte som en switch-case struktur med en variabel som du kunne hardkode inn en enkelt operasjonskode. Dette var en midlertidig løsning som var langt fra optimal men krevde veldig lite arbeid og lot oss begynne med litt fundemental testing. En struct ble laget for å sende all data til funksjonene. Arbeidet på operasjonskodene begynte effektivt og vi fikk unnagjort en god mengde av de simplere operasjonskodene. På grunn av mangel på debugging potensiale så var operasjonskode-testene ikke de beste, men siden disse var relativt enkle så var det bare et par som ga oss tilbakeslag.

## **Fase II**

Det neste steget var og få satt opp en fungerende dekoder som var ryddig og oversiktlig. Vi leste oss opp på noe som heter funksjonspekere. Eksemplene vi fant virket veldig rotete siden man måtte tilegne en variabel til hver funksjon, noe vi ikke ønsket. Etter å ha undersøkt litt, gikk vi bort ifra switch-case modellen, og heller hadde operasjonskodene i boolske funksjoner. Pekere til disse funksjonene er igjen lagret i en matrise hvor vi sender alle instruksjonene. Dette økte hastigheten, samtidig som det er mer ryddig og oversiktlig for oss å arbeide videre med.

## **Fase III**

Nå som strukturen er i orden, måtte vi finne en god måte å lese operasjonskodene på. På noen av de enklere instruksjonene kunne det fungert å bare kopiere hele operasjonskoden inn, men dette skapte problemer med en gang man skulle lese/skrive til minne eller andre registre. Operasjonskodene er på to byte, mens minnet er på en. På grunn av dette endte vi opp med å laste inn først en byte, skifte den 8 bit til venstre, for så å laste neste byte med en ‘eller’-operasjon.

## **Fase IV**

Det dukket også opp noen problemer med FX-kodene. Det er totalt ni, og siden flere av de endte på samme siffer, ble de vanskeligere å skille fra hverandre enn for eksempel 8-serien. Måten det ble løst på var å lage en 2d matrise hvor hver dimensjon tar for seg et av de siste to sifferene. Dette for å skille operasjonskodene fra hverandre, så de utfører riktig instruksjon.

## **Fase V**

For å kunne lese inn flere operasjonskoder, trenger vi en ROM-loader. På den måten kan vi lage test-ROMS, og når tiden kommer, laste inn spill som Pong. ROM-Loadern leser fra ROM-filer til en buffer ser om ROMen får plass i minnet og kopierer deretter til hoved minnet byte for byte i en løkke.

Underveis fikk vi problemer med SDL som førte til at prosjektet kom til en liten stopp. Istedefor og lære oss å sette opp et vindu via SDL laget vi en SDL/openGL hybrid siden vi hadde noen forkunnskaper. Denne ble brukt for å tegne figurer på skjermen.

## **Fase VI**

Vi fikk på plass en debugger som gjorde at vi kunne se flyten i programmet. Dette hjalp oss til å finne noen feil vi var uvitende om, slik som at operasjonskode DXYN ikke alltid satte kollisjonsflaget riktig, som gjorde at tetris brikker fallt gjennom gulvet slik som vist i testing delen av rapporten.

## **Fase VII**

Når vi skulle teste forskjellige spill fant vi at noen kræsjet med en gang på visse steder. Etter å ha sett på det konkluderte vi med at det måtte ha noe med grafikken og gjøre. Vi fant ut at det var relatert til tegning av høye sprites på det laveste nivået. Av en eller annen grunn ville programmet prøve å tegne utforbi grensen (altså utfor skjermen). Dette skapte mye problemer fordi vi allerede hadde gitt beskjed til renderen om at den ikke skulle tegne utforbi området. Vi fant ut at DXYN operasjonskoden prøvde å skrive utforbi området til GFX matrisen. Vi hadde bare sørget for at den første raden ikke tegnte utforbi området, men ikke for de neste radene. Dette førte til at matrisen ble korrupt og SDL kræsjet. Vi fikser dette problemet ved og få DXYN til å slutte og tegne hvis rekken den tegner pluss den neste rekken var over 32\*64(Resolusjonen til chip 8).

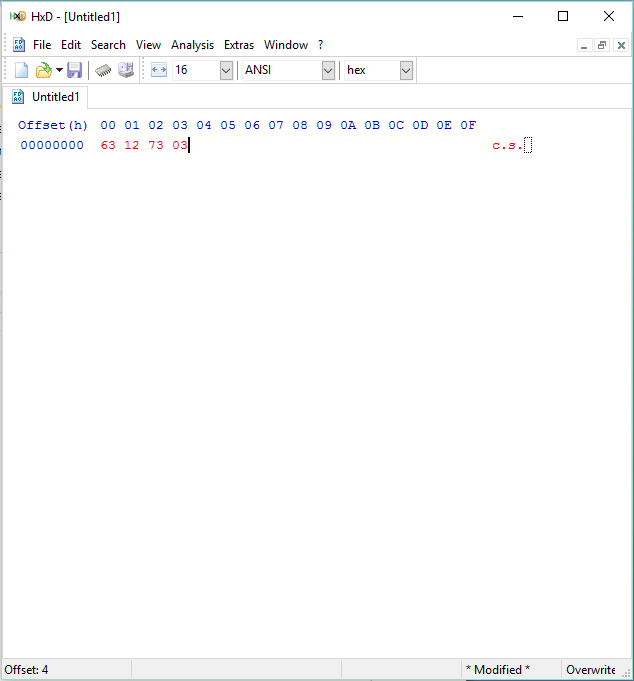
## **Fase VIII**

På dette tidspunktet hadde vi alle operasjonskodene til en viss grad. Både display, input og ROM loading fungerer til å starte noen spill. Nå kunne vi kjøre spill og fant noen store feil i operasjonskodene. De ble fort fikset og spillene kunne endelig kjøre nesten helt perfekt. Det gjenstod bare noen problemer med at noen spill kjørte utrolig sakte, mens andre virket for raske. Invaders (En klone av Space Invaders), hvor romvesenene gikk altfor fort, som gjorde det umulig å spille. Dette ble løst ved å se hvor mye tid som hadde gått mellom hver syklus. Hvis det hadde gått tilstrekkelig tid, så itereres tellerne. Når dette ble løst kjørte spillene mye bedre og vi hadde endelig en fullkommen chip-8.

# 

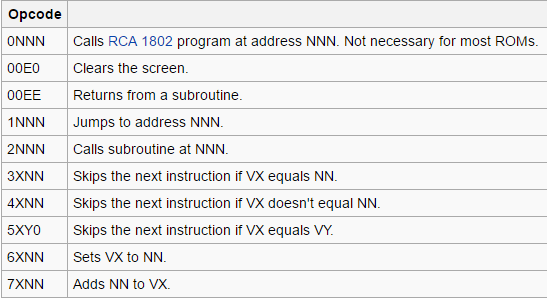
# **Testing**

## **Testing av opkoder**

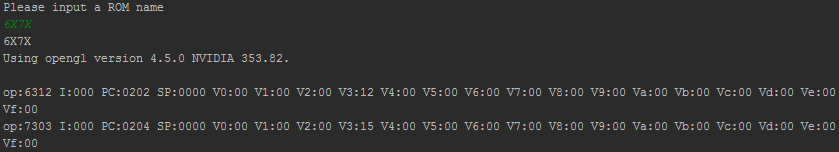
For å teste alle operasjonskodene våre utledet vi individuelle tester for hver eneste en og til slutt laget tester som testet dem i grupper. For å gjøre dette lagde vi test ROMs i en heksadesimal-editor kalt HxD. Et slikt redigeringsprogram lar oss skrive inn heksadesimal input som vi da kan lagre som en ROM fil og laste inn i ROM loaderen vår. Når loaderen kjører filen så leser den gjennom den heksadesimale inputen som programmet da tolker som forskjellige opkoder den skal gjøre noe med.   
  


*(Figur 4.1) HxD Hovedmeny. HxD er et gratisprogram utviklet av [MH-Nexus](https://mh-nexus.de/en)*

Vi kan ta hex-verdiene fra figur 4.1 og kjøre dem gjennom ROM loaderen. Det første vi vil vite er jo hvilke koder som kjører og hva de gjør. Operasjonskoder er identifisert i grupper av 4. Dette vil si at ”63 12” er en operasjon, og ”73 03” en annen. For å finne ut hvilken kode som blir kjørt og hva den gjør så tar vi en titt på figuren under.

  
*(Figur 4.2) En liten del av operasjonskode tabellen fra [Chip8](https://en.wikipedia.org/wiki/CHIP-8) sin Wikipedia side.*

Det første sifferet identifiserer hvilken operasjonskode som skal kjøres. I det første tilfellet ’6’ så sier det oss at det er opkode 6XNN som kjøres. For de 3 neste siffrene trenger vi bare følge beskrivelsen til opkoden i tabellen. I dette tilfellet blir desimalverdien 18 (12 er den heksadesimale verdien, som da gir oss 18 desimalt) lagt inn i register nummer 3. Etter denne operasjonen er utført blir opkode 7XNN utført. Den legger verdien 3 til register V3. Dette betyr at det nå burde ligge en total av 21 på register V3. Det sjekker vi i konsollen:

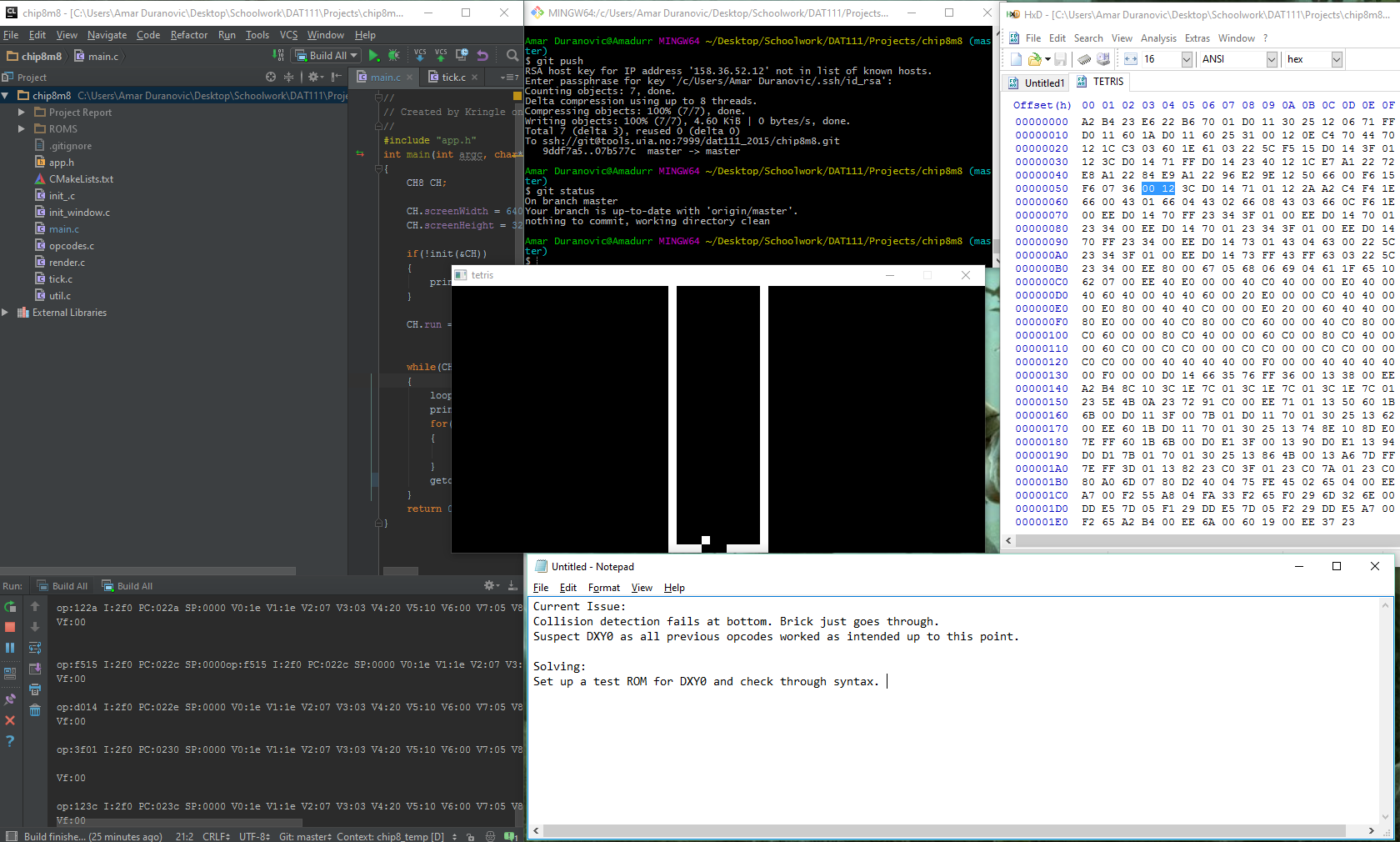


*(Figur 4.3) Konsollen til programmet.*

Ved første operasjonskode ser vi at den kjører 6312 som planlagt og setter 12 (Heksadesimalt) inn i V3. Da vet vi at den fungerer som den skal i dette tilfellet. Etterpå ser vi at op:7303 kjøres og den legger til 3 i V3, som gir en total av 15 (21 desimalt). Slik ser vi at opkode 6XNN og 7XNN fungerer som de skal i dette tilfellet.

## **Feilsøking**

Når man utvikler noe så er man så og si garantert å komme bort i feil. I figur 4.4 ser vi en ganske morsom feil vi fikk når vi prøvde å kjøre tetris. Den tegna opp banen og slapp brikken uten problemer, men når brikken kom ned til bunnen gikk den bare gjennom banen og stoppet programmet. Etter en god latter satte vi oss ned og analyserte problemet. Vi steppet programmet frem til hvor feilen skjedde, fant ut hvilke operasjonskoder som ble brukt og sjekket dem individuelt. Vi lærte at når feil oppstår, er det viktig og sette seg ned å få et perspektiv over problemet. Når man har en ide om hva som kan være feil setter man i gang med prosessen som er nødvendig for å fikse disse feilene.

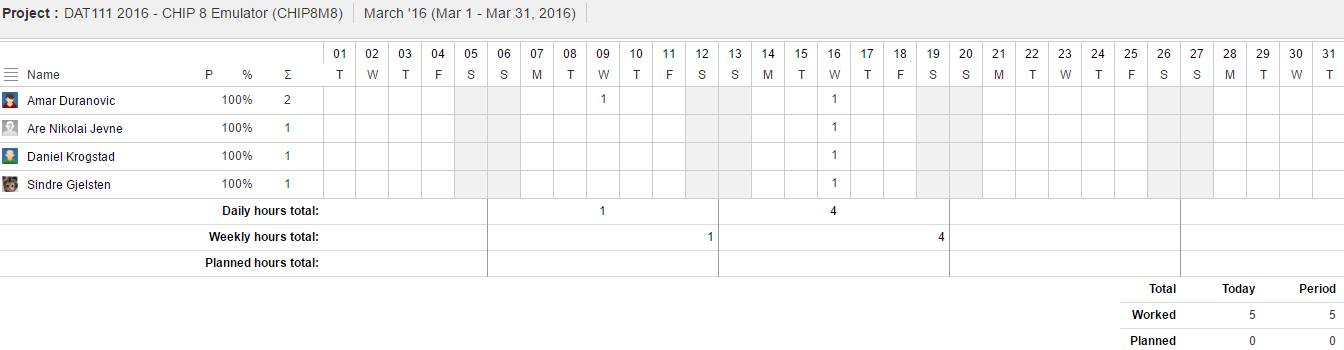
*******(Figur 4.4) Jakten på bugs.*

Nedenfor er en tabell for alle testene vi utførte på de forskjellige operasjonskodene.

|  |  |  |
| --- | --- | --- |
| Operasjonskode | Funksjon | Test |
| 00E0 | Renser skjermen | Tegner tilfeldige prikker på skjermen, kaller 00E0, ser om skjermen er klarert (Blank) |
| 00EE | Returnerer fra en sub-rutine | Kaller sub-rutinen, kaller 00EE og sjekker data. Kaller 00EE igjen og sjekker for underflow. |
| 1NNN | Hopper til adressen NNN | Gjør flere hopp mellom adresser, itererer V0 for hvert hopp, sjekker om V0 er rett på slutten. |
| 2NNN | Kaller sub-rutinen på NNN | Kaller sub-rutinen, sjekker om alt er i orden, kaller inn 16 til og sjekker for overflow. |
| 3XNN | Hopper over neste instruksjon om VX er lik NN | Skriver NN (f.eks 10) til et register (f.eks V5) og kaller 3XNN (3510). Om de ikke er like, skriv 0xFF til et register og kjør 3XNN+1 (3511). Om de er like, skriv FF til et annet register. Kjør til slutt 3XNN-1 (350F). Om like skriv 0xFF til et annet register, så utfør et register dump og sjekk hvor den mislyktes. |
| 4XNN | Hopper over neste instruksjon om VX IKKE er lik NN | Veldig likt som i 3NNN. Snu på lik/ikke lik lokasjonene. |
| 5XY0 | Hopper over neste instruksjon om VX er lik VY | Skriv NN-1 til et register, NN til neste, NN+1 på neste og NN til det siste. Sammenlign de tre første med den siste, skriv 0xFF til registere som mislykkes. |
| 6XNN | Setter VX til NN | Skriver NN til alle registere, sjekker data så overskriver et register flere ganger, sjekker alltid etter feil. |
| 7XNN | Setter NN til VX | Laster inn 1 til et register, adderer 1 til det registeret, fortsetter med dette og holder tellingen. Sjekker om første register = 0x00, om ikke, loop tilbake til den første addisjonen, sjekk data hvis lik. |
| 8XY0 | Setter VX til verdien av VY | Mye likt som 6XNN, men skriver til alle registrene fra et, så til et register fra alle. |
| 8XY1 | Setter VX til VX ELLER VY (Logisk ELLER) | Skriv 55 til et register og 0xAA til et annet. Kjør 8XY2, sjekk om vi får 0xFF fra VX men at VY enda er 0xAA. Gjenta for 0x5A, 0xA5 og til slutt 0x5A og 0x5A. |
| 8XY2 | Setter VX til VX OG VY (Logisk OG) | Likt som 8XY1, men med AND logikk. |
| 8XY3 | Setter VX til VX XELLER VY (Logisk XELLER) | Likt som 8XY1, men med XOR logikk. |
| 8XY4 | Adderer VY til VX. VF setter til 1 om det er et carry og til 0 hvis ikke. | Skriv N1 til et register, og N2 til et annet. Kjør 8YX4, Sjekk om X = N1 + N2, hvis usann hopp til enden av programmet og dump registrene. Hvis sann kjør 8XY4, og sjekk om Y = N2+(N1+N2). (Vær sikker på at tallene ikke skaper overflow). Skriv FF til register Z og kjør 8YZ4 og sjekk om Y = (N2+(N1+N2))-1(Pga binærlogikk) OG at VG = 0x01. |
| 8XY5 | VY substraheres fra VX. VF settes til 0 om det er et mente og 1 om det ikke er. | Skriv N1 til et register så N2 til et annet. Kjør 8XY45, sjekk om X = N1-N2, hvis usant hopp til slutten av programmet og dump registrene. Hvis sant kjør 8YX5 og sjekk om Y = N2-(N1-N2)(Vær sikker på å velge tall som ikke skaper underflow) Skriv 0xFF til register Z og kjør 8YZ5 og sjekk om Y = (N2-(N1-N2))+1(På grunn av binærlogikk) OG at VF = 0x00. |
| 8XY6 | Bitskifter VX til høyre med en. VF settes til verdien av det minst signifikante bitet av VX før skiftet. | Skriv 0x80 til et register, så bruk 8XY6 på det registeret og legg til 0x01 til et annet. Hvis VF != 1 loop tilbake til skiftet, hvis VF=1 sjekk data. |
| 8XY7 | Setter VX til VY minus VX. VF settes til 0 om det er et mente og 1 om det ikke er mente. | Likt som testen for 8XY5, bortsett fra at man fortsetter med det andre registeret. |
| 8XYE | Bitskifter VX til venstre med en. VF settes til verdien av det mest signifikante bitet av VX før skiftet. | Likt som testen for 8XY6, men skriv først 01 til registeret. |
| 9XY0 | Hopper over den neste instruksjonen om VX ikke er likt VY. | Likt som 5XY0 men skift ut de like med ulike og motsatt. |
| ANNN | Setter I til adressen NNN | Setter ANNN til 0x000, sjekker data, så med FFF, så 0xA5A og så 0x5A5. |
| BNNN | Hopper til adressen NNN pluss V0 | Hopp rundt et par ganger mens et register blir iterert. Sjekk data på slutten, hvis registeret ikke har rett verdi så har det gjort et feilhopp. |
| CXNN | Setter VX til resultatet av en bitvis-OG operasjon mellom et tilfeldig tall og NN | Kall CXNN et par ganger og skriv til neste register hver gang. Sjekk data og kjør hele prosessen et par ganger for å se om mønsteret gjentar seg. |
| DXYN | Sprites lagret i minne på lokasjon i indeks registeret (I), 8 bit brede. Pakker inn rundt skjermen. Hvis den tegner, renser en piksel, register VF setttes til 1 ellers er den 0. All tegning er XELLER tegning (m.a.o den veksler mellom skjerm pikslene). Sprites blir tegner i posisjon VX, VY. N er antall 8 bit rekker som må tegnes. Hvis N er større en 1, fortserrer andre linjen på posisjon VX, VY+1, og så videre | Skriv sprites til tilfeldige steder en stund, clear skjermen og tegn i en loop langs bunnen for å teste for overflows. |
| EX9E | Hopper over neste instruksjon hvis knappen som er lagret i VX er trykket ned | Tildel knapp 0 til VX så kjør EX9E i en loop som avslutter når knappen blir trykket. Iterer VX og gjenta til VX >= 0x0F |
| EXA1 | Hopper over neste instruksjon hvis knappen som er lagret i VX ikke er trykked ned | Samme som EX9E (Logikk til å hoppe over (Gitt EXA1 er på adressen 0x200)EXA1 1206 1200) |
| FX07 | Setter VX til verdien av delay timeren. (DT) | Sett DT så kjør FX07 og se i deconstructoren om VX == DT. Gjenta et par ganger. |
| FX0A | Et knappetrykk ventes og blir så lagret i VX. | Printer den trykte knappen på skjermen, venter på hvilken som helst knapp så renser skjermen og gjentar. |
| FX15 | Setter delay timeren (DT) til VX | Skriv 0x3C(Hexkode for 60, DT kjører på 60hz) til et register og kopier det til DT. Skriv DT til et register og sjekk om det er 0. Gjenta til det er 0 så avslutt. Sammenlign dette med en stoppeklokke eller timer manuelt for å se om den bruker cirka et sekund. |
| FX18 | Setter sound timeren (ST) til VX | Mye likt som FX15 men uten en sjekk loop siden den vil stoppe av seg selv. Bare sammenlign det med en stoppeklokke for å se hvor lenge den går. |
| FX1E | Adderer VX til I | Skriv noe satt data til et par registere og sett dem til I. Skriv 0xFF til V0 så hopp. Hvis den hopper til det rette stedet setter den V0 til 0x00 og hopper til slutten. Hvis ikke så hopper den bare til slutten eller kræsjer. |
| FX29 | Setter I til lokasjonen av spriten for karakteren i VX. Karakterer 0-F (Heksadesimalt) er representert av en 4x5 font | Kunne vært samme som FX0a, men anderledes feil kunne maskert hverandre. Så test denne på egenhånd SÅ test FX0A. Skriv tall fra 0x00 til 0x0F til de samsvarende registrene, så kjøre hver eneste med FX29 og printe de på skjermen på siden av hverandre. |
| FX33 | Lagrer den binærkodete desimal representasjonen av VX med de mest signifikante av de tre siffrene i adressen I. Det midterste sifferet på I +1, minst signifikante sifferes på I + 2. (m.a.o, tar desimal representasjonen av VX, plasserer hundredels sifferet i minnelokasjonen i I, tierene i lokasjonen I+1 og enerene i lokasjon I+2) | Skriv et random tall til VC og skriv tallene av det minnet med begynnelsen i 0x300. Iterer I med 3 og gjenta, last så opp 8 registere fra minne med begynnelse i 0x300. |
| FX55 | Lagrer V0 til VX (Inkluderer VX) i minne som begynner med adresse I | Skriv indeksene av hvert register til dem selv og skriv dette til minnet. Last registrene med tilfeldig data og last så tilbake til registrene fra minnet. |
| FX65 | Fyller V0 til VX (Inkluderende VX) med verdier fra minne som begynner med adresse I | Star en test med et hopp, så 00, 01, 02... 0E 0F og sett I til 202 og kjør FF65. |

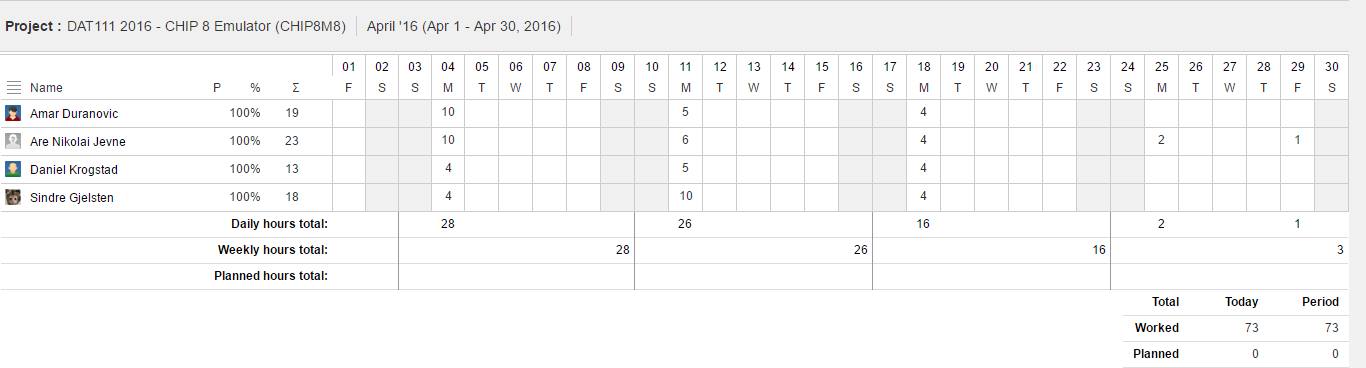
# **Timeliste**

Mars



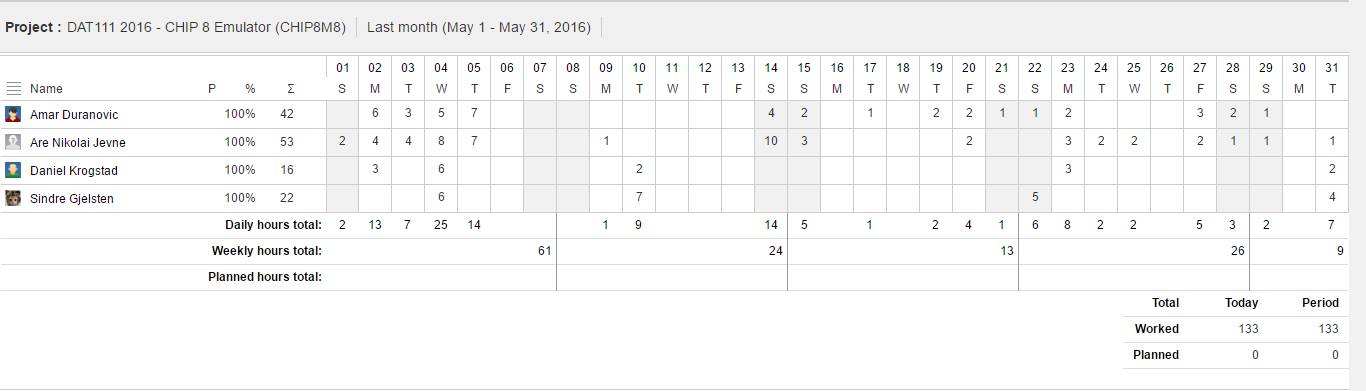
*(Figur 5.1) Timeliste Mars*

April



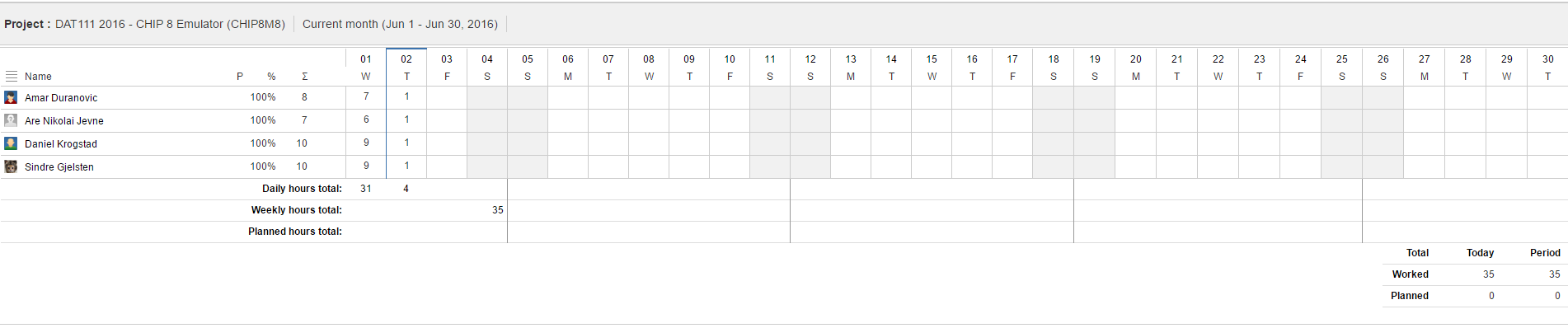
*(Figur 5.2) Timeliste April*

Mai



*(Figur 5.3) Timeliste Mai*

Juni



*(Figur 5.4) Timeliste Juni*

# **Drøfting**

Både individuelt og som en gruppe fikk vi mye innsikt i hvordan et større prosjekt skal håndteres. Alle i gruppen var uerfarne med mange av konseptene som ble grunnlaget for prosjektarbeidet vårt. Eksempler av dette var introduksjonen av verktøyet JIRA, som ble brukt i henhold til prosjektplanlegging. Vi var også relativt uerfarne på hvordan vi kunne bruke GIT effektivt i et team-miljø, så dette skapte også nye erfaringer. På grunn av alle de nye konseptene, pluss en prosjektoppgave vi ikke hadde noe forhold til i utgangspunktet, så tok det lang tid før vi kom skikkelig i gang. Til tross for alt som trakk oss tilbake, tok vi oss opp igjen og fikk en fundemental forståelse for hvordan emulatoren fungerte. Etter den forståelsen var i boks så komplimenterte tiden vi hadde brukt på JIRA + GIT oss slik at vi kunne bruke den resterende tiden mer effektivt.

## **Hva gikk galt?**

Prosjektet vårt fikk ikke oppmerksomheten det fortjente de første 6 ukene vi jobbet med det. Vi var dårlige på og sette korttidsmål i tillegg til at vi brukte for mye tid på andre prosjekter som var mer kritiske med henhold til innleveringsfrist. Målet vårt var å bli ferdige med ca. 90% av prosjektet før eksamensperioden. Grunnet nedprioritering og mye tid brukt på research av chip 8 og verktøyene våre, så ble det nærmere 25% ferdig ved starten av eksamensperioden.

Heldigvis viste det seg i eksamensperioden at tiden vi hadde brukt til research ikke var forgjeves. Når vi først satte oss ned for å jobbe effektivt med prosjektet så fikk vi gjort en utrolig stor mengde med arbeid over relativt kort tid. Selve emulatoren fikk vi kodet ferdig på litt over en uke, og da kjørte den spill heilt feilfritt.

## **Gruppen**

Til dette prosjektet ble gruppene valgt fritt og vi bestemte oss for å arbeide sammen allerede før jul. Samarbeidet innad i gruppen fungerte veldig bra og alle var åpne for forslag fra hverandre. Vi tilpasset oss til hverandres behov og alle kom konstant med tilbakemelding til prosjektet og arbeidet som ble gjort. Som nevnt tidligere begynte prosjektet tregt fordi ingen i gruppen hadde erfaring med de nye konseptene fra før. Vi delte gruppen opp slik at noen hadde ansvar for hoved-funksjonaliteten av emulatoren mens andre begynte på å lære seg de nye verktøyene tilgjengelige for oss.

## **Forandringer til neste prosjekt**

Det finnes flere ting som vi burde ta med oss videre til fremtidige prosjekter. Den første er å få en god oversikt over hvor mye jobb prosjektet inneholder. Når man ikke har en ide om hvor lang tid prosjektet kommer til å ta er det vanskelig og sette opp timelister og slik i starten. Vi har også innsett hvor viktig det er å få en bedre arbeidsfordeling fra starten av. Når man jobber ca. 4 timer i begynnelsen av prosjektet og eskalerer det til 10-14 timer så får man ikke så mye gjort i starten, og da sliter man med at stress spiller inn som en faktor på kvaliteten av produktet. Om man har en jevn arbeidsfordeling fra begynnelsen av kan man jobbe konsistent med produktet og ha en god ide om hvor mye som gjenstår hver uke. Noe som kan hjelpe til med å forbedre arbeidsfordelingen er og bli flinkere til å sette opp korttidsmål til neste gang.

# **Konklusjon**

Etter prosjektet sitter vi igjen med masse ny kunnskap. Vi har fått innsikt i hvordan emulering fungerer, samt også CPU. CPU fungerer på samme måte som en Chip-8, bare med mange flere operasjonskoder. Emulering i dag er en nostalgisk ting, som mange entusiaster verdsetter. Det at en kan lage nye datamaskiner til å fungere som de eldre og kjøre programmene til de. Git har også vært en stor del av prosjektet som gir et godt grunnlag for fremtiden, siden mange næringsorienterte prosjekter bruker Git.

# **Ordliste**

* Heksdesimal – 16 talls system, ofte brukt i dataprogrammering
* Operasjonskode – Instruksjons stavelse som angir operasjonen som skal utføres
* Sprite – en måte å vise 2d/3d bilder på data
* ROM – (read only memory) – en programfil man laster inn i minne
* Bool – Er representasjon av 2 verdier som enten er sann eller usann
* Dekoder – Tolker et data signal
* Struct – En gruppert liste med variabler
* Teller – komponent som laster inn ett tall og trekker fra 1 hver gang den får ett signal helt til den når 0

# **Kilder**

* Chip-8 Wikipedia (2016) <https://en.wikipedia.org/wiki/CHIP-8> [Lesedato: 19.05.2016]
* SDL2 Wikipedia (2016) <https://wiki.libsdl.org/APIByCategory> [Lesedato: 02.05.2016]
* Atkins, Jonathan (2009): SDL\_mixer 1.2.10 <http://tinyurl.com/h3vmpdk>[Lesedato: 27.05.2016]
* C Standard Library Reference Tutorial <http://tinyurl.com/jut8vvf> [Lesedato: 14.03.2016]
* OpenGL SDK <https://www.opengl.org/sdk/docs/man/> [Lesedato: 02.05.2016]
* Auby, Christian (2015): Introduction <http://tinyurl.com/zl94zaj> [Lesedato: 19.05.2016]
* Muller, Laurence (2011): How to write an emulator (CHIP-8 interpreter) <http://tinyurl.com/mq3ow8p>[Lesedato: 14.03.2016]
* Fegth, Gro J.S (2008): Bruk av kilder <http://tinyurl.com/35htaq6> [Lesedato: 31.05.2016]
* Greene, Thomas P (1997): Chip-8 <http://tinyurl.com/b7z2t7y> [Lesedato: 14.03.2016]
* Mikolay, Matthew (xxxx) Mastering Chip-8 <http://tinyurl.com/jzfnfx9> [Lesedato: 14.03.2016]
* RFF (2012): Sample flowcharts and templates <http://tinyurl.com/7m2d4d9> [Lesedato: 20.05.2016]
* Ambler, Scott W (2014): UML 2 sequence diagrams: an agile introduction <http://tinyurl.com/2u89e7> [Lesedato: 20.05.2016]
* Angelescu, Radu (2015): Keymapping <http://tinyurl.com/hzx4vys> [Lesedato: 21.05.2016]

**Figurliste**

* Figur 1-1: Utsikt over knappene vi bruker på tastaturet i forhold til hva de er på Chip-8 sitt Hex Tastatur.
* Figur 1.2: En liste over de 15 første operasjonskodene i Chip 8.
* Figur 1.3: Sekvensdiagram for operasjonskoden 8XY1.
* Figur 4.1: HxD Hovedmeny. HxD er et gratisprogram utviklet av [MH-Nexus](https://mh-nexus.de/en).
* Figur 4.2: En liten del av operasjonskode tabellen fra [Chip8](https://en.wikipedia.org/wiki/CHIP-8) sin Wikipedia side.
* Figur 4.3: Konsollen til programmet.
* Figur 4.4: Jakten på bugs
* Figur 5.1: Timeliste Mars
* Figur 5.2: Timeliste April
* Figur 5.3 Timeliste Mai
* Figur 5.4: Timeliste Juni