StackPointer

2025-1. Organ för Datorföreningen Stacken, KTH.

Datorföreningen Stacken

Stacken är en kårförening inom Tekniska Högskolans Studentkår. Föreningens syfte är att främja intresset för och vidga kunskaper inom datorområdet, och det har Stacken gjort sedan 1978.

Varje torsdag träffas medlemmar i vår lokal i Q-huset där vi har ett trevligt utrymme att träffas samt en datorhall. Under de senaste cirka 20 åren har drift av olika tjänster så som chat, DNS, mejl, NextCloud, och lite annat varit den större delen av Stackens aktiva projekt, men på senaste år har olika programmeringsprojekt också börjat dyka upp då och då (ett av dessa är fokuset i detta nummer av StackPointer).

Att bli medlem i Stacken kan vara ett bra sätt att få kontakt med andra som delar intressen, alternativt lära sig nya saker och få hjälp. Medlemskap är gratis för studenter, och kostar 125 kronor för icke studerande. Medlemsansökan görs på https://stacken.kth.se/.

Mejl: `styrelsen@stacken.kth.se`

StackPointer

StackPointer är organ (definition 2) för Datorföreningen Stacken på KTH. Det senaste nummret gavs ut år 2004, men intresse har nyligen dykt upp för att återuppliva StackPointer, framförallt inför Kårens dag 2025. Nya StackPointer är drivet av relativt nya medlemmar och har nästan ingen kontinuitet med gamla StackPointer, förutom att redaktören hittade de LaTeX filer som användes för 21 år sedan, och att äldre medlemmar naturligtvis får skriva artiklar. Framtida planen för StackPointer är att förhoppningsvis publicera minst ett nummer om året, och förhoppningsvis ska alla innehålla något intressant att läsa.

Mejl: `stackpointer@stacken.kth.se`
Redaktör: John Lorentzson (Duuqnd)
Färdigställd: 2025-08-06

Innehåll

Styrelsen

Förutom en medlem är styrelsen helt ny, och tre femtedelar studenter.

Ordförande Ada Björnebro
Sysselsättning: Studerar datateknik

Viceordförande Hugo Ameln
Sysselsättning: Studerar teknisk fysik

Kassör Stellan Lagerström
Sysselsättning: Jobbar som mjukvarukonsult med fokus på inbyggda system

Sekreterare John Lorentzson
Sysselsättning: Deltidsarbetslös

Ledamot Viktor Kryg
Sysselsättning: Studerar teknisk fysik

Stackens demo för Kårens dag 2025

av John Lorentzson

Varje år har Tekniska Högskolans Studentkår ett event där kårföreningar och andra relaterade organisationer får introducera sig till nya studenter. Självklart deltar Stacken, och förra året hade vi med oss en demo som visade olika effekter på en DEC VT220 terminal.

För ett tag sedan togs den tidigare dekorativa Commodore 64 datorn ut ur displaymontern där den länge varit instängd. Vi skaffade ett nytt nätagregat till den och senare en disk drive emulator. För Kårens dag 2025 har den blivit fokuset av vår demo: livecoding! Livecoding är när man modifierar ett program medans det kör, och att programmet uppdateras samtidigt.

Vi använder samma VT220 som förra året till att visa och redigera kod skrivet i ett simpelt programmeringsspråk. Tryck på en knapp så kompileras koden och skickas till C64:an, som sedan exekverar koden och ritar grafik på skärmen.

Koden för programmet i bilden:

```
if first() then
    clearScreen()
end

for y do 200 times
    x = time()
    if xor(x, y) <= x then
        putPixel(x, 200 - y)
    end
end
```

Keep It Short & Simple

av Rasmus Kaj

> Measuring programming progress by lines of code is like measuring aircraft building progress by weight.
>
> – Bill Gates (brukar det påstås. Någon annan sa det förmodligen först)

De flesta erfarna programmerare känner igen uppmaningen *"KISS"*, och vet att den uttyds *"Keep It Simple, Stupid"*. Översatt till svenska blir det "Håll det enkelt, dummer", eller kanske möjligen lite snälltolkat "Håll det enkelt (och) dumt".

Jag tycker förkortningen borde uttydas *"Keep It Short (and) Simple"*, eller "Håll det kort och enkelt".

Att folk i allmänhet känner till – och många till och med följer – uppmaningen är bra. Och det beror nog delvis på att det är lite "tufft" att kalla programkod, och de som skrev den, för dum. Men jag har två invändningar.

Den första är att världen i allmänhet, och kanske programmeringsvärlden i synnerhet, behöver mera gemenskap, förståelse, och uppmuntran och mindre av "det är coolt att vara elak".

Den andra invändningen ligger i själva orsaken till att uppmaningen *"KISS"* behövs över huvud taget. Programkod ska inte bara kunna köras och göra rätt saker, den ska också kunna underhållas, justeras, och rättas. Om någon ska kunna underhålla koden behöver någon förstå koden. Denna någon är kanske en ny utvecklare i ditt team eller någon som tar över när du har slutat. Eller du själv några månader eller år senare.

När det gäller att förstå koden i ett program finns det ett hinder som vida överstiger alla andra. Man kan ha åsikter om `goto`, om globalt muterbart state eller om överdriven användning eller avsaknad av `map`/`reduce`, asynkrona primitiver, och så vidare. Allt sådant är sekundärt. Det vanligaste hindret för att förstå koden i ett program är att den är för lång.

Så försök alltid skriva din kod så kort och läsbar som möjligt. Om en "avancerad" feature i programspråket du använder kan göra programmet några rader kortare är det troligen värt att använda den. Det är rimligt att förutsätta att den som läser ditt program i framtiden kan språket det är skrivet i, men du kan inte kräva att den som läser programmet redan vet vad det försöker åstadkomma och hur.

Det betyder naturligtvis inte att alla variabelnamn ska vara en bokstav korta och indentering och annan "frivillig" whitespace ska tas bort. Använd rimliga namn, och formatera koden enligt allmänt accepterade konventioner för språket (många moderna språk har såna konventioner formaliserade i ett formateringsverktyg. Använde det! Och använd det med så få egna inställningar som möjligt).

Att skriva rimligt kort behövs på alla nivåer. En rad bör inte vara för lång, en funktion bör inte ha för många rader, en fil bör inte innehålla för många funktioner, en katalog bör inte innehålla för många filer och hela programmets katalogstruktur bör inte vara för djup. Bryter man ut väldigt korta funktioner får man i stället många funktioner (och extra jobb att hitta på vettiga namn för dem, och kanske även skriva dokumentation för dem). Det gäller att hitta rätt balans.

Ett effektivt sätt att göra koden kortare är att använda bibliotek som redan finns. Grundläggande datastrukturer och algoritmer finns ofta i standardbiblioteket (eller i externa bibliotek om du skriver i javascript), mer specifika saker, som parsers för olika filsystem, klienter för nätverksprotokoll, etc finns i externa bibliotek (eller i standardbiblioteket om du skriver i python). Men se upp – listan över externa beroenden är också en sak som bör hållas så kort som möjligt. Det gäller att hitta rätt balans även där.

Kompilator del 1: mittenstadiet

av John Lorentzson

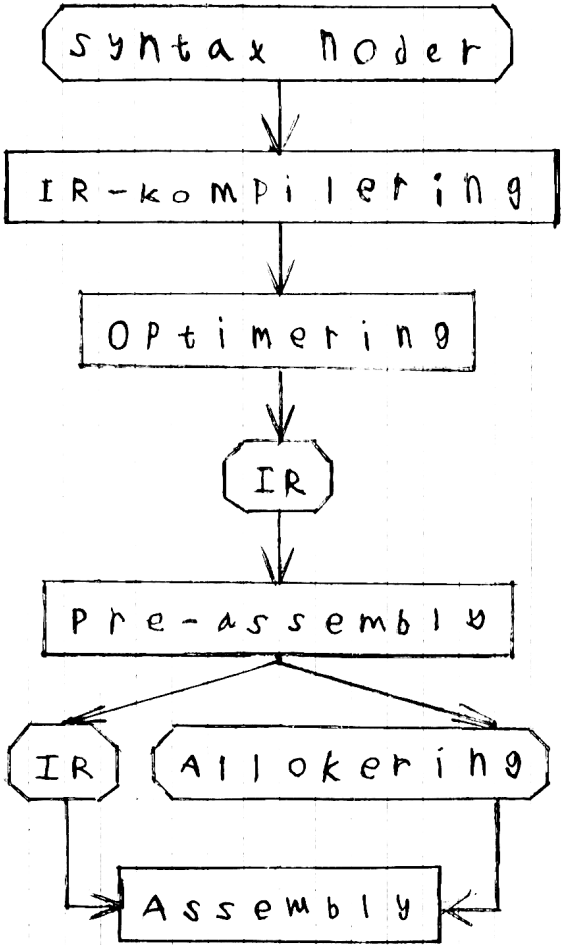
Det är inte allt för svårt att skriva en kompilator. Dessa två kompilatorartiklar kommer visa hur kompilatorn som drev vår C64-livecoding demo fungerade, och förhoppningsvis kommer den verka rimligt simpel.

Vad denna serie inte handlar om

- **Lexing eller parsing.** Jag tycker personligen att syntax är bland de tråkigaste sakerna om ett programmeringsspråk och skulle helst ha så lite som möjligt. Språket jag presenterar här har dock något involverad syntax, så en parser behövdes, men den är väldigt simpel och faller utanför seriens fokusområde. För en guide om att skriva parsers, se kapitel 4, 6, och små delar av 8 och 9 från boken Crafting Interpreters, som finns att läsa gratis på nätet.
- **Typsystem.** Språket jag har implementerat har endast en typ: positiva 8-bit heltal. Därför fanns inget behov av typsystem, varken statiskt eller dynamiskt, och det kommer därför inte diskuteras.
- **Dynamisk minnesallokering.** Som sagt finns det endast 8-bit heltal, och det betyder även att inga sorts vektorer, par, eller andra strukturer som kan skapa varierande stora objekt finns. Därför har vi ingen möjlighet att utföra dynamisk allokering, därför behöver vi inte tänka på ämnet.

Vad denna serie handler om

Denna serie kommer gå igenom processen från ett färdigt syntaxträd till maskinkod, och det som kommer mellan. Högnivåstrukturen av den relevanta delen av kompilatorn finns det ett diagram av på nästa sida.



Sammanfattning av syntaxträd

Syntax är utanför vårt fokusområde, men det är viktigt att veta vad vi arbetar med och vad som hände innan vi kom hit. Parsern genererar ett syntaxträd som är byggt av noder. Syntaxträdet matchar kodens text ganska löst, men täcker ändå allt som kan uttryckas (kommentarer ignoreras totalt). Vi har följande syntaxnoder:

- **NODE-BLOCK**: Innehåller en sekvens av andra noder.
- **NODE-PROGRAM**: Samma som ett block, fast fungerar som en wrapper runt hela programmet. Lite användbart senare.
- **NODE-EXPR**: Basklass för uttryck, i detta fall är uttryck saker som bildas av användningen av operatorer. Innehåller en lista med operander (sakerna som operatorn arbetar på).
- **NODE-EXPR-GROUPING**: Skapas av uttryck som omringats av parenteser. Innehåller det uttrycket som var omringat.
- **NODE-STANDARD-EXPR**: Basklass för alla vanliga binära operatorer (operatorer som placeras mellan dess två operander, dvs. infix notation).
- **NODE-EXPR-...**: Subtyper av `NODE-STANDARD-EXPR`. Alla vanliga binära operatorer så som addition, subtraktion, likhetstest, etc., får en egen klass vars namn börjar med `NODE-EXPR-`. Dessa innehåller inget annorlunda från andra uttryck.
- **NODE-ASSIGNMENT**: En sättning av en variabel till värdet av ett uttryck. Innehåller en destinationsvariabel och ett uttryck som ska bli till variabelns värde. Är själv ett uttryck, eftersom assignement ska returnera värdet som sattes.
- **NODE-CALL**: Ett funktionsanrop till en inbyggd funktion. Innehåller en funktion att anropa och en lista med argumenten.
- **NODE-CONDITIONAL**: En IF-sats. Innehåller ett uttryck (testet) och ett eller två block, där det nödvändiga blocket är THEN-fallet (när testet evalueras till sant), och det frivilliga blocket är ELSE-fallet (när testet blir falskt).
- **NODE-DOTIMES**: En loop, som innehåller en variabel eller konstant som bestämmer hur många gånger loopen ska köra, en variabel som används som räknare, och ett block med kod som ska upprepas.

Det blir ganska få, men det är ju ett väldigt simpelt språk. På den här nivån har variabel namn och konstanter transformerats till "referenser". En konstantreferens innehåller sitt värde, och en variabelreferens innehåller ett namn. I vissa fall kan vi hantera dessa som uttryck. Dessa transformeras ännu längre senare.

Observera att trots att `NODE-DOTIMES` accepterar en variabel som gränsvärde är det variabelns värde vid början av loopen som används. Att ändra den under loopens gång har ingen effekt på loopens längd. Detta är till för att göra språket linjärt och icke-turingkomplett, så att vi kan undvika att användarens kod kan fastna i en oändlig loop.

Basic block / `IBLOCK`

När man utför optimering och generell bearbetning av kod är det användbart att kunna tydligt följa flödet av både kod och data. Det hjälper då att begränsa hur man uttrycker koden, och jag använder en representation som heter "basic block". Ett basic block är en rak linjär kodsekvens där branching bara är tillåtet *till början* av ett block och *från slutet* av ett block. I praktiken betyder det att alla instruktioner som fungerar som ett "jump"

av något slag ger ett basic block som sin destination, och att dessa "jump" instruktioner bara får finnas som den sista instruktionen i ett basic block. I det här fallet kommer vi ha påhittade instruktioner som endast finns för kompilering och optimering, en så kallad "intermediate representation", härefter förkortat till IR.

I just denna kompilator heter klassen som implementerar basic block "`IBLOCK`", och har fält för ett namn, den första och sista instruktionen, samt nästa och föregående `IBLOCK`.

Instruktioner

`IBLOCK` innehåller "IR-instruktioner", vars klass heter `IR-INST`. Dessa är hoplänkade i en dubbellänkad lista genom `next` och `previous`-fält i instruktionerna. IR-instruktioner har även fält för en lista med inputs, en output, och vilket `IBLOCK` som instruktionen finns i. Det finns en subtyp av `IR-INST` som heter `IR-TERMINATOR`, vilket är en instruktion som avslutar ett `IBLOCK`. En `IR-TERMINATOR` har en tillagd lista med destinationer (`IBLOCK` som kodens flöde kan gå vidare till) och dess `next`-fält är alltid tomt. Om en IR-instruktion bara tar en input så kan vi prata om dess singulära `input` (som egentligen bara är det första objektet i `inputs`-listan).

Dessa är IR-instruktionerna som den här kompilatorn innehåller:

- **`IR-GETCONST`**: hämtar en konstant för användning.
- **`IR-FETCHVAR`**: hämtar värdet från en namngiven variabel för användning.
- **`IR-OPERATION`**: Basklass för vanliga binära operatorer.
- **`IR-COMPARISON`**: Subtyp av `IR-OPERATION`, basklass för tester.
- **De vanliga binära operatorerna**, så som tester och aritmetik. Subtyp av `IR-OPERATION`. Om operatorn är ett test är den även en subtyp av `IR-COMPARISON`.
- **`IR-CALL`**: anropar en inbyggd funktion, första input är funktionen som anropas, resten är argumenten.
- **`IR-ASSIGN`**: ger variabeln given i sin `output` det värde den får från sin `input`.
- **`IR-JUMP`**: en `IR-TERMINATOR` som designerar ett omedelbart jump till sin destination. Används bland annat som slutet på ett `IBLOCK` som inte utför någon branching.
- **`IR-IF`**: en `IR-TERMINATOR` som har två destinationer och en input. Om `input` blir är ett sant värde går flödet till första destinationen. Om falskt, till den andra.
- **`IR-RETURN`**: en `IR-TERMINATOR` som avslutar användarens kod.

Data, inputs och output

I kompilatorn behövs ett sätt att representera värden som kommer att finnas i det exekverade programmet. Dessa kan vara programmets variabler, konstanter, och temporära variabler som skapas när uttryck och funktionsanrop plattas ut till en platt exekverbar form (dvs. ingen nesting). Eftersom det här språket endast innehåller 8-bit heltal så är allt

vi behöver veta om icke-konstanter vilken instruktion som värdet skapas (eller definieras) av och vilken/vilka det används av. Konstanter måste naturligtvis kunna peka till sitt konstanta värde på något sätt.

Vi har dessa typer av data:

- **`IR-RESULT`**: Skapas som normal output från IR-instruktioner.
- **`IR-REUSABLE`**: Data som kan användas på flera olika ställen.
- **`IR-VARIABLE`**: En subtyp av `IR-REUSABLE`, representerar en *namngiven* variabel från programmet och har därmed ett namn. Varje `IR-VARIABLE` matchar med en variabelreferens från de övre lagren.
- **`IR-CONSTANT`**: En till subtyp av `IR-REUSABLE`, definieras av bl.a. `IR-GETCONST`.

Kompilatorn håller automatiskt koll på både vart en bit IR-data har definierats och vart den används och vi kan därmed följa flödet av data väldigt enkelt.

Konstruktion av IR

När vi kompilerar från syntaxnoder till IR-instruktioner bygger vi upp instruktionsgrafen rakt. Vi har alltid ett nuvarande `IBLOCK` som vi bygger, och vi kan interagera med det genom att antingen **sätta in** en instruktion efter den förra (om en förra finns), eller **avsluta det nuvarande `IBLOCK`** med en `IR-TERMINATOR` instruktion. När vi har avslutat ett `IBLOCK` kan vi antingen vara helt klara eller **påbörja ett nytt `IBLOCK`**.

Kompilering från syntaxnoder till IR-instruktioner

Vi går nu igenom hur varje syntaxnod kompileras till IR-instruktioner.

Men först, vissa syntaxnoder innehåller *referenser*, dvs. namngivna variabler eller konstanter. Dessa behöver också kunna kompileras. Som tur är så är det väldigt simpelt. Variabelreferenser kompileras till en `IR-FETCHVAR` och konstantreferenser kompileras till en `IR-GETCONST`. Namnen på dem kanske borde varit lite mer matchande, men det är nog lite sent för det nu…

`NODE-BLOCK` kompileras väldigt simpelt: kompilera alla noder som den innehåller i sin "statements" lista. Klart. `NODE-PROGRAM` kompileras på samma sätt som `NODE-BLOCK`, men avslutas med att sätta in en `IR-RETURN` efteråt. Dessa har ingen output.

`NODE-ASSIGNMENT` kompileras genom att först kompilera uttrycket i fältet `value` och använda den output vi fick från det som input till en `IR-ASSIGN`. Output är den `IR-VARIABLE` som matchar syntaxnodens variabelreferens. En assignment kan även användas som ett uttryck som ger ett värde, så vi sätter även in en `IR-FETCHVAR` instruktion efteråt och dess output returneras. Om den instruktionen inte behövs kommer den optimeras bort senare.

`NODE-EXPR-GROUPING` skickar bara vidare till sitt innehållna uttryck.

Kompilering av argument och instruktioner som tar dem
För att kunna gå igenom hur `NODE-STANDARD-EXPR` och `NODE-CALL` kompileras måste vi först veta hur vi ska kompilera dess argument. Detta är faktiskt ganska simpelt. Ta listan med argument från noden och kompilera alla objekt i listan. Om parsern har gjort sitt jobb ska alla dessa vara uttryck i formen av syntaxnoder eller referenser. Vi kompilerar våra argument och samlar in IR-data som ges som resultat. Detta sker rekursivt i våra argument om det behövs där också.

En subtyp av `NODE-STANDARD-EXPR` kompileras till den matchande `IR-OPERATION` instruktionen. Dess argument är kompilerade och används som inputs till instruktionen. En sådan instruktion har endast två inputs, så parsern måste se till att binära operatorer är uppdelade rätt. Som tur är så är det både enkelt och ligger utanför vårt fokusområde. Liknande sker för `NODE-CALL`, fast med arbiträrt många argument, och att dess första input är funktionen som ska anropas.

Branch
`NODE-CONDITIONAL` tar lite mer arbete. Först kompilerar den sin testnod och sparar dess resultat. Sedan skapar den antingen två eller tre nya `IBLOCK`. Eftersom ett `IBLOCK` måste vara linjärt kan vi inte fortsätta använda samma `IBLOCK` efter att vi introducerat branching, så en fortsättning behövs. Om IF-satsen inte har något eget ELSE-fall kan vi låta vårt fortsättningsblock vara ELSE-blocket, annars måste de vara separata, så i det fallet blir fortsättningsblocket ett eget block.

Vi avslutar det nuvarande blocket med en `IR-IF` som tar som input det resultat vi fick från testnoden, och har THEN-blocket och ELSE-blocket som sina destinationer. Vi påbörjar THEN-block och kompilerar THEN-fallets nod in i den, och sedan avslutar blocket med en `IR-JUMP` till fortsättningsblocket.

Om ett ELSE-fall finns startar vi ELSE-blocket och kompilerar det fallets nod. Precis som tidigare avslutar vi med ett `IR-JUMP` till fortsättningen. Sedan startar vi det nya blocket, fortsättningen. Om ett ELSE-fall inte finns så är fortsättningsblocket och ELSE-blocket samma block, och vi påbörjar det blocket nu.

Loop
`NODE-DOTIMES` har den längsta kompileringsmetoden av alla (och den minst städade). Först måste räknaren och stoppvärdet initialiseras. Räknaren initialiseras genom en `IR-GETCONST` som tar in noll och en `IR-ASSIGN` som sätter räknarvariabeln till den.

Stoppvärdet kan antingen vara en konstant eller det nuvarande värdet av en variabel och behöver bara initialiseras i det andra fallet. Vi behöver göra en temporär variabel

ur en namngiven variabel, men ingen sådan instruktion finns. Istället sätter vi variabeln genom en addition med noll, som senare kan optimeras.

Efter initialisering avslutas blocket. Ett nytt block påbörjas som senare kommer bli en branch destination, och loopens innehåll kompileras. Efteråt sätter vi in kod som adderar 1 till räknaren. Detta görs med samma IR-instruktioner som hade genererats av koden `counter = counter + 1`. Därefter jämförs den med stoppvärdet och resultatet av jämförelsen används som argument till en `IR-IF`. Ena destination är det blocket vi startade, den som innehåller loop koden, den andra är fortsättningen.

Vi avslutar loopens kompilering genom att påbörja fortsättningsblocket.

Grafmanipulation och optimering

Nu när koden är kompilerad till IR kan vi utföra optimeringar. Vi kan, tack vare strukturen av `IBLOCK` och instruktionerna, väldigt enkelt lägga till nya instruktioner samt flytta eller radera existerande instruktioner. Detta görs genom att helt enkelt peka om vart instruktionen och/eller dess grannar pekar. Vi måste också ta hänsyn till strukturen av `IBLOCK` när vi gör detta (till exempel ifall vår instruktion råkar vara den första i blocket), och vi måste även se till att vi inte låter data peka till instruktioner som inte längre finns.

De följande optimeringar används:

- Omordning av argument. Väldigt simpla argument så som variabler och konstanter är mycket simplare att ladda in, medans ett komplicerat uttryck kan kräva att ett helt annat funktionsanrop först utförs, och det kan göra att vi behöver skapa fler temporära variabler. Vi kan därför ta alla simpla argument (som vi definierar som allt förutom `IR-OPERATION` eller `IR-CALL`) och flytta dem till omedelbart ovanför instruktionen där de används.
- Deduplicering av argument. I vissa fall har vi flera av samma argument till ett anrop, till exempel innehåller anropet `line(0, 0, x, y)` två nollor. Den här optimeringen gör så att vi bara behöver en `IR-GETCONST` för det, och att vi återanvänder dess resultat. Detta blir mer komplicerat för namngivna variabler, så vi gör bara det här för konstanter.
- Borttagning av oanvända instruktioner. Om vi vet att en instruktion bara används för dess resultat, och vi ser att resultatet inte har några användare (lätt att se eftersom `IR-DATA` håller koll på det själv), så kan vi radera instruktionen.

Vidare optimering utförs i det sista “pre-assembly” steget.

Vad det lämnar oss med, och nästa steg

Vi kan nu transformera det parsern ger oss till något vi mycket lättare kan transformera och som representerar programmets beteende mer explicit, och vi kan gå vidare till stegen närmare hårdvaran. Ett litet exempel på hur IR från ett visst program ser ut finns här.

```
for x do 100 times
    if x < 50 then
        line(0, 0, x, x)
    else
        line(0, 0, x + 75, x)
    end

    line(0, 255 - x, 255, 255 - x)
end
```

⇓

toplevel:
IR-GETCONST (0)
⟶ <IR-CONSTANT 0 (ID #01)>
ASSIGN <IR-VARIABLE "x"> = <IR-CONSTANT 0 (ID #01)>
IR-JUMP to <IBLOCK *loop_1*>
loop_1:
IR-GETCONST (50)
⟶ <IR-CONSTANT 50 (ID #02)>
IR-FETCHVAR (<IR-VARIABLE "x">)
⟶ <IR-RESULT #03>
IR-TEST-LESS (<IR-RESULT #03>, <IR-CONSTANT 50 (ID #02)>)
⟶ <IR-RESULT #04>
IR-IF <IR-RESULT #04>, then: <IBLOCK *then_1*>
else: <IBLOCK *else_1*>
then_1:
IR-GETCONST (0)
⟶ <IR-CONSTANT 0 (ID #05)>
IR-FETCHVAR (<IR-VARIABLE "x">)
⟶ <IR-RESULT #06>
IR-FETCHVAR (<IR-VARIABLE "x">)
⟶ <IR-RESULT #07>
IR-CALL line(<IR-CONSTANT 0 (ID #05)>, <IR-CONSTANT 0 (ID #05)>, <IR-RESULT #06>,
<IR-RESULT #07>)
⟶ <IR-RESULT #08>
IR-JUMP to <IBLOCK *merge_1*>
else_1:
IR-GETCONST (0)
⟶ <IR-CONSTANT 0 (ID #09)>
IR-GETCONST (75)
⟶ <IR-CONSTANT 75 (ID #10)>

```
IR-FETCHVAR (<IR-VARIABLE "x">)
                                                                    ⟶ <IR-RESULT #11>
IR-PLUS (<IR-RESULT #11>, <IR-CONSTANT 75 (ID #10)>)
                                                                    ⟶ <IR-RESULT #12>
IR-FETCHVAR (<IR-VARIABLE "x">)
                                                                    ⟶ <IR-RESULT #13>
IR-CALL line(<IR-CONSTANT 0 (ID #09)>, <IR-CONSTANT 0 (ID #09)>, <IR-RESULT #12>,
             <IR-RESULT #13>)
                                                                    ⟶ <IR-RESULT #14>
IR-JUMP to <IBLOCK merge_1>
       merge_1:
IR-GETCONST (0)
                                                          ⟶ <IR-CONSTANT 0 (ID #15)>
IR-GETCONST (255)
                                                        ⟶ <IR-CONSTANT 255 (ID #16)>
IR-FETCHVAR (<IR-VARIABLE "x">)
                                                                    ⟶ <IR-RESULT #17>
IR-MINUS (<IR-CONSTANT 255 (ID #16)>, <IR-RESULT #17>)
                                                                    ⟶ <IR-RESULT #18>
IR-GETCONST (255)
                                                        ⟶ <IR-CONSTANT 255 (ID #19)>
IR-GETCONST (255)
                                                        ⟶ <IR-CONSTANT 255 (ID #20)>
IR-FETCHVAR (<IR-VARIABLE "x">)
                                                                    ⟶ <IR-RESULT #21>
IR-MINUS (<IR-CONSTANT 255 (ID #20)>, <IR-RESULT #21>)
                                                                    ⟶ <IR-RESULT #22>
IR-CALL line(<IR-CONSTANT 0 (ID #15)>, <IR-RESULT #18>, <IR-CONSTANT 255 (ID #19)>,
             <IR-RESULT #22>)
                                                                    ⟶ <IR-RESULT #23>
IR-GETCONST (1)
                                                          ⟶ <IR-CONSTANT 1 (ID #24)>
IR-FETCHVAR (<IR-VARIABLE "x">)
                                                                    ⟶ <IR-RESULT #25>
IR-PLUS (<IR-RESULT #25>, <IR-CONSTANT 1 (ID #24)>)
                                                                    ⟶ <IR-RESULT #26>
ASSIGN <IR-VARIABLE "x"> = <IR-RESULT #26>
IR-GETCONST (100)
                                                        ⟶ <IR-CONSTANT 100 (ID #27)>
IR-FETCHVAR (<IR-VARIABLE "x">)
                                                                    ⟶ <IR-RESULT #28>
IR-TEST-EQUAL (<IR-RESULT #28>, <IR-CONSTANT 100 (ID #27)>)
                                                                    ⟶ <IR-RESULT #29>
IR-IF <IR-RESULT #29>, then: <IBLOCK after_loop_1>
                       else: <IBLOCK loop_1>
       after_loop_1:
End of Program
```

En kort introduktion till 6502

av Hugo Ameln och John Lorentzson

Till Kårens dag 2025 har Stacken skapat en interaktiv demo som använder sig av en Commodore 64 och en VT220-terminal. I det här numret av StackPointer finns det ett par artiklar om hur det fungerade, så en introduktion till C64:ans processor, 6502*, är användbart. Notera att detta är förkortat och ej komplett.

Hur en simpel CPU fungerar

En typisk CPU fungerar genom att exekvera en sekvens av instruktioner från sitt minne. Dessa instruktioner består oftast av en "opcode", som väljer instruktionens beteende, och en viss mängd operander, som är data som instruktionen ska arbeta på. Olika format används av olika processorer för att lagra dessa, och 6502 har ett väldigt simpelt format: första byten är opcode, därefter kommer 0, 1, eller 2 bytes beroende på om en operand finns och om den är 1 eller 2 bytes lång.

Register

En normal CPU innehåller en viss mängd register, vilket är platser att lagra värden. 6502 har tre stycken:

- **`A`**: "Ackumulatorn", det generella registret.
- **`X`** och **`Y`**: "Indexregister", begränsad användning.

Alla register är 1 byte (8-bit) stora.

Statusflaggor

Efter att en instruktion har utförts kan den ibland sätta en "flagga" som indikerar olika saker om resultatet.

- **`Z` (zero)** sätts om resultatet var `0`, töms annars.
- **`N` (negative)** sätts om resultatet är negativt enligt tvåkomplementsform (dvs. om högsta biten är `1`).
- **`C` (carry)** sätts och läses av aritmetik instruktioner. Används t.ex. när man jobbar med 16-bitars aritmetik, se instruktionslistan nedan för mer information.

Statusflaggorna kan inte läsas direkt, men används istället av bl.a. branchinstruktioner.

*Commodore 64 innehåller egentligen en nästan totalt kompatibel variant av 6502:an som heter 6510.

Adresseringslägen

Precis som med registren arbetar 6502 med minnet en byte i taget. Det finns olika sätt för instruktioner att röra minnet, dessa heter adresseringslägen ("addressing modes"). Olika instruktioner accepterar olika adresseringslägen, men de som är relevanta för oss är följande:

- **Immediate**: Ett konstant värde som skrivs direkt i instruktionen. Syntax: `LDA #$01`.
- **Absolute**: Når värdet genom en komplett 16-bit (2 byte) minnesadress. Syntax: `LDA $D021`
- **Zeropage**: Likt absolute, men adressen är endast 1 byte lång. De första 256 bytes från `$0000` till `$00FF` i en 6502-processors minne kallas för "zeropage" och går att nå snabbare eftersom adressen är en byte kortare. Syntax: `LDA $BA`
- **Absolute och Zeropage indexed**: Når värdet genom adressen plus en offset ur en register, antingen `X` eller `Y`. Syntax: `LDA $1F, X` / `LDA $1234, Y`
- **Indirect indexed**: Hämtar först 2 bytes från zeropage-adressen skriven i instruktionen, adderar med `Y`, och använder det som en minnesadress. En typisk pointer dereference med en offset. Syntax: `LDA ($FE), Y`
- **Ackumulator**: Använder `A` registret direkt. Detta finns endast i instruktioner som har frivillig operand.

I den officiella syntaxen för 6502-assembly betyder `$` prefix att värdet efter är hexadecimalt (bas 16). För binärt används `%` t.ex. `LDA #%10101010`. `#` betyder att värdet ska laddas genom immediate adresseringsläge.

Instruktioner

Ett par användbara instruktioner är:

- **`LDA`**: Sätter in ett värde i `A` genom operanden.
- **`STA`**: Tar ett värde från `A` och sätter in det i minnet genom operanden.
- **`ADC`**: "Add with Carry", adderar `A` med ett värde via operanden, och adderar en extra `1` om carry-flaggan är satt, resultatet hamnar i `A`.
- **`SBC`**: "Subtract with Carry", subtraherar `A` med ett värde via operanden, och subtraherar en extra `1` om carry-flaggan inte är satt.
- **`ROR`**: "Rotera åt höger", flyttar alla av operandens bitar ett steg åt höger (neråt). Den tidigare lägsta biten hamnar i carry-flaggan och den gamla högsta biten får carry-flaggan som nytt värde. Ifall en operand inte skrivs använder den ackumulatorn.
- **`LSR`**: samma som ovan men `0` sätts in i den hösta biten.
- **`ROL`**: "Rotera åt vänster". Samma som `ROR`, fast andra hållet. Lägsta biten får carry-flaggan, högsta biten sätts in i carry-flaggan.
- **`ASL`**: samma som ovan men `0` sätts in i den lägsta biten.
- **`INC`**: inkrementerar ett värde i minne genom operanden. Kan *inte* användas med ackumulatorn.

- **INX**: inkrementerar X.
- **INY**: inkrementerar Y.
- **DEC**: dekrementerar ett värde i minne genom operanden. Fungerar ej på `A` likt `INC`.
- **DEY**: dekrementerar Y.
- **DEX**: dekrementerar X.
- **CLC**: tömmer C-flaggan.
- **SEC**: sätter C-flaggan.
- **JMP**: hoppar till operanden tolkad som absolut minnesadress.
- **CMP**: Subtraherar operanden från `A` utan att lagra resultat, men sätter flaggorna `C`, `N`, `Z`.
- **BEQ**: hoppar till operanden tolkad som relativ minnesadress om `Z`-flaggan är satt.
- **BNE**: hoppar till operanden tolkad som relativ minnesadress om `Z`-flaggan är tömd.
- **BCC**: hoppar till operanden tolkad som relativ minnesadress om `C`-flaggan är tömd.
- **BCS**: hoppar till operanden tolkad som relativ minnesadress om `C`-flaggan är satt.
- **BMI**: hoppar till operanden tolkad som relativ minnesadress om `N`-flaggan är satt.
- **BPL**: hoppar till operanden tolkad som relativ minnesadress om `N`-flaggan är tömd.
- **JSR**: hoppar till en adress och sparar vart ifrån hoppet gjordes. En **RTS** instruktion kan hämta den sparade adressen och hoppa tillbaka.

Program

Här är ett kort exempel på ett program som målar en linje med hjälp av en subrutin:

```
        ;; Sätt X och Y till $40 (64 i decimal) i början
        LDY #$40
        LDX #$40
for_pixel:
        ;; Målar en pixel vid X, Y
        JSR pixel_draw
        INY ; Y = Y + 1
        INX ; X = X + 1 ; Notera att $FF + $1 = $00
        ;; fortsätt loop om Y > 0
        BNE for_pixel
        RTS
```

Notera att programmet kör fram tills att `INX` tar `X` från `$FF` till `$00` genom en overflow vilket avslutar loopen. `for_pixel` är vad vi kallar en etikett eller “label”, det är inte maskinkod utan istället ett namn på en minnesadress som kompilatorn transformerar till en operand. `pixel_draw` är en till sådan etikett som har definierats någon annan stans.

Ett till exempel är aritmetik på tal som är 16-bits stora. Detta görs oftast genom att låta instruktionen som hanterar den lägre byten sätta carry-flaggan, och att instruktionen som hanterar den övre byten använder flaggan.

```
;; Multiplikation med 2 på 16-bitars tal.
;; Bitshift är ekvivalent med multiplikation med en tvåpotens på grund av
;; hur binära tal fungerar.

;; Talets lägre byte ligger i $DD och dess högre ligger i $DE.
        ASL $DD ; C-flaggan = biten som är utskiftad
        ROL $DE ; Inte ASL då vi vill använda C-flaggan
```

För annan typ av 16-bitars aritmetik, se Hugos artikel nedan.

Hur man får linjer att gå fort på en Commodore 64

av Hugo Ameln

ommodore 64 är en hemdator som lanserades 1982 etc. Det är mycket vi måste gå igenom så jag håller mig extremt kort, läs Wikipedia eller något. För att förstå innehållet i den här artikeln så rekommenderas det att du har läst den tidigare artikeln om 6502-assembly. För att måla linjer snabbt behövs en snabb algoritm, efter en kort Google sökning hittar man wikipediasidan *Bresenham's line algorithm*.

Linjealgoritmens Teori

Den här delen av artikeln är inte nödvändig för att förstå det resterande innehållet så du får skippa vidare. Vi ska måla en linje, dvs alla punkter (x, y) på linjen $y_l(x) = kx + m$. Linjen kan också uttryckas som en nivåkurva $f(x, y) = y - y_l(x)$, då uppfyller alla punkter (x, y) på linjen egenskapen $f(x, y) = 0$, vilket är egenskapen algoritmen utnyttjar. Mer specifikt så kommer alla punkter ovan linjen ha den fina egenskapen att

$$y > y_l(x) \Rightarrow f(x, y) = y - y_l(x) > 0$$

och $f(x) \leq 0$ för punkter därunder.

Om linjen börjar vid (x_0, y_0) och slutar vid (x_1, y_1) så gäller det att

$$k = \frac{y_1 - y_0}{x_1 - x_0} = \frac{\Delta y}{\Delta x} \Rightarrow f(x, y) = y - x\frac{\Delta y}{\Delta x} - m$$

För att göra det lättare för oss själva gör vi några inskränkningar. Vi begränsar oss till att $\Delta x \geq 0$, $\Delta y \geq 0$, dvs linjen skall luta uppåt. Dessutom inskränker vi att $\Delta x > \Delta y$ dvs att linjen inte får vara brant. Eftersom $f(x, y)$ bevarar sina fina egenskaper om man multiplicerar den med en positiv konstant som Δx, vilket vi gör och får

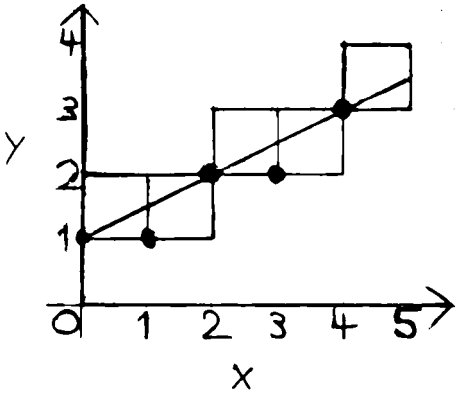
$$f(x, y) = y\Delta x - x\Delta y + M, \quad \text{där } M \text{ är konstant.} \tag{1}$$

Låt oss börja på en punkt på linjen $f(x_0, y_0) = 0$. Vi ska nu måla en punkt till höger så $x_1 = 1 + x_0$. Då skall $y_1 = y_0$ eller $y_1 = 1 + y_0$, då linjen lutar svagt. Låt oss undersöka f för punkten mellan dessa för att bestämma detta. Om $f(x_0 + 1, y_0 + \frac{1}{2}) > 0$ skall $y_1 = 1 + y_0$, annars $y_1 = y_0$. Efter detta är det bara att repetera algoritmen.

Vi är alltså bara intresserade av $D_n = f(x_n + 1, y_n + \frac{1}{2})$, vi har nu från ekvation 1 att

$$\begin{aligned}
\Delta D_n &= D_{n+1} - D_n = \Delta y - \Delta x \quad && \text{då } y_{n+1} = 1 + y_n, n \geq 1 \\
\Delta D_n &= D_{n+1} - D_n = \Delta y && \text{då } y_{n+1} = y_n, n \geq 1 \\
D_0 &= \Delta y - \frac{1}{2}\Delta x
\end{aligned} \tag{2}$$

Vi är intresserade av ΔD_n då det är väldigt effektivt att beräkna $D_{n+1} = D_n + \Delta D_n$



Notera att alla punkter som algoritmen finner inte ligger på linjen. En punkt kan inte ha större avstånd än $\frac{1}{2}$ l.e från linjen. Det skall sägas att D i koden faktiskt är $2D_n$. Detta är för att ekvation 2 inte skall innehålla bråk såsom $\frac{1}{2}$.

Linjealgoritmen

Här är linjealgoritmen i pseudokod:

Wikipedias Pseudokod

```
dx = x1 - x0
dy = y1 - y0
D = 2*dy - dx
y = y0
for x from x0 to x1
    plot(x, y)
    if D > 0
        y = y + 1
        D = D - 2*dx
    end if
        D = D + 2*dy
```

Våran Pseudokod

```
dx = x1 - x0
dy = y1 - y0
V = 2*dx +2*dy
D = 2*dy - dx + 510
dy_2 = 2*dy
for x from x0 to x1
    plot(x, y)
    if D > 510
        y = y + 1
        D = D - V
        break
    end if
        D = D + dy_2
```

Varför jag har modifierat pseudokoden lämnas som en uppgift åt läsaren.

Denna kod funkar bara för linjer i vinklar $0 \leq \alpha \leq \frac{\pi}{2}$ radianer. Men genom att göra små förändringar t.ex. att byta plats X och Y i algoritmen så målar man linjer av andra vinklar. För enkelhetens skull så undersöker vi endast linjer inom vinkeln $0 \leq \alpha \leq \frac{\pi}{2}$.

Vi kommer jobba i skärmläget Standard-bitmap-mode (320x200) men om vi begränsar oss på (255x200) så kan vi representera X-position och Y-position som 8-bitars värden. `D`, `dy_2` och `V` kommer tyvärr vara 16-bitar vilket vi får svälja. Notera att vi har adderat 510 till `D` sådant att $D > 0$.

I assembler kan vi representera for-loopen och dess makron som:

Kod Loop:

```
for_x:
    jsr pixel_draw
    ;;Inkrementera X till X_pos = X_end
    INC X_pos
    LDX X_pos
    CPX X_end
    BEQ end
    ;;Om D < $200 gå till case_2, annars case 1.
    Lag_16 D, D + 1, #$00, #$02, case_2
case_1:
    INC Y_pos
    Sub_16 D, D + 1, V, V + 1 ;D = D - V
    JMP for_x
case_2:
    Add_16 D, D + 1, dy_2, dy_2 + 1 ;D = D + 2*dy
    JMP for_x
end:
```

Makron:

```
;;a = a + b
;;Kommer avika med 1 om fast_unsafe = 1 samt C = 1.
def_macro Add_16(a_low, a_hi, b_low, b_hi, fast_unsafe = 0)
    if fast_unsafe == 0
        CLC

    LDA a_low
    ADC b_low
    STA a_low
    LDA a_hi
    ADC b_hi
    STA a_hi

;;a = a + b
;;Kommer avika med 1 om fast_unsafe = 1 samt C = 0.
def_macro Sub_16(a_low, a_hi, b_low, b_hi, fast_unsafe = 0)
    if fast_unsafe == 0
        SEC

    LDA a_low
    SBC b_low
    STA a_low
    LDA b_hi
    ADC b_hi
    STA b_hi

;; Om a < b gå till label annars så fortsätt
;;Större en makro, använder A-registert
def_macro Lag_16(a_low, a_hi, b_low, b_hi, label)
    LDA a_hi
    CMP b_hi
    BCC end
    BNE LABEL
    LDA a_low
    CMP b_low
    BCC label
end:
```

Låt mig förklara vad du just har tittat på. Konstanterna såsom `D`, `D + 1`, `V`, `V + 1` etc.. byts ut till adresser såsom `$42` och `$43` under kompilering. Dessutom byts alla makro-anrop ut till instruktioner under samma skede.

Vi kan dock göra ännu bättre. Då alla dekrement instruktioner sätter `Z`, och `C` som en `CMP #$00`, så kan vi göra något såsom:

Kod Loop:

```
LDX dx
for_x:
    jsr pixel_draw
    INC X_pos
    DEX
    BEQ end
    ...
```

En mindre detalj är att minnesadresserna vi använder såsom `$42` och `$43` alla är en byte. Programmet skulle köra om vi använde 2-byte adresser såsom `$4242` men då så blir alla instruktioner som använder dessa en CPU-cykel långsammare. Dessa unikt snabba 8-bitarsadresser kallas för zeropage.

Det dyraste steget i loopen är där vi exekverar `pixel_draw`. Skärmen har en buffert från `$4000 - $5F3F` och varje byte i bufferten korresponderar till 8 pixlar på skärmen. T.ex. för att måla dem första 8 pixlarna randiga så kan man köra

```
LDA #%10101010
STA $4000
```

Här är en bild jag har gjort för att illustrera vilka minnesadresser som korresponderar till vilka bytes av pixlar på skärmen

$4000 %10101010	$4008	$4010	$4018	...
$4001	$4009	$4011	$4019	...
$4002	$400A	$4012	$401A	...
$4003	$400B	$4013	$401B	...
$4004	$400C	$4014	$401C	...
$4005	$400D	$4015	$401D	...
$4006	$400E	$4016	$401E	...
$4007	$400F	$4017	$401F	...
$4140	$4148	$4150	$4158	...
$4141	$4149	$4151	$4159	...
$4142	$414A	$4152	$415A	...
...	...	...	...	...

Det bästa sättet att hitta rätt minnesposition att rita till är att använda en uppslagstabell, det är typ så `pixel_draw` funkar, alltså om man inte använder sig av relativ positionering.
Du kanske också har märkt att y-axeln är nedåt i programet till skilnad från matten där den är definierad uppåt. Lösningen är att rotera datorskärmen π radianer och sedan kolla på linjen via en spegel.

Vi kan snabba upp loopen avsevärt genom att räkna ut den nya pixeln att måla med hjälp av koordinaterna av den förra (vilket gör allting mer komplicerat).

```
    jsr pixel_draw ;Vi målar endast första pixeln med pixel_draw!
                   ;Vilket sätter dessa värden
                   ;byte to paint position: byte_position* + Y-register
                   ;Notera att: Y-register = (mod 8) [byte to paint position]
                   ;byte_position = [byte to paint position] - Y-register
                   ;byte av pixlar vi ska måla (innehåller en flippad bit):
                   ;                                          byte_to_paint
    LDX dx ;X = X_end - X_pos
for_x:
    ;;Målar byte_to_paint till adressen byte_position* + Y-register
    LDA byte_to_paint ;A är en byte med en enskild 1 och resten nollor.
    ORA (byte_position), Y
    STA (byte_position), Y
increment_pixel_x:
    LSR byte_to_paint ;roterar pixeln 1 bit till vänster på skärmen.
    BCC increment_pixel_x_end
move_8px_right: ;vi kan inte flytta på 1:an mer så vi måste byta adress
    ;;fixa tillbaka byte_to_paint till #%10000000
    ROR byte_to_paint
    ;;byt till en adress åt höger
    Add_16 byte_position, byte_position + 1, #$08, #$00, 1 ;C = 0 från ROR
increment_pixel_x_end:
    DEX
    BEQ end ;nu har vi målat färdigt linjen
    ;;Om D < $200 gå till case_2, annars case_1.
    Lag_16 D, D + 1, #$00, #$02, case_2
case_1:
    ;;Vi skall utföra D = D - V
    ;;Lag_16 ger att: C = 1 och att A = D
    SBC V
    STA D
    LDA D + 1
    SBC V + 1
    SYA D + 1
increment_y_pos:
    INY ;Vi kan inkrementera Y, om det når 8 behöver vi ändra på byte_position
    CPY #$08
    BNE for_x
move_8px_down: ;vi måste ändra på byte_position
    LDY #$00 ;Nollställ Y-registret
    ;CPY ger att C = 1
    ;så vi ska subtrahera #$013F + C
    Add_16 byte_position, byte_position + 1, #$3F, #$01, 1
    JMP for_x
increment_y_pos_end:
case_2:
    Add_16 D, D + 1, dy_2, dy_2 + 1, 1 ;D = D + dy_2
    JMP for_x
end:
```

Som du ser används många tricks, ett av dem är att använda osäker-flaggan till makron så att vi kan skippa CLC och SEC instruktioner, men såklart kan det gå snabbare! Vi kan faktiskt bli av med den sista CMP instruktionen. Om vi målar linjen åt andra hållet så blir CMP #$08 en CMP #$FF eftersom DEY transformerar Y = 0 till Y = $FF. Om vi nu låter Y vara ett värde högre en vad det skall och btp_mem_pos vara ett mindre en var det skall så har vi en CMP #$00 vilket DEY gör automatiskt.

En intressant detalj är att LDA #$42 är snabbare en LDA $42 med en CPU-cykel (samma för SBC). Därmed skulle koden vara snabbare om dy_2, dy_2 + 1, V, V + 1 var den samma för alla linjer.

Självmodifierande Kod

Lösningen är självmodiferande kod. LDA #$42 är egentligen 2 bytes efter varandra i minne %10101001, %01000010 så vi kan lösa problemet genom att ändra den andra byten under programmets gång. Här har vi programmet med alla optimeringar som har diskuterats:

```
selfmod:
    ;;Självmodiferande kod. Får LDA samt SBC att ta en cpu cyklel mindre.
    ;;dy_2
    ;;Modifierar LDA dy_2
    LDA dy_2
    STA case_2 + 1
    ;;Modifierar LDA dy_2 + 1
    LDA dy_2 + 1
    STA case_2 + 7
    ;;V
    ;;Modifierar SBC V
    LDA V
    STA case_1 + 1
    ;;Modifierar SBC V + 1
    LDA V + 1
    STA case_1 + 7
end_selfmod:

jsr pixel_draw ;Vi målar endast första pixeln med pixel_draw,
               ;;efter detta används relativ positionering!
               ;;Vilket sätter dessa värden
               ;;byte to paint position: byte_position* + Y-register
               ;;Notera att: Y-register = (mod 8) [byte to paint position]
               ;;byte_position = [byte to paint position] - Y-register
               ;;byte av pixlar vi ska måla (innehåller en flippad bit):
               ;;                                          byte_to_paint
               ;; Sätter C = 0
    LDX dx ;X = X_end - X_pos
    INY ;; Y måste vara 1 förmycket
    ;;därmed skall byte_position vara 1 för lite.
```

```
    Sub_16 byte_position, byte_position + 1, #$00, #$00, 1; -1 =  -$0000 - (1-C)
    INY
for_x:
    ;;Målar byte_to_paint till adressen byte_position* + Y-register
    LDA byte_to_paint ;A är en byte med en enskild 1 och resten nollor.
    ORA (byte_position), Y
    STA (byte_position), Y
decrement_pixel_x:
    ASL byte_to_paint ;roterar pixeln 1 bit till vänster på skärmen.
    BCC decrement_pixel_x_end
move_8px_right: ;vi kan inte flytta på 1:an mer så vi måste byta adress
    ;;nollställer byte_to_paint till #%00000001
    ROL byte_to_paint
    ;;byt till en adress åt höger
    Sub_16 byte_position, byte_position + 1, #$07, #$00, 1 ;-8 = -7 - (C - 1)
decrement_pixel_x_end:
    DEX
    BEQ end ;nu har vi målat färdigt linjen
    ;;Om D < $200 gå till case_2, annars case_1.
    Lag_16 D, D + 1, #$00, #$02, case_2
case_1:
    ;;Vi skall utföra D = D - V
    ;;      C = 1 pga Lag_16
    ;;      Detsutom är A = D
    SBC #$42 ;detta blir överskrivet
    STA
    LDA D +1
    SBC #$42 ;detta blir överskrivet
    SYA D + 1

decrement_y_pos:
    DEY ;Dekrementera Y-position
    BNE for_x
move_8px_up:
    LDY #$08 ;Nollställ Y-registret
    ;;DEY ger att C = 1
    ;;så vi skall subtrahera #$0140
    Sub_16 byte_positionition, byte_positionition + 1, #$40, #$01, 1 ;+320
    JMP for_x
decrement_y_pos_end:
case_2:
    Add_16 D, D + 1, #$42, #$42, 1 ;D = D + dy_2
    JMP for_x
end:
```

Jag skulle egentligen ha avslutat artikeln här men jag hittade en till förbättring! Det visar sig att algoritmen går ännu snabbare om D är ett signerat 16-bitars tal (i tvåkomplementsform). Vi låter alltså $D = 2 \cdot dy - dx$ istället för $D = 2 \cdot dy - dx + 510$. Jag kommer låta bli att förklara varför `Add_16` och `Sub_16` fortfarande fungerar (artikeln är redan för lång). När man kör instruktionen `LDA D + 1` så laddas `A` med övre byten av D,

det vet ni ju redan, men N-flaggan får samtidigt värdet av teckenbiten. Det här förenklar förgrenings-logiken betydligt (se decrement_pixel_x_end):

```
selfmod:
    ... ;Tekniskt sätt ändrar vi lite här men det är inte viktigt.
for_x:
    ...
decrement_pixel_x_end:
    DEX
    BEQ end
    ;;Om D < 0 gå till case_2
    LDA D + 1
    BMI case_2
    ...
move_8px_up:
    LDY #$08
    ; C = 1 eller 0 baserat på vad som händer inom ... området.
    ; Det ändrar på sig under iterationerna.
    Sub_16 btp_mem_pos, btp_mem_pos + 1, #$40, #$01;
    JMP for_x
    ...
case_2:
    ;; Här ser allt normalt ut men C = 1
    ;; vi löser detta genom att låta dy_2 vara 1 förlite hela tiden!
    Add_16 D, D + 1, #$42, #$42, 1 ;D = D + dy_2
    JMP for_x
end:
```

Om du kommer på ett sätt att snabba upp koden ytterligare kontakta gärna mig via mejladressen hugova@stacken.kth.se

Kompilator del 2: mot hård metall

av John Lorentzson

Kompilerad kods beteende

För att kod ska fungera ihop med annan kod måste den följa vissa regler och överenskommelser. Vi har ett par sådana. Först, vart lagrar vi variabler? Vi kommer behöva en plats att lagra värden som inte används omedelbart, och tack vare att 6502:an har så få register kan vi göra det väldigt enkelt. Vi använder varken

register `X` eller `Y` förutom i specialfall eftersom de inte kan utföra generell aritmetik och skulle komplicera kompilatorn. Användarens program har ett ställe att lagra variabler, den så kallade variabelvektorn. Variabelvektorn är en plats som ligger lågt i C64:ans minne och har en viss ospecifik längd. Den måste placeras så att ingen kod i systemet förutom den kompilerade koden använder den, och därför har den platsen valts genom överenskommelse mellan mig och Hugo.

Det är det simpla, ingen får kliva på vår data och vi får inte kliva på någon annans. Det andra vi måste bestämma är hur vi anropar de inbyggda funktionerna (primärt skrivna av Hugo). De två viktiga sakerna att bestämma är hur vi ska ge argument till funktionerna, och hur vi ska få returvärden tillbaka. En mer sofistikerad kompilator skulle kunna välja olika sätt beroende på sammanhang, men för att hålla det simpelt bestämde vi oss för ett fast system. Funktioner ger returvärden i ackumulatorn (`A`), och vi ger argument genom *argumentvektorn*. Argumentvektorn fungerar liknande till variabelvektorn, fast ska aldrig läsas från av kompilerad kod. Under en inbyggd funktions exekvering får den göra vad den vill med argumentvektorn, så vi kan därmed inte göra några antaganden om dess innehåll efteråt. En sådan här överenskommelse om hur funktioner anropas kallas för en "calling convention".

Vi bestämmer, eftersom användarens kod ska köras om och om igen, men måste kunna avbrytas av signaler från användaren, att användarens program ska avslutas med en `RTS` ("return from subroutine") maskininstruktion.

Pre-assembly

Innan vi kan mata ut maskinkod från vår IR finns det ett par optimeringar som måste göras, men som är CPU-specifika. Det första vi måste göra är att byta ut vissa operatorer mot funktionsanrop. 6502 processorn har inga instruktioner för multiplikation eller division, så det (och möjligtvis annat) måste göras i mjukvara. Som tur är har Hugo skrivit sådana funktioner, och jag antar att de är korrekta. Utbytet görs simpelt genom att gå igenom alla instruktioner, och för de som måste göras i mjukvara så sparar vi undan operatorns inputs och output, tar bort dem från den instruktionen, vi sätter in en ny `IR-CALL` instruktion ovanför, och ger den de inputs och output som vi sparade. Sedan raderas originalinstruktionen. De operatorer som återstår representerar på ett eller annat sätt riktiga maskininstruktioner.

6502 har endast en generellt användbar register, ackumulatorn (`A`). Många maskininstruktioner, primärt operatorer, tar även en direkt parameter som antingen kan vara en konstant eller någon slags referens till ett värde i minnet. Det gör `A` och den direkta operanden till operatorns båda operander. Till exempel, i 6502-assemblyspråk skrivs subtraktionen `foo - bar` som följande:

```
LDA FOO ; Ladda in värdet vid adressen FOO till ackumulatorn
SEC     ; Sätt på "carry"-flaggan, en nödvändig bökighet med 6502:ans aritmetik
SBC BAR ; Subtrahera värdet vid adressen BAR från värdet i ackumulatorn
```

Vi vet därför att det första inputvärdet till en operator måste laddas, medans det andra inte behöver laddas, eftersom maskininstruktionen i sig sköter det. Om vi vill göra saker lättare till senare så kan vi redan nu, ifall IR-instruktionen som definierar det första värdet är `IR-FETCHVAR` eller `IR-GETCONST`, flytta definitionsinstruktionen till precis ovanför användningen. För det andra värdet kan vi, om instruktionen är `IR-FETCHVAR` eller `IR-GETCONST`, byta ut det relevanta värdet mot den direkta variabeln eller konstanten. Dvs. byta ut ett `IR-RESULT` mot en `IR-VARIABLE`, eller en `IR-CONSTANT` mot ett vanligt heltal.

Värdesallokering

Trots att det typiskt heter registerallokering så kallar jag istället just den här implementationen av konceptet för "värdesallokering", eftersom vi endast har en äkta register som vi faktiskt använder. Istället allokerar vi primärt platser i minnet som om de vore register. De platser vi kan allokera värden till är:

- **Variabelvektorn**, som innehåller både namngivna variabler som måste stanna kvar mellan exekveringar av användarens kod, och temporära variabler.
- **Argumentvektorn**, som innehåller argumenten till nästa kommande funktionsanrop. För enkelhets skull antas hela argumentvektorns innehåll vara förstörd efter att ett funktionsanrop är färdigt.
- **Ackumulatorn (`A`)**, endast om värdet omedelbart används.

Värdesallokeringen fungerar genom att bestämma en **allokeringsstrategi** för varje bit IR-data som har skapats, och den följer väldigt simpla regler för att göra så.

- *Är värdet av typen `IR-CONSTANT`?* ⟶ Konstant, ge inget eget utrymme.
- *Är värdet av typen `IR-VARIABLE`?* ⟶ Namngiven variabel, spara i unik plats i variabelvektorn.
- *Saknar värdet användare?* ⟶ Spara ej.
- *Är värdets ända användare ett funktionsanrop, och finns det inget annat anrop mellan definition och användare?* ⟶ Placera direkt i rätt plats i argumentvektorn.
- *Är värdets sista användning IR-instruktionen omedelbart efter definitionen, är värdet det första inputvärdet till användaren, och är användaren en som använder ackumulatorn?* ⟶ Låt sitta i ackumulatorn, ge inget eget utrymme.
- *Är värdets definition en `IR-COMPARISON`, är den sista användningen omedelbart efter definitionen, och är användaren en `IR-IF`?* ⟶ Ska användas för branching, ge inget eget utrymme. Exakt beteende förklaras senare.
- *Annars…* ⟶ Temporär variabel, får en plats i variabelvektorn efter alla namngivna variabler.

Att ge varenda temporära värde en permanent plats i variabelvektorn är kanske inte den mest effektiva användningen av vårt RAM, så man vill helst återanvända platser för nya temporära variabler efter deras sista användningar. Detta görs, men min lösning är inte intressant nog att sätta på papper.

Maskinkodsstruktur

Instruktionssatsen för 6502 processorn är ganska simpel, så det finns väldigt lite data vi måste lagra innan vi matar ut den färdiga listan med bytes. Det finns två typer av vad jag kallar för "assemblyobjekt": "label" (`ASM-LABEL`), som helt enkelt är en markör för en minnesadress, och instruktion (`ASM-INSTRUCTION`). Dessa är hoplänkade i en lista och båda typer av `ASM-OBJECT` har en adress. Denna adress är den faktiska minnesadressen som maskinkoden kommer finnas på, eller som labeln kommer referera till. Labels har även namn, och instruktioner har en opcode (ett nummer som identifierar instruktionens typ), operand, och längd i bytes. Det finns tre längder på instruktionerna som 6502 har: 1, 2, och 3, beroende på om det finns en operand, och om operanden är 1 eller 2 bytes lång. Det är opcode som bestämmer vilket adresseringsläge som används och därmed är längden alltid samma för samma opcode.

Kodgenerering

Generering av maskinkod behöver nästan alltid göras i två pass. Under första passet ska vi mata ut maskininstruktionerna vi vill ha, och labels på de ställen i koden vi vill att maskininstruktionerna ska referera till. I första passet kan en label användas som operand, även innan den har matats ut. Detta gör det möjligt att, till exempel, hoppa framåt i programmet. Vi använder i det här passet även objekt som representerar inbyggda funktioner som operand till `JSR`. Medans vi matar ut maskininstruktioner så räknar vi upp adresserna genom de utmatade maskininstruktionernas längd. Vi vet därmed vart alla maskininstruktioner ligger i minnet, och vart alla labels sitter. Att kompilera första passet är väldigt simpelt, vi går helt enkelt igenom alla IR-instruktioner och kompilerar dem för sig, och uppdaterar adressräknaren. Mer om hur IR-instruktionerna kompileras senare.

I det andra passet måste vi transformera alla referenser till labels, `IBLOCK`, och inbyggda funktioner till de adresserna som de faktiskt representerar. Ett separat pass är nödvändigt för de första två eftersom man inte kan veta vilken adress en något kommer att få innan all kod som kommer innan den har matats ut. Detta pass är också simpelt. Vi går igenom alla genererade maskininstruktioner och byter ut operander som är `IBLOCK` mot den label som har samma namn, labels mot dess adress, och inbyggda funktioner mot funktionens adress som vi har fått från assemblyprogrammet som implementerar dem.

Hantering av data under kodgenerering

Vi behöver sätt att generellt placera data där det ska vara beroende på vad den har för allokeringsstrategi. Till exempel, för vissa strategier ska en store ignoreras och en load vara ett error, för vissa andra ska load och store röra minnesadresser. Vi behöver också ett sätt för koden som genererar instruktioner åt dessa generella load och store funktioner att se skillnad på konstanter och adresser. Att separera detta från allokeringsstrategi låter oss även använda generella load och store med egna beräknade adresser.

En funktion `DATA-REFERENCE` tar in data och konstruerar baserat på dess allokeringsstrategi en liten wrapper som innehåller det som ska bli operand (dvs. adressen där datan lagras, eller värdet ifall det är en konstant) samt om den ska användas som konstant (kallad för "immediate" när värdet används direkt som operand) eller minnesadress. Värden som inte lagras i variabelvektorn eller argumentvektorn ska aldrig refereras till som operander, så detta ger ett error.

Att generera kod som laddar data är ganska simpelt. Ifall strategin är att spara i ackumulatorn behöver inget göras, ackumulatorn ska vid den här punkten i exekveringen innehålla rätt värde. Liknande gäller för om ett värde används direkt för en branch, eller om värdet lagrades direkt i argumentvektorn, eller om den förra instruktionen redan laddade eller lagrade värdet. I alla andra fall (namngivna och temporära variabler) matas det ut en `LDA` instruktion genom en datareferens till datan.

Att lagra är lite mer komplicerat, men inte mycket. För de strategierna som inte kräver minnesplats görs ingenting, för värden som ska direkt till argumentvektorn hittar vi platsen dit det ska genom att gå igenom användarens inputs och letar efter datans position. Eftersom data kan återanvändas så matas det ut en passande `STA` instruktion för varje instans av datan i inputs listan. Vi använder i det fallet en adress beräknad direkt på plats (`argvec-position + argvec-offset`) eftersom vi inte kan få en generell adress till återanvändbara värden. För värden som lagras i variabelvektorn matas endast ut en `STA` instruktion, denna gång genom en datareferens.

Dessa två saker implementeras som funktioner, `EMIT-LOAD-DATA` och `EMIT-STORE-DATA`. Det är anroparens ansvar att se till att instruktionerna som leder upp till vad dessa funktioner kan mata ut matchar. Det vill säga att om ett värde ska lagras finns det i `A`, och om ett värde ska användas ska det användas *från* `A`. Detta visar sig vara väldigt simpelt att göra.

Till att matcha med dessa har vi även ett par "emitters" som jag kallar dem, funktioner som matar ut olika opcodes baserat på typen av datareferens som operanden är. Det finns tre versioner av instruktioner som är relevanta här: immediate, zeropage, och absolute. Dessa förklaras i tidigare artikeln om 6502-assembly. En emitter tittar på referenstypen och väljer passande opcode, immediate om det är en immediate, zeropage om en kort nog adress, annars absolute. Detta resulterar i funktioner som matar ut passande opcode och operand från en datareferens.

Kompilering av IR-instruktioner till maskininstruktioner

Vi börjar simpelt och ökar komplexiteten därifrån. Först, `IR-RETURN`. Den här är väldigt simple, den blir till `RTS`, som har opcode `$60` (decimalt 96), tar ingen operand, och har därmed en längd på 1 byte. Inget konstigt här. Sedan kommer `IR-ASSIGN` och `IR-FETCHVAR`. Dessa visar sig vara identiska. Först tar vi och kör `EMIT-LOAD-DATA` på input, `EMIT-STORE-DATA` på output. Om allokeringsstrategin kräver den kommer värdet laddas och lagras som det ska. `IR-GETCONST` är lite annorlunda, där matar vi istället direkt ut en `LDA` som tar en immediate med vårt konstanta värde. Därefter, en vanlig `EMIT-STORE-DATA` av outputen.

`IR-PLUS` och `IR-MINUS` kommer sedan, och är någorlunda mer involverade, eftersom de gör mer än bara laddning och lagring. `IR-PLUS` kompileras genom att först köra `EMIT-LOAD-DATA` på sin första input, mata ut en `CLC` instruktion (eftersom 6502 saknar en "add without carry"), och sedan använda emittern för `ADC` med en datareferens till det andra input värdet. Detta matchar vad som beskrevs tidigare, att en operators första input hamnar i `A` medans den andra blir en operand till maskininstruktionen. Sedan körs bara `EMIT-STORE-DATA` på output och så är vi klara med den. Samma sak gäller för `IR-MINUS` fast med `SEC` och `SBC` istället för `CLC` och `ADC`.

`IR-JUMP` är simpel att kompilera, framförallt om destinationen är nästa `IBLOCK`. I det fallet behöver vi inte göra något alls eftersom nästa kompilerade instruktion kommer vara destinationen ändå. Annars matas det ut en `JMP` instruktion (opcode `$4C`) med destinationen som operand. Längd blir tre (opcode + 2 bytes adress).

Innan vi går vidare till den mest komplicerade IR-instruktionen att kompilera, och det som är kopplat till den, tar vi `IR-CALL`. Först går vi igenom alla inputs och ifall en inputs allokeringsstrategi inte var att lagras direkt i argumentvektorn så matar vi ut en `LDA` för den följt av en `STA` till en adress som pekar till värdets tänkta position i argumentvektorn. Därefter matar vi ut en `JSR` instruktion (opcode `$20`) och kör `EMIT-STORE-DATA`.

Branching och tester

Tester är i princip rätt simpla: en jämförelse mellan två uttryck sker, och returnerar ett boolvärde (eller i vårt fall, 1 eller 0) som indikerar om testet lyckades (sant, `1`) eller misslyckades (falskt, `0`). Problemet kommer när vi tittar på hur instruktionerna som implementerar dessa tester fungerar. Eller rättare sagt inte gör det. Precis som många CPU-arkitekturer har 6502 en instruktion för jämförelse, `CMP`, som fungerar som en subtraktion som inte sparar sitt värde, utan bara sätter statusflaggor. Dessa flaggor går inte att direkt spara som värden utan mycket bök, och för vissa jämförelser är värdet av flera olika flaggor relevant. Som tur är finns det instruktioner som arbetar med flaggorna: branchinstruktionerna.

Vi behöver inte gå igenom varje test och vilka flaggor som betyder vilket resultat, vi går istället igenom den generella funktionen för att mata ut testkod. Vi börjar med att köra `EMIT-LOAD-DATA` på vår första input, den som ska sitta i `A`. I det fallet då vi ska lagra resultatet, och inte direkt utföra en branch, fungerar koden som matas ut ungefär så här:

1. Ladda det vanligtvis oanvända registret `X` med en konstant `1`.
2. Utför en jämförelse via `CMP`, som sätter statusflaggorna.
3. Ifall testet lyckades, hoppa över nästa steg genom passande branchinstruktion(er).
4. Ifall testet misslyckades, ladda `X` med en konstant `0`.
5. Nu, oavsett resultatet, använd `TXA` instruktionen för att flytta `X` till `A`.

Detaljerna varierar lite, eftersom vissa tester kräver två branchinstruktioner för att utföra testet korrekt, och en `NOP` (“no operation”) instruktion måste sättas in för att de relativa hoppen ska bli rätt.

Till sist görs `EMIT-STORE-DATA`.

I det fallet då testet används direkt i en branch blir saker något enklare. Vi är helt enkelt klara direkt efter att vi har matat ut branchinstruktionerna. Det är upp till kompileringen av `IR-IF` att se till att saker fortsätter korrekt.

`IR-IF` börjar med en `EMIT-LOAD-DATA`, nödvändigt ifall testet är en variabel eller annat värde istället för ett direkt test. Ifall vår input är ett test vars allokeringsstrategi är att användas direkt i en branch behöver vi bara göra en sak: sätt in en `JMP` instruktion som tar oss till ELSE-fallet av våran `IF`. Detta fungerar eftersom koden som testets kompilering matade ut är en branch som hoppar över en instruktion ifall testet är falskt, dvs. i ELSE-fallet. Därefter är vi klara med vår `IF` eftersom THEN-fallet kommer att komma direkt efter.

Ifall vår input *inte* är ett direkt test måste vi först testa värdet som kom in. `EMIT-LOAD-DATA` placerade det i `A` genom en `LDA` instruktion, som sätter “är noll”-flaggan, så allt vi behöver göra är mata ut en `BNE` instruktion som likt ett vanligt test hoppar över en instruktion, och den instruktionen den hoppar över är ett likadant `JMP` som i det andra fallet.

Sista steget

Nu har vi en lista med objekt, och de som representerar instruktioner ska vid det här laget endast innehålla siffror. Allt vi behöver göra nu är gå igenom alla instruktioner och spara deras opcode följt av operand. Och då har vi en färdig lista med bytes som en 6502 kan köra som vårt program. Då är det bara att skicka över till C64:an och låta den köra.

Jag hoppas att dessa artiklar har lyckats få kompilatorn att verka som ett någorlunda simpelt program att förstå, och att man inte måste vara någon expert för att skriva en kompilator. Källkoden är inte den renaste någonsin, men finns tillgänglig att läsa här: https://git.stacken.kth.se/Stacken/c64-livecoding

Jag älskar GNOME

av John Lorentzson

GNOME är nog det ända desktop environment på något system som försöker göra något intressant utan att antingen missförstå vad som ska gå snabbt (hint för Windows och macOS: det man gör ofta ska gå snabbt) eller att ta en helg att konfigurera. Jag älskar nördskit, men jag har inte tid eller energi att engagera mig i nördskit bara för att få datorn att bli användbar. GNOME är rent, simpelt, ser rätt snyggt ut bredvid de tråkiga kollegornas Windows datorer (inget illa menat mot Windowsanvändare), och har den bästa idén en Linux desktop environment någonsin har haft*: dynamiska workspaces. När jag vill ha mer arbetsutrymme behöver jag bara hoppa över till höger, och där finns det mer utrymme!

Det finns alltid ett tomt workspace längst till höger, oavsett hur många du redan använder. Det går att skapa nya workspaces mellan existerande genom att dra in fönster dit, och man kan snabbt skrolla genom dem, antingen med mus, touchpad, eller tangentbord, så ingen onödig context switching. Jag använder många olika program och planerar inte min datoranvändning i förväg varje månad, så att inte behöva tänka på vad som ska vara på vilken workspace är fantastiskt, jag kan bara göra fler! GNOME's workflow har börjat kännas så naturligt för mig att nästan alla andra desktop environments känns som att återvända till Windows. Nej tack.

Tangentbordsgenvägarna är för det mesta trevliga och att lägga till fler genom inställningarna är lätt. Det inbyggda `Super`+<n>† för att köra/hoppa till favoritprogram är fantastiskt när man har några stycken väldigt ofta använda program. I mitt fall Firefox, Emacs, Discord, och ett musikprogram. Genvägar finns även för hantering av workspaces, så de fungerar fortfarande bra om man inte rör musen så ofta.

Simpla stiländringar kan göras genom CSS, och det *låter* hemskt (och kanske är det) men det betyder att jag kan få en fin gradient i min top bar utan något bök. Hur man gör en gradient i CSS är redan väldokumenterat!

*Inga garantier att det är den bästa idén, eller att GNOME uppfann den.

†`Super` är tangenten som Microsoft® bestämde sig för att göra till sin egna Windows® Key, men den heter faktiskt `Super`.

Jag hatar GNOME

av John Lorentzson

Att använda GNOME utan extensions känns som en iPhone från 13 år sedan (“du skall använda enheten som designad, passar det inte vår design kan det inte göras”), men webbsidan man förväntas ladda ner extensions från som gör GNOME till en rimlig miljö att arbeta i ser cirka 5 år äldre ut än den metaforiska mobiltelefonen, och är inte lätt att hitta (antagligen för att någon av rimliga skäl skäms över hur den ser ut). Extensions är nästan totalt nödvändiga. Till exempel, vissa program fortsätter köra efter att du stänger fönstret och lämnar en liten ikon efter sig. GNOME visar inte dessa, så om du har ett sånt program kommer du inte kunna se om det fortfarande kör eller inte. Men självklart *kan* GNOME visa ikonerna. Och hur får man det? Kanske genom *Settings*? Självklart inte, det hade varit förvirrande för nya användare. Istället måste du ladda ner paketet `gnome-extensions` (som på minst ett distro heter något annat) och sätta på en extension som heter “AppIndicator and KStatusNotifierItem Support”. Kan *du* tänka dig en mer intiutiv lösning? Extensionen i sig är iallafall förinstallerad.

Att göra stiländringar förutom “dark mode (yes/no)”, “accent color”, eller bakgrundsbild kräver `gnome-tweaks`, som naturligtvis också saknar funktionalitet. Vill du att bara en av `Super` tangenterna ska öppna overview? Visst, det finns knappar för det i Tweaks, bara klicka på den du vill ha! Vill du ha tillbaka defaultbeteendet där båda fungerar? Det är en avancerad ändring som kräver terminalen. Visst är det coolt att man kan använda CSS på sin desktop environment? Ja, så coolt att det kräver en extension (som den här gången *inte* är förinstallerad) för att fungera trots att huvudfunktionaliteten sitter i GNOME.

Vill du se alla dina program? Javisst! Du kan få en appmeny som påminner om iOS från 2010 (med något slöare beteende och prestanda). Vill du inte se allt skrot som Figge Ferrum dumpar på gården din när hans program installeras? Spendera ett par timmar på att ta varenda ikon du inte vill se och dra in dem i en skräpmapp. Detta kan ta ett tag eftersom varenda försök att flytta en ikon kan leda till att alla ikoner som nuddas på vägen flyttas och skapar nya tomrum. Vill du ha automatiska kategorier (som systemet redan vet om)? Då får du sätta på en extension (som inte byter ut den vanliga appmenyn).