CHALMERS

Dynamically Recompiling Emulator

Dynamic recompilation of MOS 6502 machine code through LLVM IR using a JIT compiler

Bachelor's thesis for the department of Computer Science and Engineering

Carl Blomqvist
Matilda Falkenby
William Kalin
Felix Korch
Carl Lundgren
Markus Pettersson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Kandidatarbete

Dynamiskt Omkompilerande Emulator

Dynamisk omkompilering utav MOS 6502-maskinkod, genom LLVM IR med hjälp av JIT-kompilering

Carl Blomqvist
Matilda Falkenby
William Kalin
Felix Korch
Carl Lundgren
Markus Pettersson

CHALMERS

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola
Göteborg, Sverige 2020

Dynamically Recompiling Emulator
Dynamic recompilation of MOS 6502 machine code through LLVM IR using a JIT compiler.
CARL BLOMQVIST
MATILDA FALKENBY
WILLIAM KALIN
FELIX KORCH
CARL LUNDGREN
MARKUS PETTERSSON

Supervisor: Torbjörn Tjelldén, Computer- and Information Technology
Examiner: Arne Linde, Computer- and Information Technology

Bachelor's Thesis
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Transparent image of NES console with a controller [1].

Typeset in LaTeX
Gothenburg, Sweden 2020

Dynamically Recompiling Emulator
Dynamic recompilation of MOS 6502 machine code through LLVM IR using a JIT compiler

Carl Blomqvist
Matilda Falkenby
William Kalin
Felix Korch
Carl Lundgren
Markus Pettersson

Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This report examines the possibility to implement dynamic recompilation of machine code for the central processing unit MOS 6502. Using libraries for compilation and optimizations, from the LLVM project, an emulator for the MOS 6502 processor was developed. In addition to the dynamically recompiling emulator, an interpreting emulator was developed as well. The interpreting emulator was used as a reference when assessing the execution time of the machine code generated by the dynamically recompiling emulator. It was shown that the execution time of MOS 6502 programs was lower when executed on the dynamically recompiling emulator, compared to the interpreting emulator. The dynamically recompiling emulator has an associated startup time, which in some cases leads to the interpreting emulator finishing its execution quicker overall.

Keywords: emulation, dynamically recompiling emulator, interpreting emulator, dynamic recompilation, JIT, LLVM, MOS 6502

Sammandrag

Rapporten undersöker möjligheten att implementera dynamisk omkompilering av maskinkod menad för processorn MOS 6502 med hjälp av LLVM. Genom att kombinera LLVMs bibliotek för kompilering och optimering utvecklades en emulator för processorn MOS 6502. Dessutom utvecklades en interpreterande emulator, vilken används som utgångspunkt för att jämföra och analysera exekveringshastigheten av maskinkoden genererad av den dynamiskt omkompilerande emulatorn. Exekveringen av programmen som testades på de två emulatorerna utfördes alltid snabbare på den dynamiskt omkompilerande emulatorn jämfört med den interpreterande emulatorn. Den dynamiskt omkompilerande emulatorn har en uppstartningsfas som i vissa fall leder till att den interpreterade emulatorn överlag slutför exekveringen snabbare.

Nyckelord: emulering, dynamiskt omkompilerande emulator, interpreterande emulator, dynamisk rekompilering, JIT, LLVM, MOS 6502

Tack till

Vår fantastiska handledare, Torbjörn Tjelldén. Tack för att du var en fluga på väggen, för att du styrde oss rätt, och för att du fick oss att verkligen arbeta som ett lag! Stort tack också till vår examinator Arne Linde, som svarat på alla våra (många) frågor kring kandidatarbetet.

Innehåll

Lista över akronymer

API	Application Programming Interface
CPU	Central Processing Unit
DRC	Dynamic recompilation
IMGUI	Immediate Mode Graphical User Interface
IR	Intermediate Representation
ISA	Instruction Set Architecture
JIT	Just In Time
JMP	Jump - hoppinstruktion
LLVM	Low Level Virtual Machine
NES	Nintendo Entertainment System
NMI	Non-Maskable Interrupt
NOP	No Operation
PC	Programräknare
PPU	Picture Processing Unit
RAM	Random Access Memory
ROM	Read-Only Memory
SP	Stack Pekare

Ordlista

Bytecode	Intermediärt kodformat
Gäst	Det emulerade systemet
Label	Symboler som anger ett läge i ett assemblerprogram
Mnemonic	Minnesregel, ord som tolkas som en assemblerinstruktion
Offset	Avståndet (heltal) från början till ett visst objekts position
Opcode	En byte stor unik kod som representerar en processorinstruktion
Overhead	Extra arbete, barlast (onödigt material)
Värd	Systemet som kör en emulator
Wrapper	klass som är till för att kapsla in hela system eller program

1
Inledning

Det finns många vardagliga applikationer av emulering, såsom emulering av nostalgifyllda spelkonsoler. Medan många är bekanta med att använda en emulator kan tekniken som ligger bakom emulering verka främmande. Inledningskapitlet börjar med en bakgrund som belyser emulering och dess ursprung. Utöver bakgrund tas även projektets huvudsyfte, frågeställningar samt avgränsningar upp.

1.1 Bakgrund

Det har varit svårt att undvika det skifte som skett vad det gäller hur vi, som samhälle, konsumerar digital underhållning under de senaste tio åren. Att hyra eller köpa fysiska kopior i form av kassetter eller DVD-skivor har nästan helt och hållet ersatts av så kallade ”content delivery-systems”, utvecklade och ledda av företag såsom Steam och Netflix. Om du vill spela ett spel eller se på en film är det endast några knapptryck iväg. Till följd av att konsumenter använder olika slags system, läggs det energi och tid på att de digitala spel och filmerna ska vara plattformsoberoende. Detta är positivt för dessa digitala produkters överlevnad. Plattformsoberoende digitala produkter har på grund av dess format en mer generaliserad struktur som tillåter dess exekvering på fler värdsystem. Om något kan konsumeras på flera system och av flera människor är det större chans att en sådan digital produkt lever vidare.

Men vad händer med de produkters vars interna digitala struktur inte tillåter för plattformsoberoende konsumtion? Vad händer när den hårdvara som krävs för att dessa typer av program blir gammal eller sliten? Spelkonsoler såsom Nintendo Entertainment System (NES), som trots sin robusta interna och externa arkitektur, har ett bäst före datum. För varje år som går finns det färre och färre fungerande konsoler att tillhandhålla. Detta har drivit utvecklingen av så kallade emulatorer. Syftet som emulering uppfyller är att kunna exekvera gamla program kompilerade för en viss Instruction Set Architecture (ISA) på annan, oftare modernare, hårdvara. Detta är intressant av flera anledningar. Exempelvis möjliggör emulering användandet av program och spel skapade för spelkonsolen NES, även fast användaren inte har tillgång till originalhårdvaran.

Ordet emulering myntades för första gången år 1963 av IBM [2]. Ordet emulering fick då sin innebörd: att efterliknande en datortyp med en annan datortyp. Samma år skapade Larry Moss och Stewart Tucker den första dokumenterade emulatorn som ett tillägg till IBMs ännu icke släppta dator *System/360*. Emulatorn erbjöd

bakåtkompabilitet med program skrivna för den tidigare generationen IBM-datorer och kunde exekvera IBMs 7070 maskinkod snabbare än originalmaskinen själv [3, p. 185]. Larry Moss och Stewart Tucker var primärt ute efter att skapa ett nytt försäljningsargument för IBMs System/360, vilket också råkade bli startskottet för vad emulering kommit att bli vad det är idag.

Moss och Tuckers emulator var till största del implementerad i hårdvara som en kombination av mikrokod och elektriska ledningar. Detta tillät emulatorn att exekvera program skrivna för IBMs 7070 snabbare än vad originaldatorn klarade av [3]. Idag implementeras istället de flesta emulatorer genom mjukvara. Detta på grund av att hårdvarulösningar är dyra, och kräver speciella kretsar för varje typ av emulator som behöver göras.

En vanlig typ av mjukvaruemulering är interpretering, vilket introducerar mer overhead än vad en hårdvarulösning gör. Emulering genom hårdvara kan vara mer effektiv, genom att använda färre klockcykler på värdprocessorn, än en emulator som använder en mjukvarulösning. Detta innebär att effektiviteten på emulatorer idag minskat jämfört med hos den emulatorn Moss och Tucker skapade.

1.2 Syfte

Projektet ska resultera i två emulatorer, en interpreterande och en dynamiskt omkompilerande (DRC). DRC-emulatorn ska konstrueras med verktyg från Low Level Virtual Machine (LLVM)-projektet och baseras på kompilatortekniken Just-In-Time (JIT). Båda emulatorerna utgår ifrån processorn MOS 6502. Slutligen ska de två emulatorerna undersökas för att avgöra om det finns några påtagliga skillnader i prestanda. Med prestanda åsyftas den tid det tar att exekvera ett specifikt testprogram skrivet för MOS 6502 processorn.

1.3 Frågeställning

Projektets huvudfråga härleds direkt ifrån projektets syfte:

1. Kommer en emulator som utnyttjar JIT med verktyg från LLVM resultera i maskinkod med bättre prestanda jämfört med en helt interpreterande emulator?

Projektets inriktning på specifikt LLVM och dynamisk omkompilering innebär att det finns andra potentiella överväganden skiljt från prestanda. Som en följd av detta har två bifrågor identifierats:

1. Hur väl lämpar sig LLVM för att abstrahera hårdvara?
2. Vilka utmaningar finns vid JIT-kompilering då utgångspunkten är maskinkod snarare än källkod?

1.4 Avgränsningar

De två emulatorer som implementeras kommer endast att omfatta en MOS 6502 processor, snarare än ett helt integrerat system med grafik- och ljudkomponenter såsom NES. Det innebär att inte alla program skrivna för ett system som integrerar en MOS 6502 processor kommer att fungera på emulatorerna. Däremot kommer det vara tillräckligt för att kunna exekvera program som endast utför beräkningar.

Emulatorerna kommer ej att behöva implementera någon mekanism för synkronisering mellan processorn och andra komponenter. Detta leder till att exempelvis No Operation (NOP) instruktionen, som i en riktig MOS 6502 processor tar en klockcykel att utföra, inte kommer bidra med en klockcykel när emulatorerna exekverar maskinkod.

Båda emulatorerna kommer att implementeras i högnivåspråket C++ samt kompileras med samma kompilator och med optimeringsflaggor påslagna [E]. Alla optimeringar som sker på den genererade koden från DRC-emulatorn kommer att skötas av existerande verktyg från LLVM.

Optimeringar i kodgenereringen för DRC-emulatorn kommer inte ske. Detta sker normalt sett redan i programmets källkod, och i detta projekt hade riktlinjer angående kodstrukturering från LLVM behövt följas [4]. Detta är viktigt då olika optimeringar av LLVM Intermediate Representation (IR) kommer att vara olika effektiva beroende på hur den genererade koden ser ut, se teorikapitel 2.3. Varje testprogram skulle därmed behöva analyseras för att avgöra vilken typ av optimering som kodgenereringen bör fokusera på i syfte att uppnå lägsta möjliga exekveringstid. Att studera hur kodgenereringen i LLVM bör ske bedöms vara för tidkrävande för att kunna vara genomförbart inom projektets tidsram.

2

Teori

Syftet med emulering går att förstå utan att veta hur emulatorer fungerar i praktiken. För implementering av en emulator krävs dock, utöver grundläggande datatekniska kunskaper, en djupare förståelse för det emulerade systemet. Teorikapitlet presenterar, samt ger inblick i, centrala begrepp och koncept som är relevanta för förståelsen av de två sorters emulatorer som projektet fokuserar på. Förklaringarna såsom de står är inte tekniskt fullständiga, men de åsyftar att ge underlag att förstå projektets arbetsgång samt de resultat som senare presenteras.

2.1 Emulering

Att kunna återanvända gammal kod skriven för äldre system på nyare och modernare system kan ses som ett attraktivt koncept. Det kan appliceras inom forskningsarbete, industrin och den offentliga sektorn. Detta då gammal kod fortfarande kan visa sig lämplig att lösa samma problem som den en gång var designad för och som fortfarande är relevanta idag. Enligt IBM sker 70% av dagens affärstransaktioner i COBOL [5]. Detta språk utvecklades från början mot stordatorer, något som idag inte hade varit det främsta valet då exempelvis serverkluster kan användas istället [6]. Behovet av emulering blir tydligt från industrins håll, eftersom gammal kod hade kunnat återanvändas på modernare hårdvara istället för att underhålla stordatorer. Dessutom försvinner då behovet av att hitta eller lära upp utvecklare för ett förlegat språk som COBOL.

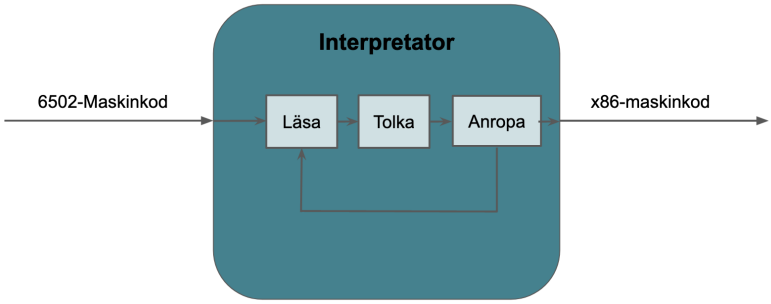
Källkod kan kompileras till en exekverbar binär fil i en form som kallas maskinkod. Varje ISA har unik maskinkod som vid exekvering tolkas av datorns processor. En emulators mål är att genom mjuk- eller hårdvara möjliggöra för ett datorsystem att bete sig som ett annat datorsystem. Kod skriven och kompilerad för en viss ISA kan exekveras på en värd med annan ISA. Detta möjliggörs genom att emulatorn efterliknar de egenskaper som gästen har, vilket leder till att värden kan erbjuda en kompatibel och funktionellt ekvivalent exekveringsmiljö.

Det finns konventionella strukturer som ofta används vid implementering av en emulator. Implementationen är avgörande för hur gästens maskinkod bearbetas och exekveras av emulatorn, respektive lösning medför sina egna för- och nackdelar. De två tillvägagångssätt som projektet undersöker är interpreterande emulering och DRC-emulering.

2.1.1 Interpretering

En rättfram implementation av en emulator utnyttjar interpretering. Interpretering går ut på att exekvera program lagrade i ett binärt format genom att läsa in instruktioner en efter en. En instruktion läses in för att sedan tolkas och det sker en översättning till ett giltigt anrop på värden som resulterar i en ekvivalent förväntad handling. Därefter upprepas denna process av att ”läsa-tolka-anropa” tills det att programmet exekverat färdigt, se figur 2.1. Om inga särskilda optimeringar sker utgör interpretering en precis funktionell motsvarighet till originalhårdvaran. Ytterligare en egenskap, som kommer från att en interpretator stegar igenom källkoden instruktion för instruktion, är att den ofta kan erbjuda bra felmeddelanden. Detta förenklar inte endast implementeringen utav en interpretator, då den blir enklare att felsöka, utan även implementering utav felsökande verktyg såsom debuggers [7, p. 2].

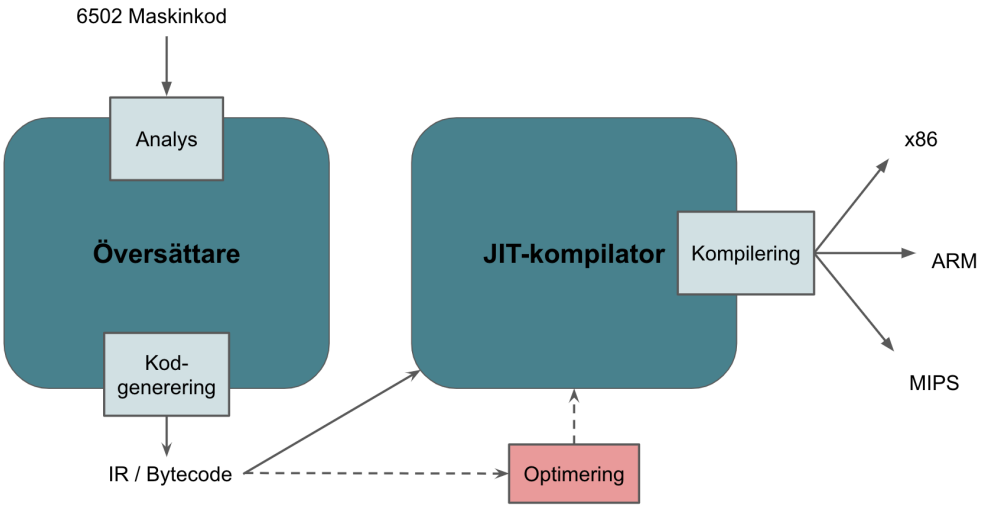
Oavsett hur koden är konstruerad eller hur många upprepningar av likadana anrop som sker tar det lika lång tid att exekvera likadana instruktioner varje gång. I en processor kan en enstaka instruktion påverka många delar av processorns interna tillstånd såsom statusflaggor, stackpekare och minnesinnehåll. Vid interpreterande emulering måste varje sådan förändring ersättas med en instruktion hos värden. Det innebär att en enstaka instruktion, som skulle utförts av gästen, kan kräva flertalet instruktioner hos värden för att genomföra samma händelseförlopp. Den overhead som introduceras av dessa extra anrop i kombination med att översättningen av varje instruktion sker medan programmet exekverar kan leda till påtagliga försämringar i prestanda hos en långsam värd i jämförelse med gästen [8, p. 5] [9].



Figur 2.1: Cykeln ”läsa-tolka-anropa” i en interpretator. Figuren illustrerar en värd med ISA x86. Maskinkod läses in och till slut anropas instruktioner på värden.

2.1.2 DRC

En DRC-emulator är fundamentalt annorlunda från en interpreterande emulator i det att maskinkoden som läses in inte exekveras direkt. Istället läses hela programmet som ska emuleras in samtidigt och kan användas för ytterligare analys. I jämförelse med en interpreterande emulator innehåller en DRC-emulator generellt sett fler komponenter. Dessa komponenter innefattar verktyg för analys, kodgenerering samt kompilering, se figur 2.2.



Figur 2.2: Överblick av en DRC-emulator med JIT-kompilering. Maskinkod läses in och till slut kompileras kod för potentiellt olika ISA såsom x86, ARM och MIPS. Optimering efter kodgenerering är valfritt.

Denna typ av emulering bygger på att programmet dynamiskt omkompileras till helt ny maskinkod. Under exekvering av programmet sker kodgenereringen samt omkompilering mot värdens ISA. Analys av programmet avgör hur kodgenereringen, och i viss utsträckning optimeringar, kommer att gå till. Ett sätt att implementera dynamisk omkompilering är med kompilatortekniken JIT. En JIT-kompilator strävar efter att se mönster i hur koden är konstruerad för att skapa en holistisk bild utav programmet. DRC-emulatorer tenderar att vara mer komplexa i sin implementation än interpreterande emulatorer. De tillåter djupgående programanalys som kan användas för att utföra optimeringar. Som namnet antyder kompileras den kod som ska exekveras, vilket leder till att behovet av tolkning som fanns hos interpreterande emulatorn försvinner [10].

Den första av de tre huvudsakliga komponenterna i DRC-emulatorn är analysdelen. Analys kallas ibland för parsning, vilket innebär att någonting delas upp i mindre bitar för att systematiskt tolkas i syfte att förstå varje dels innebörd [11]. Utmaningen med analysen är att lista ut vad i maskinkoden som är data och vad som är instruktioner, eftersom maskinkoden kan innehålla båda delarna. Detta är ett problem som är specifikt för DRC-emulatorer, men inte JIT-kompilatorer i allmänhet, då DRC-emulatorer arbetar med maskinkod snarare än källkod. I en källkodsfil återfinns ofta extra information, exempelvis subrutiner eller variabelnamn, som kompilatorn kan använda för att avgöra vilken kod som utgör instruktioner respektive data [7, p.363-364]. Den extra informationen gör tolkningen utav källkoden mindre tvetydig, men informationen bevaras ej efter att källkoden kompilerats till maskinkod [7, p.9] [12, p.71-72]. En DRC-emulator måste utföra extra tester och analys av maskinkoden för att korrekt avgöra vilka delar som utgörs av data respektive instruktioner.

Inputen till DRC-emulatorn är likt den interpreterande emulatorn en binär fil bestående av MOS 6502 maskinkod. Denna maskinkod ska sedan analyseras för att skapa en överblick och samla information som krävs för att senare generera kod. Ut-

ifrån specifikation av MOS 6502 ska adressen 0xFFFC innehålla adressen till första instruktionen [13]. Detta faktum ligger till grund när maskinkoden analyseras. Då adressen till den första instruktionen är känd, kan den byten tolkas som en instruktion och inte data.

Instruktionens storlek används för att lokalisera nästa instruktion, då storleken kan adderas till den nuvarande offseten. Denna process upprepas tills det att en hoppinstruktion stöts på. En hoppinstruktion kan vara antingen villkorlig, eller ovillkorlig. En Jump (JMP)-instruktion, vilket är en ovillkorlig hoppinstruktion, kommer alltid att hoppa till måladressen. Denna adress sparas undan och lagras som en del av resultatet av analysdelen. Villkorliga hoppinstruktioner har alltid en fortsättning på byten följt av instruktionen, då villkoret kan vara uppfyllt eller ej. På så sätt är det känt att det som följer en hoppinstruktion också kan tolkas som en instruktion. Eftersom att en JMP-instruktion alltid kommer att hoppa innebär detta att en instruktionsväg avslutas, alltså att den byte som följer en JMP-instruktion inte kan tolkas som en instruktion med hittills insamlad information. För att hitta alla kända instruktionsvägar måste varje måladress analyseras med samma algoritm som är beskriven ovan. Resultatet av algoritmen är en lista av måladresser samt en lista av instruktioner.

Efter analys av maskinkoden utförs kodgenerering, där den analyserade maskinkoden görs om till ett intermediärt format, exempelvis LLVM IR eller Java Bytecode [10]. Syftet med denna konvertering är att den intermediära representationen kan innehålla information som tidigare ej funnits tillgänglig, för att underlätta vidare analys och kompilering. I fallet då maskinkod konverteras till LLVM IR adderas exempelvis typinformation, vilket kan användas för att optimera den intermediära koden. Ytterligare en av fördelarna med ett standardiserat, intermediärt format är att en generell kompilator kan användas. Dessutom blir det enkelt att implementera stöd för ytterligare varianter av maskinkod, då det endast krävs ett program som kan konvertera denna maskinkod till det intermediära formatet.

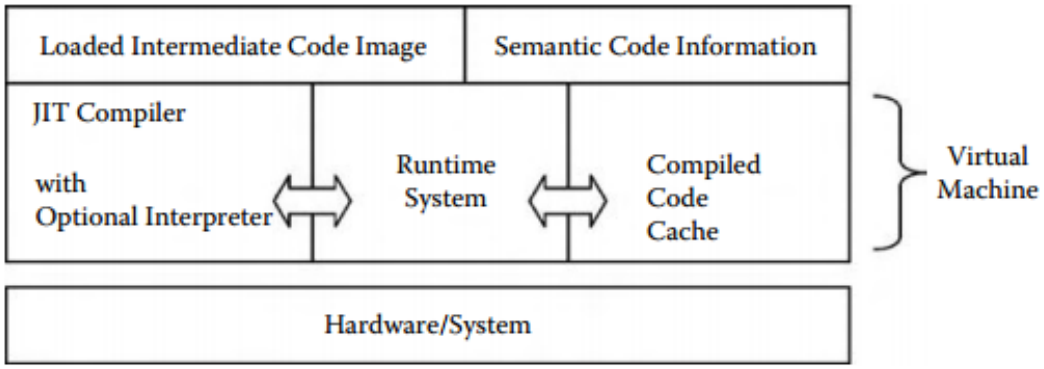
Slutligen behöver denna intermediära kod kompileras för att kunna exekveras. Att utföra kompilering under exekvering kostar förstås processortid hos värden. Det är därför viktigt att resultatet från kompileringen kan sparas undan för senare återanvändning. Det är särskilt viktigt om målet är att summan av kompileringstid och exekveringstid för en DRC-emulator ska vara så liten som möjligt. Program med block av kod som exekverar ofta, med en eller flera specifika rutiner, kan därför förväntas få ut mest nytta av DRC under loppet av hela exekveringstiden. Vid block av kod som exekveras oftare än ett förbestämt tröskelvärde kan detta block omkompileras med aggressivare optimeringar.

2.2 JIT

JIT är en kompilatorteknik som lånar element från både interpretering och statisk kompilering. Det som skiljer en JIT-kompilator från statisk kompilering är att kompilatorn är en del av själva exekveringsmiljön [14], se figur 2.3. Översättningen från

källkod till maskinkod sker därmed under själva exekveringen utav ett program snarare än innan exekveringen påbörjas. En JIT-kompilator har tillgång till information om programmet under exekvering som inte finns tillgänglig vid statisk kompilering. Genom spårning av programmets tillstånd kan en JIT-kompilator fatta relevanta beslut som påverkar kompileringsprocessen.

Beroende på vilka delar av källkoden som faktiskt exekveras kan JIT-kompilatorn välja att kompilera dessa direkt, vid ett senare tillfälle eller inte alls. Samma information används för att avgöra vilka delar av koden som bör optimeras [15]. Att identifiera så kallade *hot spots*, kod som exekveras ofta, är en viktig optimeringsteknik hos JIT-kompilatorer [16, p. 75]. Kodstycken som exekveras oftare sägs vara ”varmare”, vilket innebär att olika kodstycken är olika varma under olika tidpunkter vid exekvering. Hot spots analyseras och optimeras flitigt i välanvända JIT-kompilatorer såsom Java Virtual Machine [17]. En väl fungerande JIT-kompilator hanterar balansen mellan att fortsätta exekveringen och att spendera tid på optimering, då kompileringstid direkt påverkar den totala exekveringstiden. På grund av detta faller vissa JIT-kompilatorer tillbaka på att interpretera de kallaste kodstyckena snarare än att kompilera dem [16, p. 75]. En JIT-kompilator kan även omkompilera delar av programmet flera gånger, vilket är en viktig egenskap för att hantera exempelvis självmodifierande kod. Självmodifierande kod innebär att programmet dynamiskt skapar ny programkod, eller ändrar i befintlig programkod, som ska exekveras.



Figur 2.3: Princip för exekveringsmiljö med JIT [18].

2.3 LLVM

LLVM-projektet är ett kompilator-ramverk som ämnar att förenkla konstruktionen av nya programmeringsspråk genom att introducera en abstraktion mellan högnivåspråk och hårdvaruplattformen som kommer att exekvera koden [19]. Trots sitt namn har LLVM väldigt lite att göra med virtuella maskiner. Det startade som ett forskningsprojekt, men har vuxit till att gå ifrån ett akronym till ett samlingsnamn för en rad olika projekt.

LLVM tillhandahåller bland annat ett plattformsoberoende assemblerspråk som kal-

las LLVM IR, vilket kan skrivas för hand eller genereras från ett högnivåspråk som C++ [20]. Då den intermediära representationen ska kunna översättas till nästan vilken processoruppsättning som helst, behöver den vara agnostisk och därför inte anta något om värdprocessorn. Genom att låta kod skriven i ett högnivåspråk kompilera ned till denna representation kan ett program sedan distribueras på många oberoende ISA. I praktiken möjliggörs detta av att LLVM har separerat generering av maskinkod från generering av LLVM IR, samt att LLVM tillhandahåller verktyg för att generera exekverbar maskinkod åt många ISA [21].

LLVM-projektet innehåller dessutom en rad verktyg som automatiskt kan utföra optimeringar på befintlig LLVM IR-kod, däribland ihopslagning av liknande funktioner, eliminering av onödig kod och förenklingar av slingor [22]. Dessa verktyg kan styckvis länkas in som bibliotek i kompilatorn, och de kan användas för att optimera all programkod uttryckt i LLVM IR [23]. Hur LLVM IR-koden är upplagd är avgörande för hur mottaglig den är för olika typer av optimeringar, vilket innebär att själva genereringen utav LLVM IR-koden spelar roll för den färdigt optimerade kodens prestanda [4]. Optimeringarna sker som ett steg i kompileringsfasen, och samma optimering kan utföras en eller flera gånger. Det är upp till kompilatorn att avgöra när den resulterande koden är tillräckligt optimerad.

2.4 MOS 6502

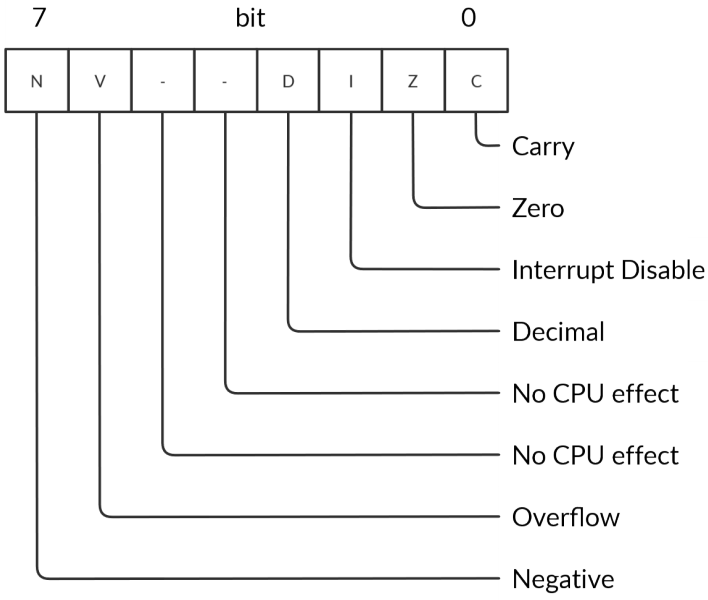
Gästen som projektet baseras på heter MOS 6502. Det är en klassisk processor som använts i flertalet konsoler och datorer. En version av den används bland annat i den klassiska spelkonsolen NES [8]. Sedan lanseringen i Japan 1983 har den MOS 6502-baserade spelkonsolen byggt upp en trogen skara entusiaster som av fri vilja sammanställt omfattande dokumentation kring konsolen som sedan publicerats fritt på internet [24]. Wikin NESDEV är en samlingsplats specifik för NES-konsolen, och den inkluderar djupgående information om MOS 6502 och hur dess register, adressering, minne samt instruktioner fungerar [24]. Intresset för konsolen bland tekniskt kunniga har lett till att det finns många implementerade emulatorer tillgängliga, exempelvis FCEUX, JNES och NESTOPIA [25] [26] [27] . En mångfald av test-program som kan säkerställa att en emulator av processorn är korrekt implementerad finns också att tillgå. Dessutom är MOS 6502 en relativt simpel processor. Den hanterar endast 8-bitars tal, och har väldigt få register. Den är simpel jämfört med modernare processorer som har större ISA, men är även simpel jämfört med processorer från samma era. Exempelvis använder Z80, en Complex Instruction Set Computer (CISC), upp till sex cykler per instruktion, medan MOS 6502 använder högst fyra [28]. Det gör den till en lämplig processor för att undersöka om LLVM kan användas för att implementera en DRC-emulator i syfte att uppnå en effektivare emulator än vad en interpreterande emulator skulle kunna åstadkomma [8] [29, p. 1].

För att korrekt implementera en emulator krävs insikt i hur den underliggande hårdvaran är implementerad. Det finns både interna samt perifera delar som är nödvändiga för att konstruera en fungerande MOS 6502. MOS 6502 är en 8-bitars processor, med en 16-bitars adressbuss [13]. Det behövs alltså en 16-bitars adress-

rymd där programmets instruktioner och data kan sparas och modifieras. För att processorn ska kunna arbeta med information ifrån minne finns det olika register som kan lagra data för olika ändamål samt instruktioner som kan arbeta på datan. Det finns även olika sätt att hämta in data från minnet beroende på ändamål.

2.4.1 Register

MOS 6502 har sex register bestående av en ackumulator, två indexregister, en programräknare, en stackpekare, och ett statusregister [13]. Ackumulatorregistret A är ett åtta bitars dataregister som används främst för att ladda och spara operationer, samt är det enda registret som aritmetiska operationer kan utföras på. Indexregistren, X och Y, är åtta bitar och används för att spara undan resultatet från ackumulatorn, eller för indexering av minnet i exempelvis slingor. Programräknare (PC) är 16-bitar stort och används för att hålla koll på vart programmet befinner sig. Stack Pekare (SP) är ett åtta bitars register som är en pekare till minnesadressen för det översta elementet i stacken. Statusregistret är ett åtta bitars register som innehåller information om resultatet från de operationer som har utförts, se figur 2.4. I detta registret representerar varje enskild bit specifik information om resultatet från varje utförd instruktion [A].



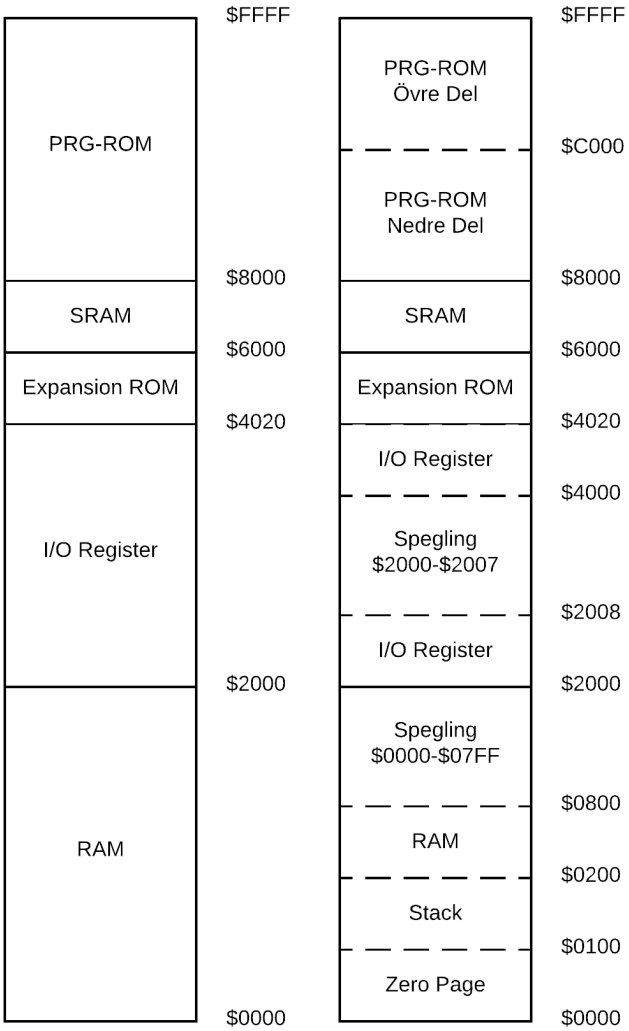
Figur 2.4: Illustrering för statusregistret för MOS 6502.

2.4.2 Adressering

Det finns tretton olika adresseringslägen för MOS 6502, med tre olika adresseringssätt [13]: direkta, indirekta och absoluta [B]. En instruktion har oftast möjlighet att använda flera olika adresseringssätt, A registret kan exempelvis laddas med hjälp av åtta olika adresseringslägen.

2.4.3 Minne och adressrymd

MOS 6502 har en 16-bitars bred adressbuss, vilket innebär en teoretisk övre gräns på 64 KB av fysiskt lagringsminne som processorn klarar av att adressera [30]. Med hexadecimal notation kan den totala minnesrymden skrivas som *$0000 - $FFFF*. Varje adress rymmer 8 bitar, vilket innebär att ett 16-bitars tal lagras på två på varandra följande minnesadresser. I minnet är data organiserat enligt Little Endian [8]. Den minst signifikanta byten är alltså lagrad före den mest signifikanta byten i ett 16-bitars tal.



Figur 2.5: Illustrering av NES minne. Regionerna Zero Page och Stack är gemensam för samtliga system med MOS 6502.

Olika system som integrerar MOS 6502 kan välja att segmentera det logiska minnet efter behov, men det finns ett par regioner utav minnet som är hårdkodade i MOS 6502s design. Däribland stacken, som består av 256 adresser och alltid återfinns på plats *$0100 - $01FF* [30].

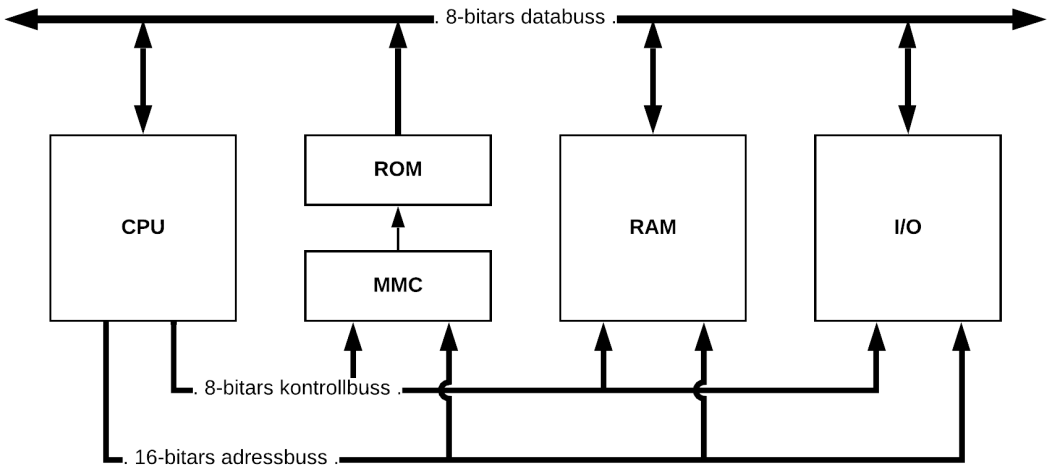
Till sist är de 6 sista bytesen, *$FFFA - $FFFF*, reserverade för avbrottshantering samt resetvektorn. Se figur 2.5 för en visualisering av NES minne. Andra viktiga minnessregioner inkluderar Zero Page samt den del av minnet där programmet lagras. Med Zero Page menas de 256 första adresserna (*$00-$FF*) och kan adresseras med endast 8 bitar. Detta var lämpligt ur prestandaskäl på den tiden processorn skapades. Genom att adressera med 8 bitar istället för 16 bitar kunde man få plats med fler instruktioner i instruktionsminnet. Idag, när vi har jämförelsevis obegränsat med minne är detta inte relevant. Dessutom kräver Zero Page adressering bara en läsning från minnet, istället för två. Detta för att det krävs en läsning per 8 bitars-tal.

Det görs ingen direkt separering av data och instruktioner i minnesrymden. Information som lagras på adresser som endast är läsbara kan ej ändras under exekvering, men vid behov kan programmerare välja att exekvera instruktioner från andra delar av minnet som dessutom är skrivbara. Detta är möjligt genom att hela minnet är direkt adresserbart, vilket öppnar upp för självmodifierande kod. Självmodifierande kod är en specifik typ av optimeringsmetod som skriver över minnesadresser som vanligtvis tolkas som instruktioner. Denna teknik kan användas för att kringgå begränsningar i datorns minnesstorlek. Ett exempel, från en NES, på hur minnet kan se ut i sin helhet finns i figur 2.5.

2.4.4 Instruktioner

MOS 6502 är en 8-bitars processor, trots att den kan adressera en 16-bitars adressrymd. Det beror på att databussen endast är 8 bitar medan adressbussen är 16 bitar, se figur 2.6. Databussens begränsade bredd innebär att processorn endast kan utföra beräkningar på tal som är 8 bitar stora åt gången [8]. Aritmetik med 16-bitars tal tar därför fler klockcykler att utföra, eftersom de måste utföras med hjälp av subrutiner i ett program. MOS 6502 har 56 instruktioner med totalt 151 varianter, alla med unika opcodes, då alla adresseringslägen räknas in.

Varje instruktion representeras som en byte i minnet. Varje instruktion följs av antingen noll, en eller två bytes till som innehåller relevant information för instruktionen beroende på adresseringsläge. Detta är ekvivalent med ett argument till en funktion eller metod i ett högnivåspråk. Detta kan vara ett numeriskt värde eller en minnesadress angett i bas 16. Hur argumentet tolkas beror på instruktionens adresseringsläge, vilket tolkas utifrån opcoden.



Figur 2.6: Illustrering av MOS 6502: dess komponenter och kommunikationsbussar

3

Metod

Projektet delades in i tre faser: skapandet av en interpreterande emulator, skapandet av en DRC-emulator med hjälp av LLVM samt testning utav dessa. Metodkapitlet beskriver implementeringen av de två emulatorerna, en debugger, en assemblator samt applicerad testmetodologi. Val av verktyg och tillvägagångssätt motiveras kontinuerligt.

3.1 Interpreterande emulator

I den första fasen implementerades den interpreterande emulatorn. Som nämns i teorikapitlet är det kritiskt att denna efterliknar den riktiga processorn så nära som möjligt. Detta uppfylldes genom att låta ett internt tillstånd i den interpreterande emulatorn representera motsvarande tillstånd i den fysiska gästprocessorn. Varje register representerades av varsin variabel i emulatorn, där storleken på varje variabel följer specifikationen för MOS 6502. Exempelvis representerades alla 8-bitars register med typen *int8_t* och PC med typen *int16_t*.

Den interpreterande emulatorn tillhandahåller funktionalitet för att läsa in, tolka och exekvera maskinkod. Vid exekvering av emulatorn läses först instruktioner från en binärfil in och tolkas. Efter att en opcode lästs in anropas motsvarande funktion ur en lista. I listan finns funktioner som utför samma förändringar på det interna tillståndet som om instruktionen utförts på processorn som modelleras. Eftersom de flesta instruktioner i MOS 6502 kan använda olika typer av adresseringslägen abstraherades dessa ut till separata funktioner. Funktionerna i listan startar därför med att anropa korrekt adresseringslägesfunktion för inhämtning av data ifrån emulatorns virtuella minne. Denna abstraktion minskade kodduplicering eftersom adresseringsläget alltid utför samma handling. Emulatorn blev därmed enklare att felsöka.

Viss funktionalitet i emulatorn modulariserades. Läsning och skrivning till minnesbussen implementerades som två funktionspekare istället för den faktiska funktionaliteten. Detta förenklar för användaren vid brukning av emulatorn som en del av ett större system där en MOS 6502-emulator behövs, men där minnet behöver delas med andra komponenter.

3.1.1 Debugger

För att underlätta utvecklingen av den interpreterande emulatorn skapades en debugger. Denna debugger är ett grafiskt verktyg som tillåter användaren att se emulatorns interna tillstånd, samt sätta brytpunkter i koden som exekveras. Det finns även funktionalitet för att fritt ändra i emulatorns RAM-minne. Debuggern implementerades med hjälp utav ramverket IMGUI [31]. Ifall behov av att återanvända IMGUI för flera grafiska verktyg skulle uppstå i projektet skapades en wrapper för ramverket.

3.2 DRC-emulator

Med vad som nämns i teorikapitel 2.1.2, utgörs DRC-emulatorn av tre komponenter bestående av analys, kodgenerering och kompilering. Likt den interpreterande emulatorn finns ett internt tillstånd som modellerar en MOS 6502. Detta tillstånd modifieras, precis som i den interpreterande emulatorn, under exekvering.

Under analysen använder DRC-emulatorn, likt den interpreterande emulatorn, opcoder som läses in från binärfilen. Skillnaden är att den interpreterande emulatorn tolkar opcoderna för exekvering, medan DRC-emulatorn endast använder dem för att hämta information om programmet. Information om instruktionens storlek, namn och huruvida det är en hoppinstruktion eller ej tillgängliggörs. Algoritmen som beskrivs i teorikapitel 2.1.2, användes för att producera analysdelens utdata bestående av två listor: en med måladresser och en med instruktioner. Instruktionerna lagras med avseende på dess offset i binärfilen i stigande ordning.

Indata för kodgenereringen tas från utdatan av den föregående analysen, alltså informationen om maskinkoden. För att skapa ett kontrollflöde instansierar programmet BasicBlock, vilket är en typ försedd av LLVM för att representera block av kod. Dessa block fylls senare med LLVM IR-kod som motsvarar MOS 6502-instruktioner. Den första instruktionen i blocket är en plats i koden dit en hoppinstruktion någon gång kommer att hoppa. BasicBlocken avslutas alltid med en kontrollflödesinstruktion, och kommer därmed hoppa till ett annat BasicBlock. Undantaget är en terminatorinstruktion som terminerar programmet och returnerar en statuskod.

BasicBlocken är till en början tomma, den enda tillgängliga informationen för dem är vilken offset de startar på. Detta kan liknas med labels i assembler. Därefter kan kodgenerering för varje block ske genom att iterera över listan med instruktioner, instruktionens opcode används för att avgöra vilken LLVM IR-kod som ska genereras. LLVM IR-koden beskriver förändringen som ska ske på det interna tillståndet. Processen att använda opcoder från maskinkoden kan liknas med den som används i den interpreterande emulatorn. Skillnaden är att den interpreterande emulatorn utför en förändring på det interna tillståndet, medan DRC-emulatorn genererar kod som beskriver en förändring som kommer ske under exekvering. När en instruktion stöts på som också är ett mål för en hoppinstruktion, så kallas en funktion som berättar för LLVM var koden fortsättningsvis ska genereras. Resultatet av detta

steget blir en LLVM-modul som beskriver hela programmet.

Innan kompileringen av LLVM-IR koden sker finns möjlighet för optimering. Detta sker genom något som LLVM kallar för optimization passes [22]. De passes som används är tagna från ett projekt där LLVM används för att skapa en dynamiskt omkompilerande emulator av MIPS [F] [32]. Dessutom används en del av koden från detta projekt för att introducera JIT funktionaliteten i DRC-emulatorn.

Kompilering av den intermediära koden, som beskrivet i teoridelen, automatiserades med hjälp av LLVM. Kompileringen utförs endast en gång under exekvering, strax efter den ursprungliga analys som sker när programmet laddas in i DRC-emulatorns minne.

3.2.1 Indirekta hopp

En av instruktionerna som förhindrar möjligheten till statisk omkompilering är hoppinstruktionen JMP Indirect. Till skillnad från JMP Immediate, är måladressen okänd innan exekvering av programmet. JMP Indirect fungerar genom att titta i minnet på en angiven adress och hämta måladdressen därifrån. Den angivna adressen är dock en del av instruktionen, och blir känd vid exekvering.

Lösningen på denna utmaning var att implementera en hopptabell som kontrollflödet kan hoppa till för att slå upp det block av kod som ska exekveras. Denna lösningen är inte fullständig då den inte hanterar alla möjliga scenarier. Bristen med implementationen är att blocket som måladressen är associerad med måste vara känd vid kompileringstid. Kravet för att blocket ska vara känt innebär att det måste finnas en giltig instruktionsväg dit redan innan exekvering, alltså ett villkorligt eller ovillkorligt hopp dit. Då endast en kompilering sker under exekvering är det möjligt för hopptabellen att vara ofullständig, särskilt för ett större program, om någon väg ej var känd under detta kompileringstillfälle.

3.3 Testning

I projektets sista fas genomfördes olika tester på emulatorerna. För den interpreterande emulatorn genomfördes ett omfattande verifieringstest för processorfunktionaliteten. För DRC-emulatorn användes en testsvit bestående av flera mindre test, inspirerade av det som användes för den interpreterande emulatorn. Emulatorernas tester skapades för att säkerställa att de fungerade enligt specifikation för MOS 6502, det vill säga att de fungerade som en MOS 6502 skulle göra. Slutligen genomfördes tre olika program för prestandatestning på emulatorerna. Dessa ligger till grund för resultat och diskussion.

3.3.1 Testmetod för verifiering

För testning av den interpreterande emulatorn användes ett test skrivet av Klaus Dormann [33]. Testet säkerställer att varenda instruktion ändrar det interna till-

ståndet exakt enligt specifikation, och består av hela 6000 rader assemblerkod. Om implementationen av emulatorn har brister visas detta för användaren genom att testprogrammet fastnar i en oändlig slinga specifik för det misslyckade testet. Därefter kan användaren inspektera källkoden på den rad där testet fastnade och därmed ta reda på vilken instruktion som är felimplementerad. Kommentarer och variabelnamn i källkoden tillåter användaren att förstå vad problemet bör vara. När emulatorn itererat igenom hela testet hamnar PC-värdet på en bestämd adress som indikerar att testet har lyckats och att emulatorn är korrekt implementerad.

Då DRC-emulatorn inte använder PC-registret var det ej möjligt att använda Dormanns test. Istället utvecklades enskilda tester för varje instruktion. För att testa DRC-emulatorns funktionalitet utnyttjades den interpreterande emulatorns korrekthet som referens. Testprogrammen som skapades var strukturerade för att förenkla verifiering. Ett lyckat test returnerar statuskod 0, medan ett misslyckat test returnerar statuskod 1. Detta gör att testerna kan skrivas på ett väldefinierat sätt. Varje testfil testade en specifik instruktion och fungerar genom att testa flaggor och register, programmet sparade undan resultatet från de olika registren efter varje utförd instruktion. Genom att därefter exekvera samma testprogram på den interpreterande emulatorn erhölls ett förväntat resultat. Detta resultat sparades därefter undan och lades in i testprogrammets källkod. Varje test kan därefter var för sig verifiera att resultatet blivit som förväntat. När testet därefter exekverades på DRC-emulatorn kunde eventuella fel i implementationen upptäckas.

3.3.2 Assemblator

Till en början skrevs de funktionella testerna i maskinkod. För att möjliggöra snabbare implementering av nya testprogram för MOS 6502 implementerades en assemblator. Assemblatorn tillät testprogrammen att skrivas i assembler snarare än maskinkod. Assembler innehåller mnemonics och labels som underlättar utvecklingen av program. Det antogs också att assembler skulle leda till mindre felbenägna program, och mindre tid skulle därför behöva spenderas på felsökning av de egenskrivna testprogrammen. Då varje assemblator på marknaden är unik, betyder det att en inlärningstid måste avläggas för att göra sig bekant med en specifik produkt. Projektets ändamål krävde inte avancerade funktioner, utan större vikt lades vid att mnemonics följde specifikationen för MOS 6502. En egenutvecklad assemblator kan designas för att exakt möta detta behov. Utöver detta kan också utökningar till assembler implementeras vid behov, vilket utnyttjades till att implementera stöd för ofta använda hjälpfunktioner. Exempelvis lades ett kortkommando till för att skriva ut emulatorernas olika register.

3.3.3 Metod för prestandatest

När båda emulatorerna var verifierade och färdigställda exekverades tre program: Bubble sort [D.1], Fibonacci [D.2] och Sieve of Eratosthenes [D.3] för att mäta prestandan. I detta fallet åsyftar prestanda exekveringstid, som mättes genom att använda systemklockan. Dessa program varierar i algoritmisk komplexitet, se re-

spektive bilaga. Eftersom DRC-emulatorn består av fler steg än enbart exekvering så mättes även tidsåtgången för komponenterna som utför parsning, kodgenerering samt kompilering med eventuell optimering av den genererade koden.

Mätningarna utfördes i två omgångar för DRC-emulatorn, både med och utan optimering av den genererade LLVM IR-koden. Om LLVM IR-koden optimeras påverkas både kompilerings- och exekveringstiden. För att minimera påverkan från enskilda avvikande resultat, genomfördes varje test tio gånger. Därefter noterades medelvärdet för varje uppmätt resultat för respektive emulator och konfiguration. Samtliga mätningar utfördes på två datorer av varierande processorkvalitet [E]. Alla tester exekverades på båda datorerna, med syfte att testa prestanda skillnader på två olika värdarkitekturerna med olika ISA. Ena värden är betydligt långsammare i jämförelse med den andra.

4

Resultat

Resultaten presenteras genom tre underrubriker, för att senare kunna besvara huvudfrågan i diskussion. Varje underrubrik redovisar insamlad data från varje enskilt prestandatest. Kompileringstiden i tabellerna inkluderar även tiden för optimeringen.

4.1 Bubble sort

Det första programmet som användes för att generera information kring exekveringseffektivitet var Bubble sort [D.1]. Detta program har den största algoritmiska komplexiteten av samtliga prestandatest, och exekverades med 2^{13} element att sortera. Millisekunder användes som största måttenhet för att meningsfullt jämföra resultaten mellan emulatorerna på den kraftfullare datorn. Mikrosekunder uteslöts då exekveringstiden översteg 100 000 mikrosekunder. Resultaten som presenteras, i både figur 4.1 och figur 4.2, visar att DRC-emulatorn presterar bättre än den interpreterande emulatorn.

Exekvering på den kraftfullare datorn [E.1] visar att DRC **utan** optimering är cirka 9 gånger snabbare än den interpreterande emulatorn, medan DRC **med** optimering är cirka 10 gånger snabbare. Tiden uppmätt för kompilering respektive exekvering för DRC med optimering understiger motsvarande uppmätta värden för DRC utan optimering. Kompileringen slutfördes cirka 37% procent snabbare, medan exekveringen slutfördes cirka 12% procent snabbare.

Exekvering på den svagare datorn [E.2] visar en större absolut tidsskillnad mellan de två emulatorerna, medan den procentuella skillnaden är mindre jämfört med exekveringen på kraftfullare datorn. Relativt den interpreterande emulatorn exekverar DRC **utan** optimering cirka 5,5 gånger snabbare, medan DRC **med** optimering exekverar cirka 7 gånger snabbare. I relation till exekveringstiden spenderar även den svagare datorn mindre tid på kompilering och eventuell optimering, jämfört med den kraftfullare datorn. Den svagare datorn spenderade cirka 0,5% av den totala tiden på att utföra kompilering, både med och utan optimering, medan den kraftfullare datorn spenderade cirka 0.9% - 1.2% av den totala tiden för samma ändamål.

Parsningen utav programmet visar inte någon mätbar skillnad mellan varken de två datorerna eller DRC-konfigurationerna när tiden mäts i millisekunder. Relativt de andra faserna, kompilering och exekvering, är tidsåtgången för parsning försumbar.

	Interpreterade	**DRC**	**DRC** (Optimerad)
Parsning	-	34 μs	41 μs
Kompilering	-	11 ms	8 ms
Exekvering	8843 ms	973 ms	872 ms
Total tid	**8843 ms**	**984 ms**	**880 ms**

Figur 4.1: Resultat från Bubble sort på den kraftfullare datorn [E.1].

	Interpreterade	**DRC**	**DRC** (Optimerad)
Parsning	-	222 μs	305 μs
Kompilering	-	57 ms	46 ms
Exekvering	65560 ms	11918 ms	9273 ms
Total tid	**65560 ms**	**11975 ms**	**9320 ms**

Figur 4.2: Resultat från Bubble sort på den svagare datorn [E.2].

4.2 Fibonacci

Det andra programmet som användes för att generera information kring exekveringseffektivitet var Fibonacci [D.2]. Programmet beräknade det 24:e talet i Fibonacciserien, vilket är 46368. Det är det största tal i serien som ryms i ett 16-bitars tal. Resultaten som presenteras, i både figur 4.3 och figur 4.4, visar att den interpreterande emulatorn presterar bättre än DRC-emulatorn.

Den kraftfullare datorn visar att DRC **utan** optimering är cirka dubbelt så snabb den interpreterande emulatorn, och det samma gäller för DRC **med** optimering. Skillnaden i tid uppmätt för kompilering respektive exekvering mellan DRC med optimering och DRC utan optimering är tillräckligt liten att den kan försummas.

Exekvering på den svagare datorn visar, likt föregående test, en större absolut tidsskillnad mellan de två emulatorerna. Relativt den interpreterande emulatorn exekverar DRC **utan** optimering exakt 12,8 gånger snabbare, medan DRC **med** optimering exekverar cirka 21,3 gånger snabbare. I relation till exekveringstiden spenderar den svagare datorn mer tid på kompilering och eventuell optimering, jämfört med den kraftfullare datorn. Båda datorerna spenderade cirka 99,7% av den totala tiden på att utföra kompilering plus eventuell optimering.

När mätningarna sker mikrosekunder blir skillnaderna i parsning mellan de två datorerna samt DRC-konfigurationerna plötsligt jämförbara. Relativt kompileringsfasen är tidsåtgången för parsning återigen försumbar, medan det konsekvent tar längre tid att parsa än att exekvera programmet.

	Interpreterade	**DRC**	**DRC** (Optimerad)
Parsning	-	15 μs	13 μs
Kompilering	-	5205 μs	5215 μs
Exekvering	2 μs	< 1 μs	< 1 μs
Total tid	**2** μs	**5220** μs	**5229** μs

Figur 4.3: Resultat från Fibonacci på den kraftfullare datorn [E.1].

	Interpreterade	**DRC**	**DRC** (Optimerad)
Parsning	-	98 μs	97 μs
Kompilering	-	41524 μs	32879 μs
Exekvering	64 μs	5 μs	3 μs
Total tid	**64** μs	**41627** μs	**32979** μs

Figur 4.4: Resultat från Fibonacci på den svagare datorn [E.2].

4.3 Sieve of Eratosthenes

Det tredje programmet som användes för att generera information kring exekveringseffektivitet var Sieve of Eratosthenes [D.3]. Den övre gränsen sattes till 208, vilket begränsades av en bugg i implementationen utav testprogrammet. Beräkningarna utförs korrekt, men om den teoretiska maxgränsen 255 används istället så avslutas programmet direkt. Programmet gav ett resultat likt det som genererats med hjälp Fibonacci testet. Som dokumenterat i tabell 4.5 och 4.6 är den interpreterande emulatorn markant snabbare än DRC, sett till totaltid. Däremot har DRC-emulatorn likt tidigare tester en kortare exekveringstid på de båda datorerna.

Den kraftfullare datorn exekverar DRC **utan** optimering cirka 18 gånger snabbare än den interpreterande emulatorn, medan DRC **med** optimering är cirka 28 gånger snabbare. Även tiden som uppmättes för kompilering respektive exekvering för

DRC med optimering understiger motsvarande uppmätta värden för DRC utan optimering. Kompileringen slutfördes cirka 12% procent snabbare, medan exekveringen slutfördes cirka 50% procent snabbare. Det existerar en marginell skillnad i tiden det tar för parsning av programmet att utföras, men procentuellt är den icke-optimerade DRC-emulatorn cirka 10% snabbare.

Likt resultatet för både Bubble sort och Fibonacci uppvisar den svagare datorn en mätbart lägre exekveringstid när DRC-emulatorn används jämfört med den interpreterande emulatorn. Till skillnad från Bubble sort, men i linje med Fibonacci, spenderas överlägset mest tid på kompileringsfasen. För både DRC med och utan optimering gick cirka 99,6% av tiden åt till kompilering och eventuell optimering. Endast **kompilering** för DRC utan optimering tog cirka 144,1 gånger så lång tid jämfört med den interpreterande emulatorns **exekveringstid**. För DRC med optimering var motsvarande faktor 122,9.

När kompileringen väl var färdig exekverade DRC utan optimering cirka 10,4 gånger snabbare än den interpreterande emulatorn. Motsvarande faktor för DRC med optimeringar var cirka 26,4.

	Interpreterade	**DRC**	**DRC** (Optimerad)
Parsning	-	23 μs	25 μs
Kompilering	-	8880 μs	7942 μs
Exekvering	55 μs	3 μs	2 μs
Total tid	**55** μs	**8906** μs	**7969** μs

Figur 4.5: Resultat från Sieve of Eratosthenes på den kraftfullare datorn [E.1].

	Interpreterade	**DRC**	**DRC** (Optimerad)
Parsning	-	170 μs	165 μs
Kompilering	-	49586 μs	42086 μs
Exekvering	344 μs	33 μs	13 μs
Total tid	**344** μs	**49789** μs	**42264** μs

Figur 4.6: Resultat från Sieve of Eratosthenes på den svagare datorn [E.2].

5
Diskussion

I diskussionskapitlet presenteras en djupare analys av resultatet, emulatorerna, de verktygen som användes och svårigheterna som uppkom under projektet. Kapitlet innehåller även etisk reflektion över potentiella effekter på samhället, samt generella tankar kring förekomst av tidigare publicerat material kring dynamisk omkompilering.

5.1 Analys av resultat

Anmärkningsvärt i testresultatet är att bara ett utav de tre prestandatesterna, Bubblesort, har totaltider som är lägre för DRC-emulatorn än den interpreterande emulatorn. I de andra två prestandatesterna, Fibonacci och Sieve of Eratosthenes, är den interpreterade emulatorns totaltid markant lägre. I dessa två specifika fall kan man argumentera för att det är onödigt att använda DRC-emulatorn. Problemen som uppkommer med att dra denna slutsatsen, är att program som används i verkligheten sällan består av enskilda algoritmer som exekveras en enstaka gång. Dessutom är de sällan så små och korta som våra test.

Dessa prestandatest valdes på grund av att svårigheten att implementera dem passade med projektets tidsram. Om mer avancerade prestandatest hade valts fanns risken att de inte skulle hunnits färdigställas.

Utöver detta finns det främst två anledningar till varför vi inte valt att använda större program för testerna. Den första anledningen är att DRC-emulatorn, i sin nuvarande form, inte kan exekvera alla program, eftersom den inte kan exekvera självmodifierande kod [5.3.3]. På grund av detta minskades utbudet av kandiderade program för testning. Den andra anledningen är att de flesta program som används inte är ändliga, vilket leder till att programmet måste stoppas manuellt. Dessa program består ofta av flera komponenter, såsom ett GUI och möjlighet att hantera input. Att implementera detta låg utanför projektets omfattning. Med denna typ av tester blir det tydligt att program som är små och har kort förväntad exekveringstid utförs snabbare utav den interpreterande emulatorn. Då den interpreterande emulatorn ej utför någon kompilering eller parsning tillåts den i specifika fall att vara mer effektiv än DRC-emulatorn.

Värt att notera är även att ett externt program hade kunnat används till att mäta tidsåtgången istället för att använda systemklockan. Ett sådant program, som hade

givit ett mer exakt och rättvist resultat, hade varit prestandaanalysverktyget *perf*. Perf kan mäta använd processortid, vilket är ett bättre mått än passerad tid. Motivering till att använda perf, istället för vår metod, är att perf har möjlighet att räkna det exakta antalet klockcykler som utförts under exekveringen av program. Det intressanta med att kunna mäta antalet klockcykler är att dem direkt svarar mot de instruktioner som utförs av processorn. Passerad tid kan variera beroende på andra systemprocesser som exekveras parallellt med emulatorn. Av denna anledning är antal klockcykler ett mer tillförlitligt mått på prestanda än passerad tid. Anledningen till att projektet inte använder sig av ett verktyg som perf är för att detta hade krävt att DRC-emulatorn delades upp till tre separata program som exekverar de tre delarna för sig.

Med optimerad LLVM IR-kod blev både kompileringstiden och exekveringstiden kortare för DRC-emulatorn. Det senare var väntat, då optimering av den genererade koden bör leda till, åtminstone, lika effektiv maskinkod som utan optimering. Att kompileringen tog kortare tid för DRC-emulatorn **med optimering** var dock oväntat. I kompileringen sker eventuella optimeringar som ett separat steg efter kodgenereringen, men innan kompilering från LLVM IR till maskinkod. Anledningen till att kompileringen sker snabbare när optimeringar också utförs tros ligga i att den resulterande LLVM IR-koden blir betydligt kortare efter att den är optimerad. Det finns då mindre arbete att utföra under kompileringen från LLVM IR till exekverbar maskinkod. Till följd av det oväntade resultatet, och den förmodade hypotesen, inspekterades den icke-optimerade samt den optimerade maskinkoden för alla prestandatest som använts för att producera projektets resultat. Det visade sig att den optimerade LLVM IR-koden i genomsnitt endast var 41,3% så lång som den icke-optimerade.

Det går även att härleda från resultatet att man tjänar mer på att använda DRC-emulatorn på den svagare datorn, kontra den kraftfullare datorn. I de flesta fall verkar exekveringstiden för både den interpreterande emulatorn och DRC-emulatorn öka markant när en svagare processor används, medan kompileringstiden inte verkar öka lika mycket. Det är svårt att veta exakt varför kompileringstiden inte påverkas, trots att den ena datorn har en lägre klockhastighet. Utöver klockhastigheten är skillnaden mellan datorerna deras ISA, och det är möjligt att ARM är effektivare på att kompilera LLVM IR än vad x86 är.

5.2 Emulatorer

I denna sektion diskuteras implementeringen utav emulatorerna, relaterade utmaningar samt för- och nackdelar med respektive emulator.

5.2.1 Interpreterande emulator

Genom verifiering med hjälp av Klaus Dormanns test kan vi dra slutsatsen att den interpreterande emulatorn korrekt efterliknar MOS 6502 processorn. Testet är så pass omfattande att emulatorn kunde anses som korrekt. Testet mäter huruvida

emulatorns interna tillstånd förändras på samma sätt som den ursprungliga processorn hade gjort efter en given instruktion. Det existerar liknande test för andra processorer, exempelvis Z80 [34]. Om en emulator för en processor som saknar ett sådant test utvecklas behövs förstås en liknande testsvit skrivas, vilket kommer öka tiden för utveckling markant.

Exakt beräkning av cykler eller modellering på transistornivå sker ej. Detta medför att emulatorn inte är identiskt till MOS 6502 vilket heller inte var syftet med vår implementation. Detta är inte heller ett krav för att kunna exekvera majoriteten av program skrivna för MOS 6502. Den interpreterade emulatorn tar heller inte hänsyn till synkronisering med andra komponenter vilket originalprocessorn gör.

Syftet med skapandet av den interpreterande emulatorn var att jämföra hur en JIT-kompilator skiljer sig mot denna och inte att uppfylla funktionen med kommunikation med andra komponenter. Exempel på komponenter som skulle kunna kopplas till emulatorn är komponenter för hantering av grafik och ljud. För att exempelvis en NES ska kunna fungera utan att användaren ska märka av några störningar behöver dessa komponenter kommunicera. Om denna kommunikation inte sker synkroniserat, kan användaren se att spelet inte flyter på som förväntat eller att spelet har grafiska artefakter.

5.2.2 Dynamiskt omkompilerande emulator

Genom att utnyttja den interpreterande emulatorns korrekthet kan gruppen med stor säkerhet påstå att de instruktioner som hittills implementerats följer specifikationen för MOS 6502 på ett trovärdigt sätt.

Som noterats i alla resultat som presenterats i resultatkapitlet spenderar DRC-emulatorn mindre tid i kompileringsfasen då optimering är aktiverat kontra avaktiverat. Det enda undantaget för denna observation är kompileringen på den snabbare datorn för Fibonacci-testet, se tabell 4.3. Då optimering kräver ett extra steg är det inte garanterat att den totala tiden blir lägre med optimering påslaget. I fallet med fibonacci reduceras inte prorgammet tillräckligt mycket för att det ska bli en tidsvinst i kompileringsfasen.

Att DRC-emulatorn med optimeringar spenderade mindre tid i kompileringsfasen tycktes initialt vara motsägelsefullt. DRC-emulatorn måste spendera extra tid på att utföra optimeringar, då detta är ett helt separat steg som sker innan den faktiska kompileringen. Orsaken tros vara att den icke-optimerade IR-koden innehåller väldigt många instruktioner. Efter optimering har denna kod kortats ner betydligt, till följd av exempelvis **Instruction Combining** och **Aggressive DCE** [F]. Detta innebär att kompileringen efter det att optimeringen är utförd kommer ha mindre arbete att utföra. Således blir den totala tiden kortare vid optimering än om optimering inte utförs. Tiden för att optimera koden är alltså kortare än vad skillnaden i att kompilera icke-optimerad och optimerad LLVM IR-kod är.

Likt en fysisk MOS 6502 använder den interpreterande emulatorn PC-registret för att veta vart i koden den är. Registret ökar med 1 efter varje utförd instruktion, för att manipulera vilken del av koden som exekveras ändrar hoppinstruktioner PC-registret. DRC-emulatorn behöver dock inte förlita sig på PC-registret på samma sätt, eftersom den använder sig av LLVMs kontrollflöde genom BasicBlocks istället. På grund av detta blir PC-registret överflödigt.

DRC-emulatorn är skriven på ett sätt så att den kan vidareutvecklas för att vara en del av ett större system. Om en koppling med komponenter för grafik och ljud skulle göras behöver funktionalitet för synkronisering mellan dem implementeras.

5.2.3 LLVM

Vi valde att använda LLVM är för att det är ett lättillgängligt verktyg som förenklar arbetet att kompilera kod. Det är ett open-source projekt vilket gör det enkelt att hämta och integrera i vårt byggsystem. LLVM tillgängliggör även verktyg för att kompilera ramverket på ett plattformsoberoende sätt. En annan fördel är att det är skrivet i C++ vilket medför integrationen förenklas då vi ej behöver implementera någon brygga mellan olika språk. Trots detta var användandet av ramverket LLVM i arbetet till en början svårt att ta till sig, då LLVM i sig är ett stort projekt och innehåller många olika klasser. Storleken av projektet medförde problem att veta vad som var relevant för arbetet. Men när det var avklarat visade det sig att LLVM, i den formen som användes till projektet, var lättanvänt och rättframt i hur dess Application Programming Interface (API) användes från högnivåspråket C++.

Fördelen med att delegera arbetet med optimering och kompilering till LLVM är att eventuella förbättringar i de optimeringar som projektet tillhandahåller leder till prestandavinster i DRC-emulatorn. Samma resonemang kan även appliceras på det ständigt ökande antal ISA som LLVM kan kompilera maskinkod för. Båda dessa scenarion är troliga då LLVM är under aktiv utveckling, sponsrad av stora företag såsom Apple och Qualcomm [35].

5.3 Frågeställning

I denna sektion kommer huvudfrågan att besvaras utifrån de resultat som redovisats. Bifrågorna diskuteras och analyseras med utgångspunkt i erfarenheter från utförandet utav projektet.

5.3.1 Huvudfråga

Huvudfrågan lyder *Kommer en emulator som utnyttjar JIT med verktyg från LLVM resultera i maskinkod med bättre prestanda jämfört med en helt interpreterande emulator?*

Analysen och kompileringen som sker i DRC-emulatorn tillåter för potentiellt ytterligare optimering av programmet som exekveras på emulatorn, medan prestandan

utav den interpreterande emulatorn i princip är bunden till emulatorns implementation. Då den totala tiden räknas, från det att emulatorn startar tills att programmet är slutfört, är den interpreterande emulatorn snabbare om programmet har en kort exekveringstid. Anledningen till detta är att DRC-emulatorn har ytterligare två tidskrävande steg som måste utföras innan exekvering av programmet kan ske. När enbart exekveringstid jämförs mellan de två emulatorerna är DRC-emulatorn tveklöst snabbast. Svaret på huvudfrågan blir då ja, den resulterande maskinkoden blir mer effektiv när en JIT-kompilator, som nyttjar verktyg från LLVM, används jämfört med en helt interpreterande emulator.

5.3.2 Bifråga 1

Bifråga 1 lyder *Hur väl lämpar sig LLVM för att abstrahera hårdvara?*.

Svårigheten med att abstrahera hårdvara med LLVM ligger inte i att lyckas representera det interna tillståndet, då denna process är väldigt lik den för en interpreterande emulator. Istället ligger svårigheten i att korrekt uttrycka hur varje instruktion hos MOS 6502 påverkar detta interna tillstånd. Generering av IR-kod skedde genom anrop till funktioner tillhandahållna av LLVM-biblioteket. LLVM kommer med en rad intuitiva klasser som används för att bygga LLVM IR. Om man kommer från en bakgrund där man besitter grundläggande kunskap om instruktioner och olika ISA anser vi att dessa klasser är lättolkade och intuitiva att använda.

Vid generering av LLVM IR har användaren inte fullständig kontroll över vilken, eller vilka, instruktioner som kommer att genereras. Detta kan ses som en nackdel, men det möjliggör LLVM att vara agnostisk, det vill säga inte bunden till en specifik, underliggande ISA. Denna brist på kontroll är något som följer av att LLVM IR kan kompileras mot flertalet ISA. När man skapar IR-representationen räknar man med att LLVM också sköter genereringen av den faktiska maskinkoden för värdprocessorn. Det enda vi då behöver fokusera på är att hitta LLVM IR-instruktioner som översätts till en likvärdig handling som den ursprungliga maskinkoden vill åstadkomma.

LLVM hjälpte alltså till att reducera det totala arbetet med implementeringen av DRC-emulatorn genom att tillhandahålla verktyg för kompileringen. Detta är annars en tidskrävande komponent att implementera. Slutsatsen som kan dras är att användandet av LLVM lämpade sig väl för vårt ändamål och kan jämna ut utvecklingstiden mellan en DRC-emulator och en motsvarande interpreterande emulator.

5.3.3 Bifråga 2

Bifråga 2 lyder *Vilka utmaningar finns vid JIT-kompilering då utgångspunkten är maskinkod snarare än källkod?*.

De problem som uppstår då utgångspunkten är maskinkod snarare än källkod beskrivs i teorikapitlet 2.1. Maskinkod saknar mycket information om programmet,

och detta löstes i viss mån under DRC-emulatorns analysdel [3.2].

Det finns ett antal instruktioner i MOS 6502s ISA som gör det omöjligt att statiskt omkompilera koden. En av dessa är JMP Indirect, detta är en instruktion som utför ett hopp till en adress som inte är känd vid kompilering. Med andra ord räcker det inte att titta på binärfilen för att veta vilken adress som är målet. Normalt sett då källkod kompileras till maskinkod har kompilatorn kännedom om all maskinkod. Indirekta hopp kan därför enkelt genereras av kompilatorn vid behov. På grund av att det inte finns något sätt att uttrycka indirekta hopp i LLVM IR, var detta inte en möjligheten för projektet.

Vår plan var istället att använda dynamisk omkompilering för att möjliggöra indirekta hopp i LLVM IR. Syftet med omkompileringen var att under exekvering skapa ett nytt BasicBlock vars start är måladressen för instruktionen; JMP Indirect. Då ett nytt block genererats och lagts till i den dynamiska hopptabellen, beskriven i metodkapitlet 3.2, kan instruktionen då genomföras. I brist på tid kunde tyvärr inte en sådan lösning implementeras. Istället avslutas programmet som exekveras av emulatorn med en felkod när JMP Indirect försöker hoppa till en adress som inte är en start av ett BasicBlock.

Samma problematik som finns kring JMP Indirect gäller även för RTS, denna instruktion är till för att returnera från en känd subrutin. Vid första anblick kan det verka oklart varför denna instruktion är svår att hantera. Svaret är att på tiden då MOS 6502 var aktuell användes alla möjliga sätt för att minska antal processorcykler samt reducera mängden minne som behövde användas. Det ledde till att man använde instruktioner på ett sätt som de inte var tänka till. RTS använder stacken för att ta reda på vart kontrollflödet ska återvända. Vanligvis är det instruktionen som efterföljer subrutinsanropet, men detta kan ändras genom att manipulera stacken och tvinga processorn att hoppa någon annanstans. Detta leder till exakt samma problem som vid indirekta hopp, det vill säga att det inte finns en känd instruktionsväg till målblocket.

Eftersom vi endast omkompilerar programmet en gång, saknas stöd för självmodifierande program. Inte heller går det att dra nytta av de eventuella prestandavinster som skulle kunna fås från att iterativt optimera kodblock som återanvänds ofta. Däremot optimerar DRC-emulatorn i sin nuvarande form hela koden, och är således redan optimalt optimerad. Vid dynamisk omkompilering hade detta dock behövts göras under exevering när nya block genereras.

5.4 Verktyg

Under arbetets gång har olika verktyg tagits fram. Verktygen som skapades var en debugger och en assemblator. Vi som grupp insåg inte under planeringsfasen behovet av de verktyg som i slutändan implementerades, vilket introducerade avvikningar från den ursprungliga handlingsplanen. Debuggern bidrog till att den interpreterande emulatorn kunde felsökas på ett strömlinjeformat sätt. Assemblatorn hjälpte till

vid implementeringen utav de funktions- och prestandatester som hjälpte till att producera resultat i projektets slutskede.

5.4.1 Debugger

För att lättare analysera och verifiera att den interpreterande emulatorn fungerade på förväntat sätt, implementerades en grafisk debugger. För att hantera utritning av debuggerns gränssnitt övervägdes först OpenGL som grafiskt bibliotek [36]. OpenGL uppfyllde det grundläggande krav vi hade på att grafiken skulle vara plattformsoberoende, men det bedömdes vara för komplicerat för att hinna implementera debuggern i rimlig tid. Efter att gruppen lagt till kravet att det grafiska biblioteket även skulle vara snabbt att utveckla i riktades fokus åt andra alternativ.

Det första valet var ett bibliotek som kallas SDL [37]. Det ansågs vara ett bra alternativ då det stödjer alla möjliga operativsystem och versioner, och är skrivet i i ANSI-C vilket medför kompatibilitet även med gamla kompilatorer. I biblioteket ingår ett API som ger en tillgång till ljud, grafik och indata. Men allt eftersom projektet fortskred behövde fokus läggas på emulatorerna, vilket medförde att debuggern nedprioriterades. På grund av detta övergavs SDL till förmån för ramverket Qt genom biblioteket IMGUI [38] [31]. Detta bibliotek har samma fördelar med avseende på kompatibilitet som SDL. Samtidigt gör Qt det enklare att iterera på en design då det finns möjlighet till att skapa GUI med hjälp av visuella verktyg.

Funktionalitet som implementerades innefattar: möjlighet att ladda valfritt program under exekvering, starta, stanna och stega igenom programmet. Att starta ett program innebär att emulatorn exekverar instruktioner en efter en och inte stannar förrän man trycker stopp. Att stega igenom ett program innebär att bara en instruktion utförs innan användaren trycker på stegknappen igen. Detta har varit till stor användning då innehållet av alla register kunde studeras efter varje instruktion. Även en specifik ruta som visar maskininstruktionen i läsbar text finns tillgänglig för att förenkla felsökningsprocessen.

5.4.2 Assemblator

När det var dags att implementera program för att utvärdera emulatorernas funktionalitet och prestanda skrevs de första programmen i maskinkod. Denna metod visade sig snabbt vara tidskrävande, kognitivt jobbig och framförallt felbenägen. Gruppen valde då att skriva en assemblator för att underlätta denna process. Möjligheten att skriva assembler visade sig vara markant lättare då labels och mnemonics bidrog till att semantiken hos ett givet program blev lättare att tolka. Det blev tydligare vilka instruktioner som anropades samt att alla i gruppen hade tidigare erfarenhet av assembler. Det visade sig också användbart att ha programmen i assembler vid felsökning av prestandatesterna.

Det finns flera anledningar som motiverar att en implementera en egen assemblator istället för att använda en existerande. Det gav gruppen möjlighet att själv kun-

na bestämma över den assemblersyntax som användes. Detta underlättade arbetet för alla i gruppen, då hjälpfunktioner enkelt kunde definieras med hjälp av macro-definitioner. Macro-definitioner åsyftar koncisa kodstycken som kan återanvändas. Utmaningen med att själva implementera konvertering från en relativt småskalig assembler till maskinkod gav god insikt i hur källkod och maskinkod hänger ihop. Denna förståelse förenklade bland annat arbetet med parsningen av maskinkod i DRC-emulatorn. Fördelarna med att implementera assemblatorn själva, i kombination med den snabbare implementationen utav testprogram, ansågs berättiga den sammanlagda krävda tidsåtgången.

5.5 Etik

Eftersom emulatorer tillåter exekvering utav program på plattformar som användaren inte har fysisk tillgång, eller äganderätt till, bidrar det till etiska samt juridiska tveksamheter. Ett vanligt argument mot emulering är att det i praktiken kan leda till ökad användning av piratkopierad mjukvara. Då emulering i sig inte är ämnat för piratkopierad mjukvara, och varken inkräktar på någons integritet eller autonomi, är det svårt att göra en tydlig diskvalificering av emulation som koncept på etiska grunder. Av denna anledning är emulering tillåtet enligt svensk lag [39].

Det finns för samhället både positiva och negativa etiska aspekter att ta hänsyn till vid emulering av hårdvara.

5.5.1 Samhällsnyttiga aspekter

Med positiva etiska aspekter åsyftas huruvida vidareutveckling av DRC-emulatorn kan gynna flera delar av samhället. Till denna kategori återfanns två aspekter: cirkulär ekonomi och feltolerans.

Genom att äldre processorer abstraheras till en nivå där de kan användas på nyare och snabbare processorer bidrar detta till att äldre program, exempelvis äldre bankprogram, kan fortsätta brukas på modernare datorer. Det är dock inte bara äldre program som kan gynnas av detta. Spelkonsoler och deras spel hade varit spelbara på moderna datorer och kan därmed fortsätta leverera ett underhållningsvärde. Abstrahering av maskinkod bidrar inte bara till att äldre program och spel blir användbara på nytt, det kan också bidra till en mer cirkulär ekonomi. Företag behöver exempelvis inte spendera pengar på att utveckla ny mjukvara varje gång de uppdaterar ett system och dess hårdvara.

Det positiva med att kunna återanvända exempelvis ett banksystem är att den gamla koden enkapsulerar många år av domänspecifik kunskap, buggfixar och lärdomar. Vid omskrivning av större system finns alltid en överhängande risk att säkerhetskritiska buggar återintroduceras. Det finns också en stor risk att helt nya buggar tillkommer i den nya implementationen, särskilt om systemen inte är särskilt väl testade till att börja med eller om personer som besitter djup domänkunskap inte längre finns att tillgå.

5.5.2 Samhällsdestruktiva aspekter

Med negativa etiska aspekter menas huruvida vidareutveckligen av DRC-emulatorn kan vara destruktiv för samhället. Till denna kategori återfinnes tre aspekter: piratkopiering, fel upplevelser och digitala säkerhetsintrång.

Vid samhälldestruktiva etiska aspekter är en uppenbar oetisk aspekt piratkopiering. Piratkopiering innebär att man plagierar någon annans arbete och distribuerar detta arbetet olagligt till andra. Hur piratkopiering relaterar till just detta projekt är att vidareutveckling av DRC-emulatorn kan leda till en emulator av en spelkonsol, exempelvis NES. Emuleringen av spelkonsolen i sig klassas inte som piratkopiering. Piratkopieringen sker vid konsumtion av spelen, det är först då den immateriella rätten hos upphovsinnehavaren kränks.

De fysiska spelen kan inte spelas på en emulerad spelkonsol, utan ROM-filerna måste kopieras ifrån de fysiska kassetterna till ett digitalt format. Detta sker via en så kallad ROM-dumper, ett litet kretskort som man stoppar kassetten i för att sedan kopiera in ROM-filen till datorn. Dessa ROM-filer kan sedan distribueras mellan diverse aktörer olagligt, genom webbsidor som The Pirate Bay eller dylikt.

Om vidareutveckling av DRC-emulatorn skulle fortskrida, vore inte bara piratkopiering aktuellt utan även fel upplevelser. Det finns många emuleringsfel som då kan uppstå och som påverkar användarupplevelsen negativt. Tydliga exempel inkluderar fördröjning mellan användarens inmatning och att en förändringen sker, eller att grafiska artefakter uppstår vid utritning av en scen från ett spel. Båda dessa buggar riskerar att manifestera sig vid vidareutveckling utav projektets nuvarande DRC-emulator, då den i nuläget varken hanterar synkronisering eller grafik. Dessa störningsmoment kan leda till att användaren upplever kraftig irritation, men det kan också ge en negativ bild av företaget som gjort spelet från början.

Det negativa med att kunna återanvända exempelvis ett banksystem är att den gamla koden enkapsulerar många år av domänspecifik kunskap. Detta har tidigare nämnts som en positiv egenskap med gamla system, vilket det är i vissa avseenden. Under åren som systemen har funnits har, förhoppningsvis, intrikata buggar identifierats och fixats utan att senare återintroduceras. Systemets tillämpning är också förhoppningsvis välkänd efter många år i drift, och därmed nära buggfritt om det fortsätter att användas på samma sätt. En kritisk punkt nås när systemet av någon anledning behöver uppdateras med ny funktionalitet som innebär ändring av den ursprungliga källkoden. Gammal källkod som enkapsulerar många år av domänspecifik kunskap kräver antagligen att mjukvaruutvecklarna besitter expertis och erfarenhet inom branschen. Detta kan medföra att koden blir känslig för, eventuellt svårarbetad vid, introduktion av ny funktionalitet över tid. En känslig fråga för de företag som besitter gamla system blir då: *”Vad är planen när era utvecklare slutar/försvinner?”*.

System som ej uppdateras till att använda ny teknik, vid behov, riskerar att fastna med säkerhetskritiska buggar. I värsta fall är dessa svåra att fixa. I de fall då

systemen förlitar sig på lusbefästa, binära bibliotek kan buggarna i praktiken bli omöjliga att fixa [40]. Det går endast att spekulera om potentialen av att utnyttja buggar för skadlig avsikt, men helt klart är att systemen kan nyttjas på andra sätt än vad de var designade för.

5.6 Källkritik

Dynamisk omkompilering är inte ett nytt påfund i sig. Däremot är det inte, för ändamålet emulering, ett ämne inom vilket det finns bred forskning tillgänglig. Detta innebär att det varit svårt att hitta vetenskapliga källor för vissa delar av projektet. I den mån det varit möjligt har vi använt oss av vetenskapliga artiklar, men vi har ibland fått förlita oss på hobbyprojekt, blogginlägg och wiki-sidor likt NESDEV [24]. I dessa fall har det varit viktigt för oss att granska källan noga, för att se om den är relevant och tillförlitlig. Ofta är det personer med akademisk bakgrund, eller som arbetar på framstående IT-företag, som författat dessa källor, och då har vi dömt dem värdiga för källhänvisning.

På wiki-sidor är det svårare att spåra författarnas trovärdighet, och här har vi i den mån det är möjligt valt att använda de källor som författarna använt istället för själva texten på wiki-sidan.

6
Slutsats

Denna rapporten har visat att en DRC-emulator kan implementeras med hjälp av LLVM. Användandet av LLVM tillåter utvecklare utan tidigare erfarenhet av kompilatorkonstruktion att nyttja JIT-kompilering för att generera effektiv maskinkod. LLVMs abstraktioner döljer intrikata detaljer kring hur den faktiska kodgenerering, av både LLVM IR samt maskinkod, går till. Detta bidrar till att en DRC-emulator blir enklare att implementera än om dessa steg hade behövts utföras manuellt. Inget extra arbete behövde utföras för att tillåta DRC-emulatorn att kompileras till olika ISA.

Rapporten har visat att den resulterande maskinkoden från dynamisk omkompilation är mer effektiv än maskinkoden som motsvarar samma arbete i en interpreterande emulator. Tack vare analys av den binära maskinkoden undviker DRC-emulatorn att tolka en instruktion i taget, vilket är huvudanledningen till den mer effektiva maskinkoden.

Det finns komplikationer som uppstår vid JIT-kompilering av maskinkod. Detta inkluderar, men är inte begränsat till, att information som hjälper till att avgöra om binär data ska tolkas som en instruktion eller data ej finns tillgänglig. Denna nödvändiga information måste urskiljas under exekvering, vilket leder till att extra logik måste implementeras i den dynamiskt omkompilerande emulatorn jämfört med om programmets källkod finns tillgänglig.

Källförteckning

[1] E. Amos, Creator, *Transparent version of NES console*, Tillstånd: "I, the copyright holder of this work, release this work into the public domain. This applies worldwide. In some countries this may not be legally possible; if so: I grant anyone the right to use this work for any purpose, without any conditions, unless such conditions are required by law.", 2010. [Online]. Tillgänglig: `https://commons.wikimedia.org/wiki/File:NES-Console-Set.png` , hämtad: mars 14, 2020.

[2] R. F. Rosin, "Contemporary Concepts of Microprogramming and Emulation", *ACM Comput. Surv.*, årg. 1, nr 4, s. 197–212, dec. 1969, ISSN: 0360-0300. DOI: `10.1145/356556.356559`. [Online]. Tillgänglig: `https://doi-org.proxy.lib.chalmers.se/10.1145/356556.356559`.

[3] C. Y. Baldwin och K. B. Clark, *Design Rules: The Power of Modularity Volume 1*. Cambridge, MA, USA: MIT Press, 1999, ISBN: 0262024667.

[4] LLVM-admin team, "Performance Tips for Frontend Authors", 2020. [Online]. Tillgänglig: `https://llvm.org/docs/Frontend/PerformanceTips.html` , hämtad: mars 20, 2020.

[5] *IBM Z The banking platform for the future*, 2020. [Online]. Tillgänglig: `https://www.ibm.com/industries/banking-financial-markets/resources/core-banking/ibm-z-banking-platform/` , hämtad: april 17, 2020.

[6] A. Tikkanen, "Computer basis", *Britannica Academic*, [Online]. Tillgänglig: `https://academic-eb-com.proxy.lib.chalmers.se/levels/collegiate/article/computer/117728` , hämtad: april 29, 2020.

[7] A. V. Aho, *Compilers, Principles, Techniques, & Tools*, 2. utg. USA: Pearson Addison-Wesley, 2007, ISBN: 978-0-321-49169-5. , hämtad: april 05, 2020.

[8] P. Diskin, "Nintendo Entertainment System Documentation", 2004. [Online]. Tillgänglig: `http://www.nesdev.com/NESDoc.pdf` , hämtad: febr. 07, 2020.

[9] M. Fayzullin. (2000). How To Write a Computer Emulator, Department of Computer Science, University of Maryland, [Online]. Tillgänglig: `http://fms.komkon.org/EMUL8/HOWTO.html` , hämtad: april 20, 2020.

[10] B. Goetz, "Dynamic compilation and performance measurement", [Online]. Tillgänglig: `https://www.ibm.com/developerworks/library/j-jtp12214/index.html` , hämtad: maj 08, 2020.

[11] IT-ord, ”parsning”, 2019. [Online]. Tillgänglig: https://it-ord.idg.se/ord/parsning/ , hämtad: maj 08, 2020.

[12] S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, Land: United States: Morgan Kaufmann, Publichers Inc., 1998, ISBN: 978-1-55860-320-2.

[13] “MOS Microprocessors Programming Manual, MCS6500, MICROCOMPUTER FAMILY, PROGRAMMING MANUAL”, Norristown, PA, USA: MOS Technologies Inc., 1976. [Online]. Tillgänglig: http://archive.6502.org/books/mcs6500_family_programming_manual.pdf , hämtad: april 27, 2020.

[14] *Foundations of Programming Languages - Just-In-Time Compilation*, 2015. [Online]. Tillgänglig: https://www.youtube.com/watch?v=yQ27DjKnxwo&t=1s.

[15] J. Aycock, ”A Brief History of Just-In-Time”, *ACM Computer Surveys*, årg. 35, nr 2, s. 97–113, juni 2003. DOI: 10.1145/857076.857077.

[16] S. Oaks, *Java Performance: The Definitive Guide*. O’Reilly Media, Inc., 2014, ISBN: 9781449358457. [Online]. Tillgänglig: https://www.oreilly.com/library/view/java-performance-the/9781449363512/ , hämtad: april 17, 2020.

[17] (2020). TJava SE HotSpot at a Glance, [Online]. Tillgänglig: https://www.oracle.com/java/technologies/javase-jsp.html , hämtad: juni 02, 2020.

[18] S. Kh, Скриншот, Princip för JIT exekveringsmiljö, 2015. [Online]. Tillgänglig: https://commons.wikimedia.org/wiki/File:%D0%A1%D0%BA%D1%80%D0%B8%D0%BD%D1%88%D0%BE%D1%82_2015-12-11_01.29.19.png , hämtad: maj 03, 2020.

[19] LLVM-admin team, “LLVM: 10 Short Years Since 1.0”, 2013. [Online]. Tillgänglig: https://llvm.org/devmtg/2013-11/slides/Adve-LLVM10yr.pdf , hämtad: mars 28, 2020.

[20] LLVM-admin team. (2020). LLVM - Building a JIT, [Online]. Tillgänglig: https://llvm.org/docs/tutorial/BuildingAJIT1.html , hämtad: febr. 12, 2020.

[21] LLVM-admin team. (2020). The LLVM Target-Independent Code Generator, [Online]. Tillgänglig: https://llvm.org/docs/CodeGenerator.html , hämtad: maj 10, 2020.

[22] LLVM-admin team, “LLVM’s Analysis and Transform Passes”, 2020. [Online]. Tillgänglig: https://llvm.org/docs/Passes.html , hämtad: mars 05, 2020.

[23] LLVM-admin team, “Using The LLVM Libraries”, 2020. [Online]. Tillgänglig: https://releases.llvm.org/2.8/docs/UsingLibraries.html , hämtad: mars 20, 2020.

[24] NESDEV, “Nesdev wiki”, 2020. [Online]. Tillgänglig: http://wiki.nesdev.com/w/index.php/Nesdev_Wiki , hämtad: febr. 28, 2020.

[25] dr. Fone. (2020). Topp 10 NES Emulatorer - Spela NES-spel på andra enheter, [Online]. Tillgänglig: http://global.drfone.biz/sv/emulator/nes-emulator.html#part1 , hämtad: juni 02, 2020.

[26] dr. Fone. (2020). Topp 10 NES Emulatorer - Spela NES-spel på andra enheter, [Online]. Tillgänglig: http://global.drfone.biz/sv/emulator/nes-emulator.html#part2 , hämtad: juni 02, 2020.

[27] dr. Fone. (2020). Topp 10 NES Emulatorer - Spela NES-spel på andra enheter, [Online]. Tillgänglig: http://global.drfone.biz/sv/emulator/nes-emulator.html#part3 , hämtad: juni 02, 2020.

[28] T. Scherrer. (2020). Z80 Family Official Support-Page, [Online]. Tillgänglig: http://www.z80.info/ , hämtad: juni 02, 2020.

[29] V. Rosario, M. Hato, R. Azevedo och E. Borin, "A Methodology for Optimization of Interpreters", sept. 2018, s. 1. DOI: 10.1109/WSCAD.2018.00040.

[30] A. Jacobs. (2002). Basic Architecture, [Online]. Tillgänglig: http://www.obelisk.me.uk/6502/architecture.html , hämtad: maj 05, 2020.

[31] O. Cornut, "Dear ImGui: Bloat-free Immediate Mode Graphical User interface for C++ with minimal dependencies", 2020. [Online]. Tillgänglig: https://github.com/ocornut/imgui , hämtad: maj 10, 2020.

[32] H.-K. Arntzen, "An unusual recompiler experiment – MIPS to LLVM IR", 2020. [Online]. Tillgänglig: https://themaister.net/blog/2019/01/27/an-unusual-recompiler-experiment-mips-to-llvm-ir-part-1/ , hämtad: maj 14, 2020.

[33] K. Dormann, "Tests for all valid opcodes of the 6502 and 65C02 processor", 2013. [Online]. Tillgänglig: https://github.com/Klaus2m5/6502_65C02_functional_tests , hämtad: april 28, 2020.

[34] M. Cecconi. (2020). A csharp emulator for the Zilog Z80 CPU, [Online]. Tillgänglig: https://github.com/sklivvz/z80 , hämtad: juni 02, 2020.

[35] The LLVM Foundation, "Sponsors", 2020. [Online]. Tillgänglig: https://llvm.org/foundation/sponsors.html , hämtad: maj 01, 2020.

[36] Khronos Group, "OpenGL Overview", 2020. [Online]. Tillgänglig: https://www.opengl.org/about/ , hämtad: maj 10, 2020.

[37] SDL Community, "About SDL", 2020. [Online]. Tillgänglig: https://www.libsdl.org/ , hämtad: maj 10, 2020.

[38] The Qt Company, "Why QT?", 2020. [Online]. Tillgänglig: https://www.qt.io/why-qt , hämtad: maj 10, 2020.

[39] SFS 2018:1099, *Lag (1960:729) om upphovsrätt till litterära och konstnärliga verk*, Justitiedepartementet L3. [Online]. Tillgänglig: http://www.riksdagen.se/sv/Dokument-Lagar/Lagar/Svenskforfattningssamling/Forordning-200990-om-statsb_sfs-2009-90/?bet=2009:90 , hämtad: jan. 02, 2017.

[40] K. Thompson. (1984). Reflections on Trusting Trust, [Online]. Tillgänglig: https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf , hämtad: maj 10, 2020.

[41] P. Pritchard, "Linear prime-number sieves: A family tree", 1987. DOI: 10.1016/0167-6423(87)90024-4. [Online]. Tillgänglig: https://doi.org/10.1016/0167-6423(87)90024-4 , hämtad: maj 08, 2020.

A

Statusregister av MOS 6502

Bit	Namn	Förkortning	Betydelse
0	Carry	C	Sätts till 1 om resultatet överskrider 8 bitar.
1	Zero	Z	Sätts till 1 om resultatet blir 0.
2	IRQ Disable	I	Stänger av all avbrottshantering om biten är 1.
3	Decimal mode	D	Processorn hamnar i decimalt läge där varje byte endast representerar talen 0 till 9.
4	BRK command	B	Om biten är 1, triggas ett NMI.
5	Används ej	-	-
6	Overflow	V	Sätts till 1 om en räkneoperationer hamnar utanför talrymnden.
7	Negative	N	Sätts till 1 om högsta biten i ett resultat är satt. Det vill säga om resultatet är negativt.

Figur A.1: Tabell över statusregistret för MOS 6502.

B

Samtlig adressering av MOS 6502

Adresseringslägena är uppdelade i två distinkta adresseringssätt: indirekta och direkta.

B.1 Direkt

Immediate - Nästkommande adress innehåller värdet som gällande instruktion kommer att använda.

Implied - Instruktioner med denna typ av adressering använder inte minnet, utan ändrar bara processorns interna tillstånd. Exempelvis instruktionen CLC, som nollställer Carry-flaggan.

Relative - En minnesadressering som hanterar villkorliga hoppinstruktioner kopplade mot statusregistet.

Accumulator - Läser inte minnet, utan förändrar värdet i register A.

B.2 Indirekt

Indirect - Används enbart av JMP instruktionen. Detta adresseringssätt hämtar den 16-bitars adress som följer instruktionen, och läser 16-bitar från den adressen. Det värdet som läses används sedan av instruktionen.

Indexed Indirect - Använder värdet i X registret som adderas med värdet på nästkommande minnesadress. Summan är en 8-bitars Zeropage-adress som läses för att hämta värdet som instruktionen ska använda sig av.

Indirect Indexed - Använder Y registret. Nästkommande minnesadress används som en pekare till en Zeropage-adress, innehållet av denna adress läggs ihop med innehållet av den efterföljande adressen. Ihopsättningen av dessa bildar bas-adressen som sedan adderas med Y för att få den slutgiltiga adressen. Ett väldigt vanligt adresseringssätt som används för att exempelvis indexera en sträng vars adress bestäms under körning.

B.3 Absolut

Absolute - De två kommande minnesadresserna sätts ihop, resultatet är en 16 bitars pekare till en minnesadress. Minnet läses på denna plats, och värdet som finns där används.

Absolute, X / Absolute, Y - Dessa två fungerar på samma sätt som Absolute, men X respektive Y registret adderas till den adress som hittas efter instruktionen. Detta är användbart i exempelvis loopar.

Zeropage - Denna adresseringsmetod kan enbart komma åt de första 256 minnesadresserna. Fördelen med detta är att de kräver färre cykler och tar mindre plats i instruktionsminnet än övriga adresseringssätt.

Zeropage, X / Zeropage, Y - Dessa fungerar på samma sätt som Absolute X/Y gör, men kommer återigen endast åt de första 256 minnesadresserna.

C

Instruktionslista för MOS 6502

I tabellerna [C.1],[C.2],[C.3] visas alla instruktioner för MOS 6502, med instruktions förkortning, förklaring och vilka flaggor som berörs. För flaggor är det enbart de flaggor som benämns med bokstav som berörs.A Om Enbart flaggan nämns kan den både sätta 1 eller 0. Annars står det explicit vad flaggan sätts till.

Förkortning	Förklaring	Påverkade flaggor
ADC	Add with carry	Z,C,N
AND	Logical AND	Z,N
ASL	Arithmetic Shift Left	Z,C,N
BCC	Branch if Carry Clear	-
BCS	Branch if Carry Set	-
BEQ	Branch if Equal	-
BIT	Bit Test	Z,N,V
BMI	Branch if Minus	-
BNE	Branch if Not Equal	-
BPL	Branch if Positive	-
BRK	Force Interrupt	B
BVC	Branch if Overflow Clear	-
BVS	Branch if Overflow Set	-
CLC	Clear Carry Flag	C
CLD	Clear Decimal Mode	D
CLI	Clear Interrupt Disable	I

Figur C.1: Tabell över alla instruktioner för MOS 6502.

Förkortning	Förklaring	Påverkade flaggor
CLV	Clear Overflow Flag	V
CMP	Compare	Z,C,N
CPX	Compare X Register	Z,C,N
CPY	Compare Y Register	Z,C,N
DEC	Decrement Memory	Z,N
DEX	Decrement X Register	Z,N
DEY	Decrement Y Register	Z,N
EOR	Exclusive or	Z,N
INC	Increment	Z,N
INX	Increment X	Z,N
INY	Increment Y	Z,N
JMP	Jump	-
JSR	Jump to subroutine	-
LDA	Load accumulator (A)	Z,N
LDX	Load X	Z,N
LDY	Load Y	Z,N
LSR	Logical shift right	Z,C,N=0
NOP	No operation	-
ORA	Or with accumulator	Z,N
PHA	Push accumulator	-
PHP	Push processor status	-
PLA	Pull accumulator	Z,N
PLP	Pull processor status	From stack
ROL	Rotate left	Z,C,N
ROR	Rotate right	Z,C,N

Figur C.2: Tabell över alla instruktioner för MOS 6502.

Förkortning	Förklaring	Påverkade flaggor
RTI	Return from interrupt Register	From stack
RTS	Return from subroutine or	-
SBC	Subtract with carry	Z,C,N,V
SEC	Set carry	C=1
SED	Set decimal	D=1
SEI	Set interrupt disable	I=1
STA	Store accumulator (A)	-
STX	Store X	-
STY	Store Y X	-
TAX	Transfer accumulator to X	Z,N
TAY	Transfer accumulator to Y	Z,N
TSX	Transfer stack pointer to X	Z,N
TXA	Transfer X to accumulator	Z,N
TXS	Transfer X to stack pointer	-
TYA	Transfer Y to accumulator	Z,N

Figur C.3: Tabell över alla instruktioner för MOS 6502.

D

Prestandatest källkod

D.1 Bubble sort

Bubble sort är en sorteringsalgoritm som går igenom en lista och jämför två element åt gången. Om det första elementet är större än den andra byter de plats i listan. Denna implementationen traverserar alltid listan från den ena änden till den andra, på så sätt kommer de största elementen till slut att ”bubbla upp” och hamna i ena änden av listan. För varje omgång blir minst ett värde rättplacerat, vilket innebär att denna implementation utav Bubble sort har komplexitet $\mathcal{O}(x^2)$.

Listing D.1: Bubble sort.

```
#macro PAGE_COUNT 0 #32 #endmacro

.org $8000

list:

.db 27, 103, 255, 98, 181, 158, 253, 200
.db 23, 126, 145, 140, 25, 27, 33, 139
.db 13, 33, 218, 42, 16, 233, 232, 114
.db 14, 89, 49, 132, 53, 134, 115, 50
.db 240, 5, 52, 56, 136, 251, 83, 238
.db 225, 62, 199, 17, 251, 42, 11, 39
.db 15, 159, 232, 162, 228, 82, 111, 101
.db 113, 154, 219, 28, 93, 161, 116, 225
...
; Listing of data to sort repeats several more times.
; In total 8192 (2^13) elements were sorted.
...

.org $E000
main:
    LDA #<list          ; A = list_L
    LDX #>list          ; X = list_H
    STA $00             ; List address will be on ZP $00
    STX $01
```

```
main_loop:
    LDA is_sorted
    BNE done
    LDA #1
    STA is_sorted    ; Before looping, set done to true
                     ; (the algorithm will set it to false
                     ; if it's not done)
    LDA base_page
    STA $01          ; Reset page
    LDA #0
    STA page_index   ; Reset page index
    LDY #0           ; Reset iterator
pass:
    LDA ($00), Y     ; A = first element
    CPY #$FF
    BEQ inc_page
    INY
pass_cmp:
    CMP ($00), Y     ; Compare to next element
    BEQ pass         ; If ==, goto next
    BCS swap         ; If >, goto 'swap'. If < goto next
    JMP pass
inc_page:
    INC $01          ; Increase page by 1
    INC page_index
    LDY PAGE_COUNT
    CPY page_index   ; If == to page count, go to next main_loop
    BEQ main_loop
    LDY #0
    JMP pass_cmp
swap:
    LDX #0
    STX is_sorted    ; Swap means done = false
    CPY #0
    BEQ page_edge_swap
    STA tmp
    LDA ($00), Y
    DEY
    STA ($00), Y
    INY
    LDA tmp
    STA ($00), Y
    JMP pass
page_edge_swap:
    STA tmp
    LDA ($00), Y
```

```
    DEY
    DEC $01
    STA ($00), Y
    INY
    INC $01
    LDA tmp
    STA ($00), Y
    JMP pass

done:
    LDA #0
    STA $200F       ; Exit 0
    LDY #0          ; Reset iterator
    LDA base_page
    STA $01         ; Reset page
    LDA #0
    STA page_index  ; Reset page index
print_loop:
    LDA ($00), Y
    STA $2009
    INY
    CPY #$FF
    BEQ print_inc_page
    JMP print_loop
print_inc_page:
    INC $01         ; Increase page by 1
    INC page_index
    LDY PAGE_COUNT
    CPY page_index  ; If == to page count, go to next main_loop
    BEQ exit
    LDY #0          ; Else, clear Y and start on next page
    JMP print_loop
exit:
    LDA #0
    STA $200F       ; Exit 0

is_sorted:
.db 0

tmp:
.db 0

base_page:
.db $80

page_index:
```

```
.db 0


.org $FFFC
.dw main ; Reset routine
```

D.2 Fibonacci

Fibonacci-programmet bygger på Leonardo Fibonaccis berömda talserie. Serien räknar rekursivt ut nästa tal i serien genom att summera de två föregående talen. Startvärdena för serien är 0 respektive 1, och från dessa två värden börjar serien alltså att beräknas. Eftersom denna implementation direkt använder de två föregående talen för att beräkna det nästföljande talet är den algoritmiska komplexiteten $\mathcal{O}(N)$.

Listing D.2: Fibonacci.

```
.org $8000

main:
        LDA  #0
        STA  $F0
        STA  $F1
        LDA  #1
        STA  $F2
        LDA  #0
        STA  $F3
        LDX  #0
loop:   LDA  $F2
        STA  $0F1B,X
        INX
        PHA
        LDA  $F3
        STA  $0F1B,X
        STA  $F5
        PLA
        STA  $F4
        ADC  $F0
        STA  $F2
        LDA  $F5
        ADC  $F1
        STA  $F3

        LDA  $F4
        STA  $F0
        LDA  $F5
        STA  $F1
        INX
        CPX  #48        ; FIB(24) max that 16 bit can hold
        BMI  loop
exit:
        LDA #0
```

```
        STA $200F

.org $FFFC
.dw main ; Reset vector
```

D.3 Sieve of Eratosthenes

Programmet Sieve of Eratosthenes finner primtal i ett interval $[2, n)$. Programmet startar vid 2 och genomsöker alla efterföljande tal upp till n. Alla multiplar av det nuvarande talet som understiger den övre gränsen n stryks. Alla tal i intervallet som ej strukits när programmet exekverat färdigt är endast delbara med sig själv, samt 1, och är därför primtal. Den algoritmiska komplexiteten för denna implementation är $\mathcal{O}(N \cdot log(logN))$ [41, p. 33].

Listing D.3: Sieve of Erathostenes

```
.org $8000

main:   LDA  #$64       ; n
        STA  $D0        ; value of n
        LDA  #$00
        LDX  #$00
setup:  STA  $1000,X    ; populate array
        ADC  #$01
        INX
        CPX  $D0        ; read value of n from memory
        BPL  set
        JMP  setup
set:    LDX  #$02       ; first prime number
sieve:  LDA  $1000,X    ; find non-zero
        INX
        CPX  $D0
        BPL  sieved
        CMP  #$00
        BEQ  sieve
        STA  $D1        ; current prime
mark:   CLC
        ADC  $D1
        TAY
        LDA  #$00
        STA  $1000,Y
        TYA
        CMP  $D0
        BPL  sieve
        JMP  mark
sieved: LDX  #$01
        LDY  #$00
copy:   INX
        CPX  $D0
        BPL  copied
```

```
        LDA   $1000,X
        CMP   #$00
        BEQ   copy
        STA   $3000,Y
        INY
        JMP   copy
copied: TYA                 ; how many found
        LDA #0
        STA $200F           ; exit 0

.org $FFFC
.dw main                    ; Reset vector
```

E

Specifikationer för testdatorer

Komponent	Specifikation
CPU	2.5 GHz Intel Core i7-4870HQ (Turbo upp till 3.7 GHz)
ISA	x86-64, 64-bit
Minne	16GB 1600MHz DDR3L
Operativsystem	GNU/Linux 5.6.10
Kompilator	Clang 10
Kompilatorflaggor	-O3 -DNDEBUG

Figur E.1: Specifikationer för Macbook Pro 2015.

Komponent	Specifikation
CPU	Quad Core 1.2GHz Broadcom BCM2837
ISA	ARM, 64-bit
Minne	1GB 500MHz LPDDR2 SDRAM
Operativsystem	GNU/Linux 5.4.0
Kompilator	Clang 10
Kompilatorflaggor	-O3 -DNDEBUG

Figur E.2: Specifikationer för Raspberry Pi 3 Model B.

F

Optimeringar vid dynamisk omkompilering

Nio olika optimeringar användes vid omkompilering som utfördes av DRC-emulatorn. Alla dessa optimeringar implementeras som en del av LLVM-projektet.

Constant Propagation - Använder Immediate för kända konstanter.

Instruction Combining - Kombinerar instruktioner för att utföra färre och enklare instruktioner.

Aggressive DCE - *Aggressive dead code elimination* antar att värden är ”döda” tills motsatsen bevisats.

Loop Simplify CFG - Assisterar i flödesplanen för slingor.

LICM - Raderar så mycket kod inuti en slinga som möjligt.

Loop Sink - Itererar genom alla instruktioner i slingas preheader och lägger dem i slingan där frekvensen är lägre.

Reassociate - Främjar oberoende ordning på uttryck.

New GVN - Optimeringen garanterar att värden hamnar i samma kongruensklass för varje körning av programmet.