
Faculteit Toegepaste Wetenschappen

Academiejaar 2004–2005

# JikesRVM voor een 64-bit x86 platform

Frederik De Schrijver

Promotor: Prof. dr. K. De Bosschere
Scriptiebegeleiders: A. Georges en K. Venstermans

Scriptie voorgedragen tot het behalen van de graad van
Burgerlijk Ingenieur in de Computerwetenschappen

# JikesRVM voor een 64-bit x86 platform

Frederik De Schrijver

Scriptie voorgedragen tot het behalen van de graad van
Burgerlijk Ingenieur in de Computerwetenschappen

Academiejaar 2004-2005

Promotor: Prof. dr. K. De Bosschere
Begeleiders: Andy Georges, Kris Venstermans
Faculteit Toegepaste Wetenschappen
Universiteit Gent

Vakgroep: Elis
Voorzitter: Prof. dr. J. Van Campenhout

## Samenvatting

JikesRVM is een open-source virtuele machine voor de uitvoering van java byte-code. x86-64 is een 64-bit uitbreiding op de populaire x86 instructieset. Om JikesRVM werkende te krijgen op een x86-64 platform zijn aanpassingen nodig aan de broncode.

**Trefwoorden:** x86-64, AMD64, EM64T, Java, JikesRVM, porteren

Gent, Juni 2005

De promotor

Prof. dr. ir. K. De Bosschere

De begeleiders

Andy Georges - Kris Venstermans

De auteur

Frederik De Schrijver

# Woord vooraf

*Allereerst wil ik Prof. K. De Bosschere bedanken voor het aanreiken van dit bijzonder interessant onderwerp. Verder wil ik ook mijn begeleiders Andy Georges en Kris Venstermans bedanken om mij op weg te zetten met en te begeleiden in het grote kluwen dat de broncode van JikesRVM is.*
*Mijn oprechte dank gaat ook uit naar Julie Van den Ostende en Koen De Keyser voor het herhaaldelijk herlezen van deze scriptie om ze zo foutloos mogelijk te krijgen.*
*Tot slot wil ik ook mijn ouders bedanken voor hun steun doorheen mijn ganse studietijd.*

*Frederik De Schrijver*
*Gent, 1 juni 2005*

# Inhoudsopgave

# Lijst van figuren

# Lijst van tabellen

# Hoofdstuk 1

# Inleiding

JikesRVM is een open-source virtuele machine voor de uitvoering van java byte-code. Ze wordt ontwikkeld door onderzoekscentra en universiteiten over de gehele wereld. Door de open structuur van JikesRVM is het mogelijk uitbreidingen en aanpassingen te schrijven om allerlei fenomenen, die te maken hebben met de uitvoering van java code, te bestuderen. Zoiets zou met de virtuele machine - van bijvoorbeeld Sun - een stuk moeilijker gaan.

Om het gedrag van java code op verschillende platformen te bestuderen moet de virtuele machine aangepast worden aan deze specifieke platformen. Op dit moment werkt JikesRVM enerzijds op AIX, PPC, PPC64 (Linux en Mac OS X) en op x86 (Linux) anderzijds. Het zou goed zijn JikesRVM ook te kunnen draaien op de x86-64, de 64-bit uitbreiding op x86.

De relatief nieuwe x86-64 processoren breiden de x86 instructieset uit zodat deze ook kan werken met woordbreedten van 64-bit. Het grote voordeel van deze reeks processoren is dat ze in hardware ook gewone 32-bit x86 code kunnen uitvoeren. Voor het gebruik van de 64-bit woordbreedte moeten programma's echter aangepast worden. Bij de meeste programma's kunnen deze aanpassingen gebeuren door de broncode te compileren met een aangepaste x86-64 compiler. Bij JikesRVM zit de compiler echter ingebouwd in de eigen code, en zal het aanpassen enig programmeerwerk omvatten.

In een eerste deel wordt de belangrijkste theorie besproken die nodig is om te begrijpen waarom bepaalde aanpassingen nodig zijn. Allereerst zal gekeken worden hoe de x86-64 architectuur in elkaar zit, en waar de verschillen liggen met x86-32. Vervolgens wordt iets dieper ingegaan op de wereld van java, en de plaats die de java virtuele machine daarin inneemt. Tenslotte wordt gekeken op welke manier JikesRVM een dergelijke virtuele machine implementeert.

In een tweede deel worden de aanpassingen besproken die nodig zijn om JikesRVM op x86-64 aan de praat te krijgen. Bij de aanpassingen wordt ook telkens besproken hoe deze aanpassing werd doorgevoerd op de broncode van JikesRVM.

# Deel I

# Theorie

# Hoofdstuk 2

# x86-64

De x86-64 architectuur is ontworpen als uitbreiding op de populaire x86 architectuur. Met een nieuwe mode en een nieuwe prefix wordt de wereld van de 64-bit technologie ook voor de x86 gebruiker beschikbaar gesteld.

De volledige specificatie van de x86-64 is bijzonder uitgebreid. AMD bezorgt alle informatie in vijf handleidingen: [1], [2], [3], [4] en [5].

Bij Intel houdt men het bij 2 volumes: [6] en [7]. Intel legt wel de nadruk op het feit dat hun implementatie een uitbreiding is op de oude IA-32[1] architectuur. Een deel van de informatie die AMD in hun handleidingen opneemt dient dus bij Intel gezocht te worden in de handleidingen van de IA32 architectuur.

In de opeenvolgende onderdelen worden de belangrijkste verschillen met de oude x86-32 architectuur uit de doeken gedaan. De nadruk zal vooral liggen op de hexadecimale voorstelling van de instructies. Voor een beschrijving die zich meer richt op de eindgebruiker wordt verwezen naar [8].

## 2.1 Geschiedenis

In 1999 maakte AMD[2] de ontwikkeling van de x86-64 architectuur bekend. In tegenstelling tot de Itanium[3] architectuur van Intel is de x86-64 architectuur achterwaarts compatible met de x86 architectuur. Op een x86-64 processor blijft het mogelijk in hardware x86-32 programma's te draaien. Voor de IA-64 is dat alleen in software mogelijk. Oorspronkelijk zag Intel geen reden om ook ondersteuning te bieden voor een dergelijke architectuur en werkte voornamelijk verder aan de IA-64. In 2002 echter zorgt Intel ervoor dat ze in het bezit komt

[1] Intel Architecture
[2] Advanced Micro Devices
[3] IA-64

van het gebruiksrecht op de x86-64 architectuur en start ook de ontwikkeling van een eigen implementatie.

Op dit moment verkopen zowel AMD als Intel producten met ondersteuning voor de x86-64 architectuur. De producten van AMD gaan door het leven onder de naam AMD64. Intel koos ervoor haar producten geen nieuwe naam te geven, maar de ondersteuning voor x86-64 te zien als een uitbreiding. De uitbreiding gaat door het leven onder de naam EM64T[4]. In het verder verloop van dit werk zal de naam x86-64 gebruikt worden om geen verwarring te stichten met een specifieke implementatie.

In [9] wordt deze geschiedenis iets meer uitgediept. Daarnaast wordt ook een overzicht gegeven van de plaats van deze processoren, zowel op server- als op desktopvlak. Zowel op server- als desktopvlak scoort AMD zeer goed met zijn 64-bit processoren. Nu ook alle nieuwe Intel processoren ondersteuning bieden voor de x86-64 architectuur zal deze een nog belangrijkere rol gaan spelen op de markt.

## 2.2 Nieuwe kenmerken

Zoals eerder vermeld, werd de x86-64 architectuur ontwikkeld als uitbreiding bovenop de x86-32 architectuur. Deze uitbreidingen zijn:

- Registeruitbreidingen
  - 8 nieuwe general-purpose registers
  - Alle 16 general-purpose registers zijn 64-bit breed
  - 8 nieuwe 128-bit XMM registers
  - Uniforme byte-register adressering voor alle general-purpose registers
  - Een nieuwe instructie prefix (REX) om de uitgebreide registers aan te spreken
- Long mode
  - Tot 64-bits aan virtuele adressen
  - 64-bit instructie-pointer
  - Nieuwe instructie-pointer-relatieve data-adresseermode
  - Een platte adresruimte

[4]Extended Memory 64 Technology

## 2.3 Modes

Om zowel achterwaarts compatibel te zijn met oude besturingssystemen, als ten volle gebruik te kunnen maken van nieuwe mogelijkheden, bezit de x86-64 verschillende modes waarin hij opgestart of gebruikt kan worden. Op Figuur 2.1 wordt een overzicht gegeven van de verschillende modes met hun kenmerken.

<table>
<tr><th colspan="2" rowspan="2">Operating Mode</th><th rowspan="2">Operating System Vereist</th><th rowspan="2">Programma Hercompilatie Vereist</th><th colspan="2">Standaard</th><th rowspan="2">Register Uitbreidingen</th><th>Typical</th></tr>
<tr><th>Adres Breedte (bit)</th><th>Operand Breedte (bit)</th><th>GPR Width (bit)</th></tr>
<tr><td rowspan="3">Long Mode</td><td>64-bit Mode</td><td rowspan="2">Nieuw 64-bit OS</td><td>Ja</td><td>64</td><td rowspan="2">32</td><td>Ja</td><td>64</td></tr>
<tr><td rowspan="2">Compatibility Mode</td><td rowspan="2">Nee</td><td>32</td><td rowspan="2">Nee</td><td>32</td></tr>
<tr><td>16</td><td>16</td><td>16</td></tr>
<tr><td rowspan="4">Legacy Mode</td><td rowspan="2">Protected Mode</td><td rowspan="3">Legacy 32-bit OS</td><td rowspan="4">Nee</td><td>32</td><td>32</td><td rowspan="4">Nee</td><td rowspan="2">32</td></tr>
<tr><td>16</td><td>16</td></tr>
<tr><td>Virtual-8086 Mode</td><td>16</td><td>16</td><td>16</td></tr>
<tr><td>Real Mode</td><td>Legacy 16-bit OS</td></tr>
</table>

Tabel 2.1: Operating Modes

De modes zijn ingedeeld in twee categorieën:

**Long mode** Voor nieuwe 64-bit besturingssystemen

**Legacy mode** De modes van de gewone 32-bit x86 architectuur

Legacy mode bevat gewoon de oude x86 modes zoals we die kennen van de x86 processoren. De nieuwe x86-64 processoren kunnen dus perfect alle vroegere x86 besturingssystemen draaien. Hiervoor dient men gewoon in een van de Legacy sub-modi op te starten. Dit is bijzonder handig, want na al die jaren hebben deze besturingssystemen een hoge graad aan stabiliteit en betrouwbaarheid bereikt. Men kan dan als 2de besturingssysteem nog het oude 32-bit besturingssysteem gebruiken tot alle nodige programma's naar het nieuwe zijn herschreven.

Indien men toegang wil krijgen tot de nieuwe mogelijkheden moet men opstarten in Long mode. Long mode bevat twee sub-modi:

**64-bit mode** Voor het gebruik van de 64-bit extensies

**Compatibility mode** Om oude x86 programma's zonder hercompilatie te draaien op het nieuwe platform

In tegenstelling tot de sub-modi van de Legacy mode kan men niet in de sub-modi van de Long mode opstarten. Men start gewoon in een Long mode een nieuw 64-bit besturingssysteem

op. Dit besturingssysteem draait in 64-bit mode en kan dus volop gebruik maken van de uitbreidingen. Een gevolg hiervan is ook dat een 64-bit driver beschikbaar moet zijn voor de onderdelen van het systeem. De afwezigheid van een 64-bit driver is een van de redenen waarom het overschakelen van 32-bit naar 64-bit op een bepaald systeem niet altijd een succes is.

Verschillende linux, unix, bsd distributies, evenals Windows XP zijn beschikbaar voor deze mode. In de toekomst zullen ook andere Windows versies volgen.

Om programma's in 64-bit mode te draaien moeten ze gecompileerd zijn met een x86-64 compiler. Ook deze programma's kunnen dan gebruik maken van alle nieuwigheden. Omdat het aanpassen van een programma van x86-32 naar x86-64 niet altijd van een leien dakje loopt, en om de gebruiker in staat te stellen om oude 32-bit programma's te blijven gebruiken, heeft Long mode de sub-mode Compatibility mode. In deze mode kunnen - in hardware - 32-bit programma's uitgevoerd worden. Voor statisch gelinkte programma's vormt dit nooit een probleem. Voor dynamisch gelinkte programma's moeten ook alle gelinkte onderdelen in 32-bit aanwezig zijn.

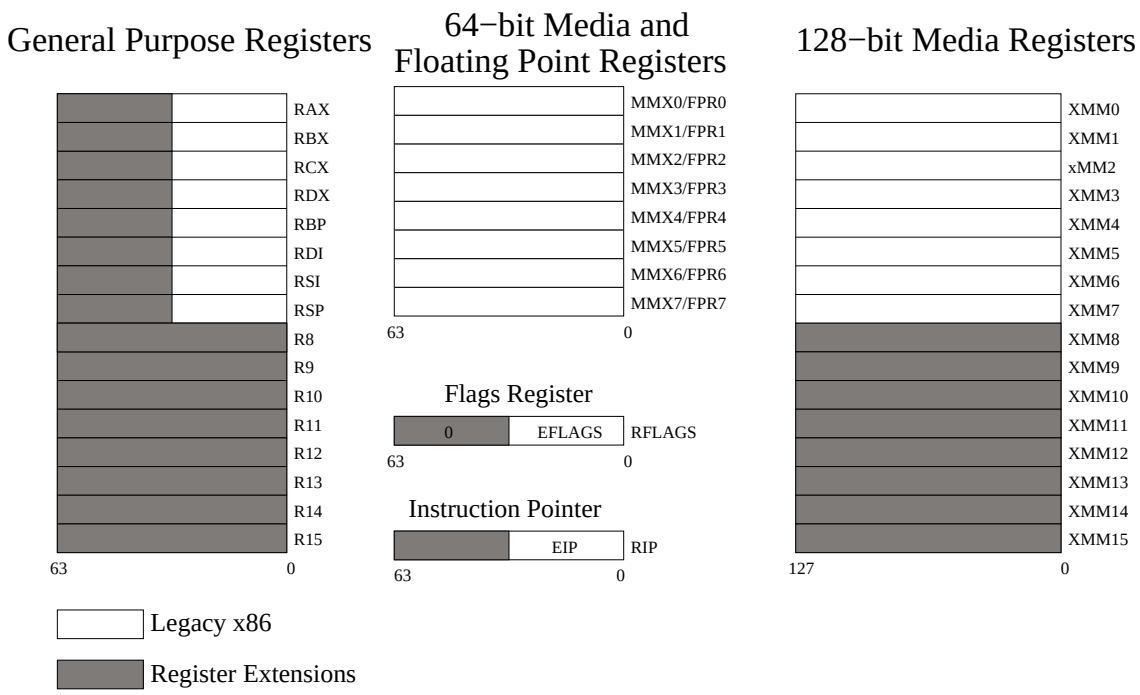
## 2.4 Registeruitbreidingen

De verschillen tussen de registers van de x86- en x86-64 architectuur zijn weergegeven op Figuur 2.1. Naast een verdubbeling van het aantal general-purpose- en 128-bit media registers, zijn de general-purpose registers nu ook 64-bit breed. Let ook op de nieuwe naamgeving voor de general-purpose registers. De eerste E in de benaming van de general-purpose registers is overal vervangen door R.

Afhankelijk van de opbouw van de instructie kan men van een general-purpose register ook het 8-bit, 16-bit of 32-bit gedeelte aanspreken. Op dit vlak is er een verschil tussen x86-32 en x86-64. In 32-bit kon men, de minst beduidende en de op een na minst beduidende byte, rechtstreeks opvragen van EAX, EBX, ECX en EDX. Deze bytes kregen dan de naam AL, BL, CL en DL respectievelijk AH, BH, CH en DH. Voor de andere registers kon men alleen het 16-bit minst beduidende gedeelte en het volledig register aanspreken.

In 64-bit mode werd de uniforme byte-register adresseermode geïntroduceerd. Deze laat toe van elk register het minst beduidende 8-, 16-, 32- en 64-bit gedeelte aan te spreken. In Tabel 2.2 worden de benamingen van de registers weergegeven, afhankelijk van welk gedeelte aangesproken dient te worden. Ook AH, BH, CH, en DH blijven aanspreekbaar, weliswaar niet als onderdeel van de uniforme byte-register adresseermode. Indien men wenst te spreken over een register, maar men niet wil specifiëren welke operandbreedte men wenst te gebruiken, gebruikt men een benaming met een kleine letter r[5].

[5] rAX, rBX, rCX, ..., r8, ...



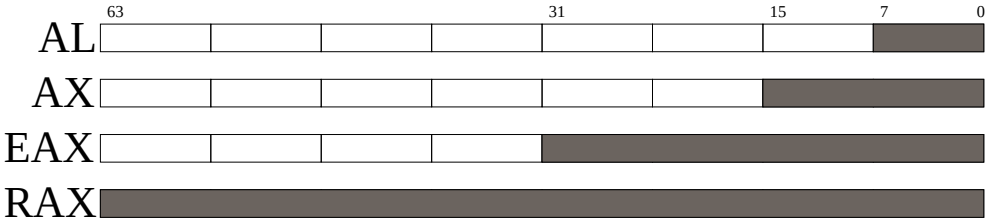

Figuur 2.1: Registeruitbreidingen

Op welke manier men de instructies moet opbouwen om al deze verschillende onderdelen aan te spreken wordt verder uitgewerkt in paragraaf 2.6.1.

In Figuur 2.2 worden voor rAX de verschillende mogelijkheden van de uniforme byte-register adresseermode weergegeven. Voor alle andere general-purpose registers krijgt men een gelijkaardige figuur.

## 2.5 Flat Adress Space

In 64-bit mode worden geen segmenten meer gebruikt. De adresruimte is een groot blok waarbinnen gewerkt wordt. In Figuur 2.3 en Figuur 2.4 wordt dit geïllustreerd. Ze vergelijken de x86-32 situatie met de nieuwe x86-64 situatie. DS, ES en SS worden absoluut niet




Figuur 2.2: Uniforme byte-register adressering voor rAX

| | | register | | | |
|---|---|---|---|---|---|
| REX-bit | reg-veld | 64-bit | 32-bit | 16-bit | 8-bit |
| 0 | 000 | RAX | EAX | AX | AL |
| 0 | 001 | RCX | ECX | CX | CL |
| 0 | 010 | RDX | EDX | DX | DL |
| 0 | 011 | RBX | EBX | BX | BL |
| 0 | 100 | RSP | ESP | SP | SPL |
| 0 | 101 | RBP | EBP | BP | BPL |
| 0 | 110 | RSI | ESI | SI | SIL |
| 0 | 111 | RDI | EDI | DI | DIL |
| 1 | 000 | R8 | R8D | R8W | R8B |
| 1 | 001 | R9 | R9D | R9W | R9B |
| 1 | 010 | R10 | R10D | R10W | R10B |
| 1 | 011 | R11 | R11D | R11W | R11B |
| 1 | 100 | R12 | R12D | R12W | R12B |
| 1 | 101 | R13 | R13D | R13W | R13B |
| 1 | 110 | R14 | R14D | R14W | R14B |
| 1 | 111 | R15 | R15D | R15W | R15B |

Tabel 2.2: Registermapping

meer gebruikt, CS, FS en GS soms, maar voor heel zeldzame gevallen. CS, FS en GS zijn trouwens eenzelfde register geworden, wat zich in de header-bestand ucontext.h[6] resulteert in een verwijzing hiernaar: `REG_CSFSGS`.

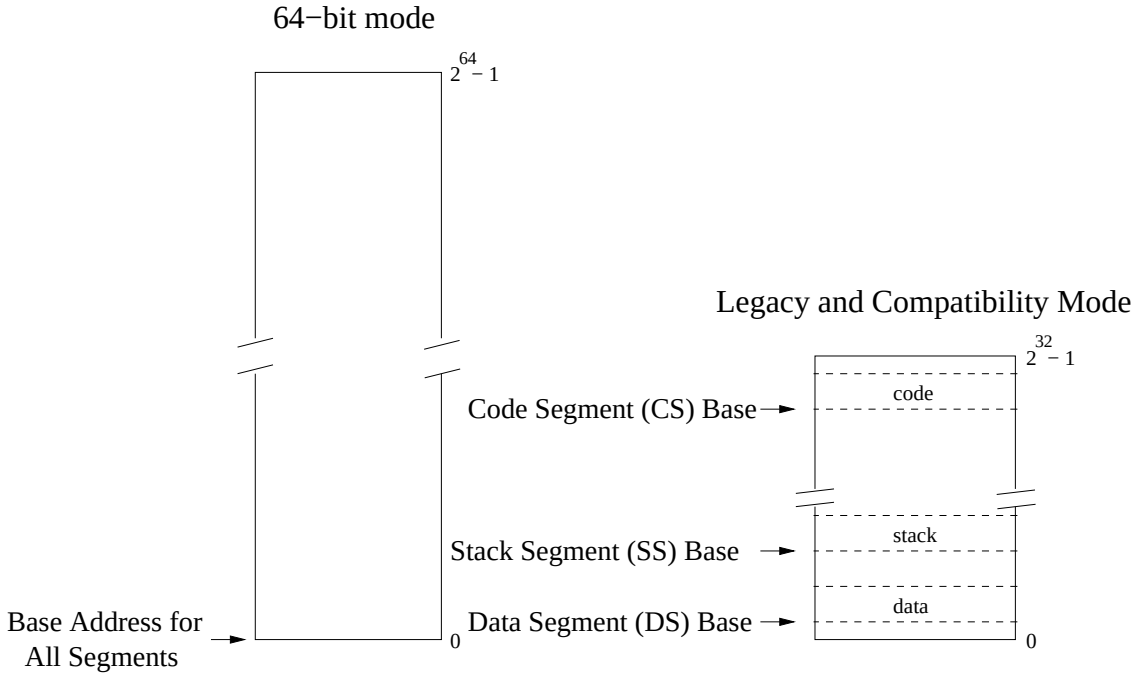
## 2.6 Instructieset

Voor de implementatie van JikesRVM is kennis vereist van de opbouw van instructies op het laagste niveau. De instructies dienen door JikesRVM zelf byte per byte gevormd te worden. In de hierop volgende onderdelen worden die delen van de instructieset besproken die voor de implementatie belangrijk zijn.

### 2.6.1 Instructie-opbouw

De x86-64-instructieset maakt gebruik van instructies met variabele lengte. Een instructie is minimaal 1 byte en maximaal 15 byte lang. De algemene opbouw is weergegeven in Figuur 2.5. Elk blokje stelt een byte voor. Een instructie wordt opgebouwd door het volgen van

[6] /usr/include/ucontext.h




Figuur 2.3: Adresruimte

een van de pijlen doorheen de figuur. Sommige onderdelen zijn optioneel, sommige kunnen meermaals voorkomen en sommige kunnen niet voorkomen zonder andere.

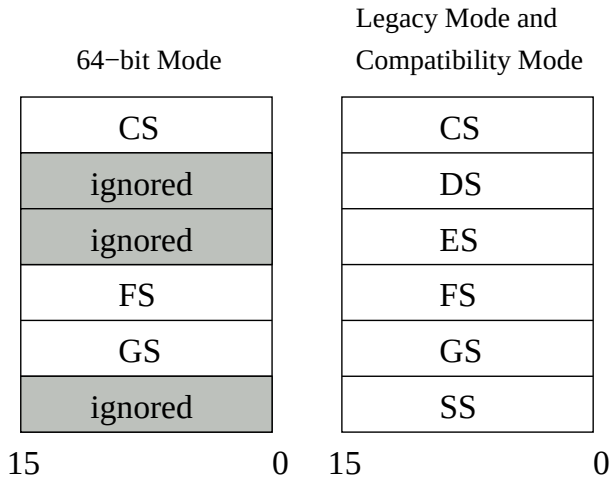
Een instructie begint met maximaal 4 legacy prefixen. In theorie kan men combinaties genereren van meer dan 4 prefixen. Voor een dergelijke combinatie zal echter minstens een instructie de werking van een andere teniet doen, zodat met 4 prefixen hetzelfde resultaat kan bekomen worden. Bij overmatig gebruik van prefixen, zou de totale lengte van de instructie voorbij 15 bytes kunnen gaan en dit leidt tot een 'general protection exception'.

In 64-bit mode kan ook de REX-prefix gebruikt worden. De REX-prefix moet altijd net voor de instructie opcode komen. Op de REX-prefix wordt dieper ingegaan in paragraaf 2.6.2.

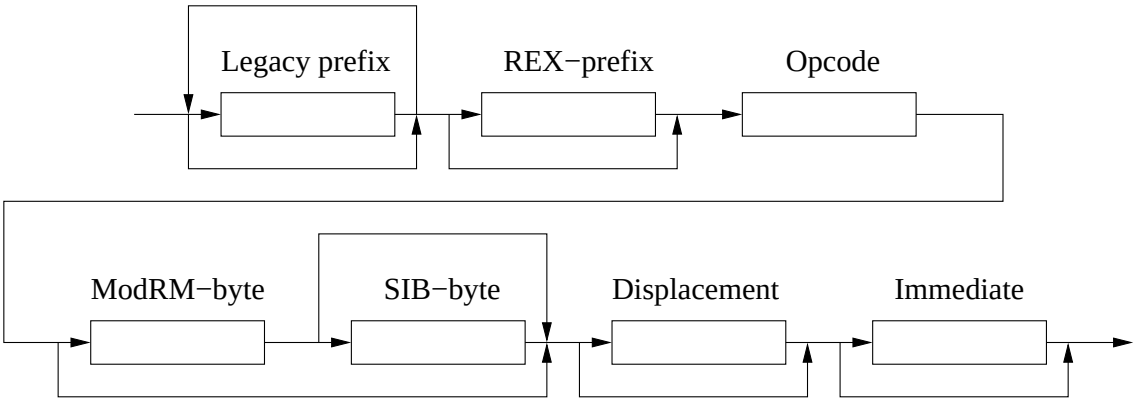
De opcode is een sequentie van minimaal 1 en maximaal 3 bytes. De opcode is de kern van de instructie en encodeert zijn doel. De opcodes kunnen opgesplitst worden in verschillende groepen:

**general-purpose** De general-purpose instructies (x86) zijn instructies voor algemeen gebruik. Instructies voor geheugenbewerkingen, controleverloop, bewerkingen op gehele getallen behoren tot deze categorie. Deze instructieset is de originele instructieset van de x86-processoren. De hierop volgende instructiesets zijn uitbreidingen hiervan.

**x87** De x87 floating point uitbreiding dient voor ondersteuning van reële getallen.




Figuur 2.4: Segmentregisters




Figuur 2.5: Registerencodering op x86-64

**MMX** Multi-Media Extensions. Uitbreidingen om snel bewerkingen te doen op bytes, ook in parallel.

**SSE/SSE2** Streaming SIMD Extenions. Uitbreidingen om in parallel bewerkingen toe te laten op 4 floats of 2 longs.

**3DNow!** Uitbreiding specifiek van AMD.

**SSE3** Uitbreiding specifiek voor Intel. Zal in de toekomst ook door AMD in hun producten geïntegreerd worden.

Omdat de laatste twee niet beschikbaar zijn voor alle x86-64 platformen is het geen goed idee deze in een algemene x86-64 versie van JikesRVM te verwerken. Op 32-bit kan men er ook

[6]Single Instruction, Multiple Data

alleen maar van uitgaan dat de x86-instructieset met x87 uitbreidingen aanwezig is. Op de x86-64 systemen kan zonder problemen x86, x87, MMX en SSE/SSE2 gebruikt worden.

Om aan te duiden met welke registers een instructie werkt zijn twee systemen voorhanden. Het meest gebruikte systeem is de ModRM-byte. Voor sommige instructies bestaat echter een mode om het te gebruiken register rechtstreeks in de opcode op te nemen. Dit gebeurt door als opcode de som van de opcode en de laagste 3 bits van het registernummer - overeenkomstig met de tweede kolom van Tabel 2.2 - te nemen. De rol van het REX-bit wordt later besproken.

Een voorbeeld van een dergelijk instructie is PUSH. PUSH RBX kan geëncodeerd worden in een byte: 0x50 + 0x3 = 0x53.

In alle andere gevallen dient een ModRM-byte gebruikt te worden. Een ModRM-byte kan eventueel ook gevolgd worden door een SIB-byte. Deze laatste kan echter niet op zichzelf bestaan. De ModRM-SIB combinatie wordt gebruikt om de vele adresseermodi die de x86 instructieset rijk is, te encoderen. Op de ModRM- en SIB-byte wordt verder ingegaan in paragraaf 2.6.2.

Vervolgens kan nog een displacement worden meegegeven. Een displacement is een vaste verplaatsing ten opzichte van een berekend adres. De displacement maakt op zich ook deel uit van de adresseermodes met de ModRM- en SIB-byte.

Uiteindelijk kan ook nog een constante worden meegegeven om in de instructie te gebruiken. Bij de meeste instructies is deze constante maximaal 32-bit. Alleen een 64-bit MOV is beschikbaar om 64-bit constanten in het systeem te laden.

### 2.6.2 REX-prefix

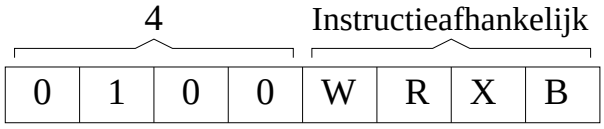
De afkorting REX staat voor Register Extensions. De REX-prefix dient dan ook om de registers op 2 manieren uit te breiden.

- Het aantal registers general-purpose en 128-bit media registers wordt uitgebreid van telkens 8 naar telkens 16.
- Bij de instructies waar de operandbreedte standaard 32-bit is breidt hij deze uit tot 64-bit.

De REX-prefix is enkel bruikbaar in 64-bit mode. Als deze gebruikt wordt moet hij onmiddellijk voor de opcode geplaatst worden. Afhankelijk van zijn rol bezit de REX-prefix een waarde tussen de 0x40 en de 0x4F. In Figuur 2.6 wordt de opbouw van het REX-prefix per bit voorgesteld. Op x86-32 komt de range 0x40-0x4F overeen met de opcodes voor de onmiddellijke versie van INC en DEC (opcode + register). Als opcode zijn deze dan ook ongeldig in 64-bit mode.

In de volgende sectie worden de verschillende bits besproken. In een eerste sectie wordt de W-bit besproken. Vervolgens wordt de invloed van de drie andere bits op het de ModRM- en de SIB-byte besproken. In Tabel 2.3 wordt een samenvatting gegeven van de invloed van de velden van de REX-byte op de instructie. Uiteindelijk wordt ook nog de invloed van de B-bit buiten de ModRM- en SIB-bytes besproken.

De registernummers in Tabel 2.2 worden vanuit het opzicht van de REX-prefix weergegeven. Op zich kan men ze nummeren van 0 tot en met 15. In code geschreven in een taal op hoger niveau dan de machinecode, kunnen ze ook op deze manier weergegeven of gebruikt worden. In de instructie zullen ze echter altijd gebruikt worden zoals de Tabel 2.2. De drie minst beduidende bits komen rechtstreeks in de opcode of in ModRM- of SIB-byte. De meest beduidende bit komt in de REX-prefix.




Figuur 2.6: REX-prefix

| Mnemonic | Bit Positie | Definitie |
|---|---|---|
| - | 7-4 | 0100 |
| REX.W | 3 | **0** Standaard operandbreedte<br>**1** 64-bit operandbreedte |
| REX.R | 2 | 1-bit (hoog) uitbreiding aan het ModRM reg veld, om toegang tot 16 registers toe te laten. |
| REX.X | 1 | 1-bit (hoog) uitbreiding aan het SIB index veld, om toegang tot 16 registers toe te laten. |
| REX.B | 0 | 1-bit (hoog) uitbreiding aan het ModRM r/m veld, SIB base veld, of opcode reg veld, om zo toegang tot 16 registers toe te laten. |

Tabel 2.3: REX Prefix Velden

### W-bit

In 64-bit mode is de standaard adresbreedte 64-bit. De operandbreedte is voor de meeste instructies echter standaard 32-bit. Om bij deze instructies toch 64-bit als operandbreedte

te hebben moet men in de REX-prefix de W-bit op 1 zetten. De operandbreedte bepaalt op hoeveel bits een bepaalde instructie wordt uitgevoerd. Mede dankzij de uniforme byte-adresseermode is het mogelijk bewerkingen op 8-, 16-, 32- en 64-bit uit te voeren. Dit maakt het gemakkelijk om bewerkingen op bijvoorbeeld bytes, shorts, integers, longs en zo uit te voeren met het gewenste overflow gedrag.

Het verschil in operandbreedte resulteert natuurlijk ook in een verschil in instructie. De opcode voor 16-, 32- en 64-bit is dezelfde. Voor 8-bit is het altijd een andere opcode. Het verschil tussen 16-, 32- en 64-bit wordt geëncodeerd door middel van prefixen. De meeste instructies hebben standaard een 32-bit operandbreedte. Om de operandbreedte 16-bit te maken dient een prefix gebruikt te worden: 0x66. Om dit naar 64-bit om te zetten moet de W-bit op 1 worden gezet. De W-bit heeft voorrang op het prefix 0x66.

Het is ook belangrijk te weten wat er met de rest van de inhoud van een register gebeurt, als er minder dan 64-bit in dit register opgeslagen moeten worden. Als het resultaat van een 32-bit bewerking opgeslagen wordt in een 64-bit register, worden de 32 meest significante bits op 0 gezet. Bij 8- en 16-bit operandbreedte blijven de niet aangesproken bits onveranderd.

Sommige instructies hebben standaard een 64-bit operandbreedte. Deze instructies worden weergegeven in Tabel 2.4. Bij deze instructies is het W-bit van geen belang. Er is voor deze instructies ook geen mogelijkheid, in 64-bit mode, om een instructie met 32-bit operandbreedte te vormen.

| | | |
|---|---|---|
| CALL (Near) | LOOP | POPFQ |
| ENTER | LOOPcc | PUSH imm8 |
| Jcc | LTR | PUSH imm32 |
| JrCXZ | MOV CR(n) | PUSH reg/mem |
| JMP (Near) | MOV DR(n) | PUSH reg |
| LEAVE | POP reg/mem | PUSH FS |
| LGDT | POP reg | PUSH GS |
| LIDT | POP FS | PUSHFQ |
| LLDT | POP GS | RET (Near) |

Tabel 2.4: Instructies met standaard 64-bit operandbreedte

### ModRM-byte en SIB-byte

De ModRM- (eventueel in combinatie) met een SIB-byte wordt gebruikt om de verschillende combinaties van adresseren van registers en geheugen binnenin een instructie te encoderen.

Vooreerst dient er onderscheid gemaakt te worden tussen drie types instructies.

- Instructies met 2 operands
- Instructies met 1 operand
- Instructies zonder operand

Bij een instructie met 2 operands is een van de twee operands altijd de inhoud van een register. Het andere kan een complexere vorm aannemen. Volgende vormen zijn beschikbaar:

register

[register]

[register + displacement]

[index<<scale+disp]

[displacement]

[base+index<<scale+disp ]

De modes tussen vierkante haakjes berekenen het adres van wat tussen haakjes staat en nemen hetgeen op dit adres staat als operand. Afhankelijk van de opcode is het eerste of het tweede argument van de bewerkingen de complexere vorm. Indien het resultaat opgeslagen dient te worden, wordt het opgeslagen in het eerste argument. Dit kan dus ook rechtstreeks in het geheugen zijn.

Een opmerking dient geplaatst te worden bij [displacement]. In 32-bit mode is dat een rechtstreeks adres. In 64-bit mode wordt dit gedecodeerd als [RIP+displacement]. Dit is de nieuwe instructiepointer-relatieve adresseermode.

Instructies met een operand gebruiken als operand de complexere mode. Omdat het gedeelte, om het andere argument te encoderen in de ModRM-byte, dan niet nodig is voor de adressering wordt dit gebruikt om met een opcode verschillende instructies overeen te laten komen. Dus afhankelijk van het reg-gedeelte van het ModRM-byte kan eenzelfde opcode een verschillende betekenis krijgen.

Bij instructies zonder operand is alleen de opcode nodig, en geen ModRM-byte.

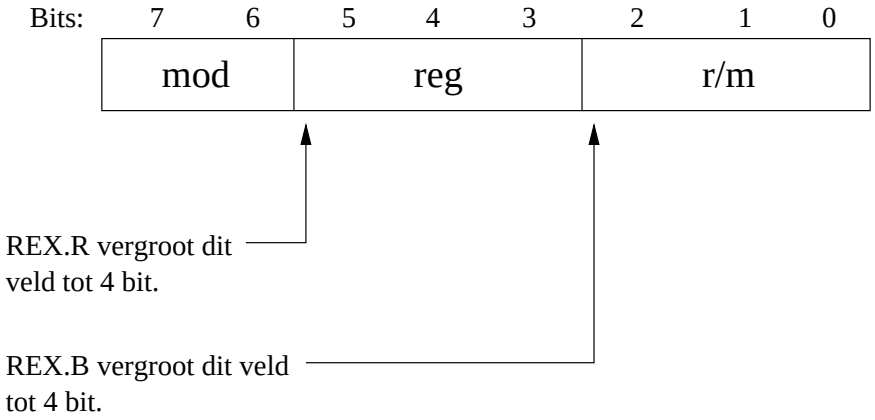
Een ModRM-byte bestaat uit drie delen:

**mod** Mode

**reg** Register

**r/m** Register/Memory

Figuur 2.7 toont hoe een ModRM-byte samengesteld is voor de x86-instructieset. De pijlen tonen de invloed van het REX.X- en REX.B-bit. Voor de x86-64-instructieset blijft het uitzicht van het byte hetzelfde. In het decoderen van de instructie wordt het reg en r/m veld, verlengd met REX.R, respectievelijk REX.B als meest significante bit. Dit laat toe in deze velden gebruik te maken van de 16 registers. De reden dat deze bit vooraan in een prefix wordt gestopt is dat het ModRM-byte anders geen byte meer zou zijn, maar 10 bit breed.




Figuur 2.7: ModRM-byte

Voor de SIB-byte gaat ongeveer hetzelfde op als voor de ModRM-byte.

Een SIB-byte bestaat uit drie delen:

scale

index

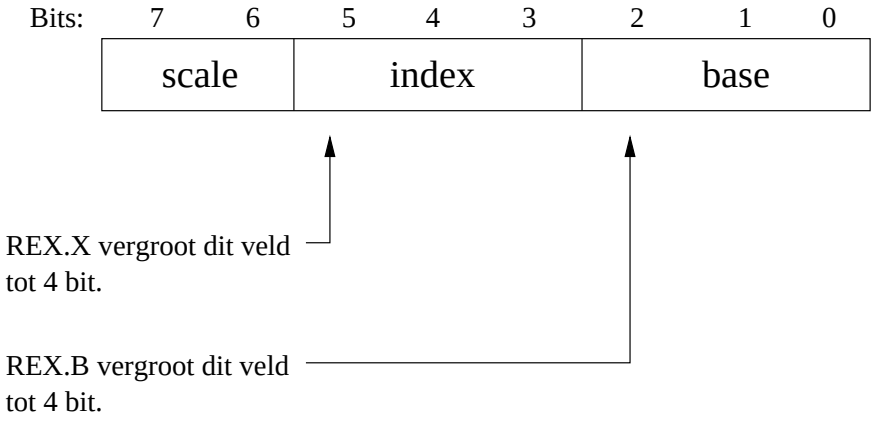
base

Figuur 2.8 toont hoe een SIB-byte samengesteld is voor de x86-instructieset. Het heeft dus dezelfde vorm als het ModRM-byte. Alleen werkt op het index-veld de REX.X-bit in, in plaats van de REX.R-bit bij de ModRM-byte. Dezelfde opmerkingen gelden als bij de ModRM-byte.

Hoe men de ModRM- en SIB-bytes exact moet opbouwen, is te vinden in Appendix 3 van [3].

### B-bit

Voor de instructies, die gebruikt kunnen worden in de vorm `opcode + register`, kunnen alleen maar de 3 minst significante bits van het registernummer gebruikt worden, omdat anders overlap zou kunnen optreden met andere opcodes. Daarom wordt de meest significante bit geëncodeerd in de REX-prefix. De B-bit wordt bij de decodering van de instructie gebruikt om het register te bepalen.




Figuur 2.8: SIB-byte

Als men dus PUSH R11 wil doen, krijgt men *0x41 0x53* in plaats van *0x40 0x53* voor PUSH RBX. Indien men de volledige 4-bit van het registernummer had opgeteld bij *0x50* krijgt men *0x5B* wat overeen komt met POP RBX.

### 2.6.3 Instructies ongeldig in 64-bit mode

Bepaalde instructies uit de x86-instructieset zijn niet meer geldig in 64-bit mode. Volgende instructies mogen niet meer gebruikt worden in 64-bit mode.

**ASCII aanpassingen** AAA (0x37), AAD (0xD5), AAM (0xD4), AAS (0x3F)

**ARPL** Opcode gebruikt als MVSXD (0x63)

**BOUND** (0x62)

**CALL** 0x9A versie

**Tiendelige aanpassingen** DAA (0x27), DAS (0x2F)

**DEC** in de range 0x48-0x4F, wegens overlap met REX-prefix

**INC** in de range 0x40-0x47, wegens overlap met REX-prefix

**INTO** (0xCE)

**JMP** 0xEA versie

**Load Far Pointer** LDS (0xC5), LES (0xC4)

**POP** POS DS (0x1F), POP ES (0X07), POS SS (0x17)

**PUSH** PUSH DS (0x1E), PUSH ES (0x06), PUSH SS (0x16)

**Push All** PUSHA , PUSHAD (60)

**SAHF en LAHF**

**System** SYSENTER (0x0F 0x34), SYSEXIT (0x0F 0x35), deze zijn ongeldig in volledig Long mode

## 2.7 Parameters doorgeven

Een ingrijpende verandering werd ingevoerd voor het doorgeven van parameters tussen functies. Dit zal vooral van belang zijn bij het aanroepen van C functies vanuit de virtuele machine. Binnen de virtuele machine hoeft hier geen rekening mee gehouden te worden.

Op x86-32 worden alle parameters doorgegeven langs de stack. Bovenop de stack zit de eerste parameter van de functie, de tweede parameter daar onder, en zo verder. Op x86-64 worden ook parameters langs de stack doorgegeven.

Voor het doorgeven van gehele waarden worden general purpose registers gebruikt, voor reële waarden worden *xmm* registers gebruikt. Verwijzingen blijven langs de stack doorgegeven worden.

De eerste zes gehele waarden kunnen via general purpose registers doorgegeven worden. Daarvoor moeten volgende registers gebruikt worden:

- RDI
- RSI
- RDX
- RCX
- R8
- R9

Als er meer gehele parameters zijn, dienen deze op de stack te worden geplaatst volgens de regels van x86-32.

Voor reële waarden moeten eerst de *xmm* registers gebruikt worden. *xmm0* tot en met *xmm7* (in die volgorde) dienen gebruikt te worden voor de eerste 8 reële parameters. Indien er meer parameters zijn dienen deze opnieuw op de stack te worden geplaatst. Bij het gebruik van *xmm* registers mag niet vergeten worden dat *al* een bovengrens voor het het aantal *xmm* registers, dat gebruikt wordt om reële parameters door te geven, moet bevatten.

Meer details omtrent deze conventie zijn te vinden in [10].

## 2.8 Implementaties

Op dit moment bieden zowel AMD als Intel processoren aan met ondersteuning voor x86-64. De desktopproducten van AMD gaan door het leven onder de naam Athlon64, de server producten onder de naam Opteron. Deze laatste kunnen voorkomen in configuraties tot en met 8 processoren.

Intel heeft ervoor gekozen x86-64 te zien als een uitbreiding op hun vroegere producten. Op de desktop bezit Pentium 4 van de 6xx reeks deze uitbreidingen. Ook sommige van het nieuwste Celeron-D processoren, Intel's budgetprocessoren, bezitten de ondersteuning voor EM64T. De Xeon reeks bezit al een tijdje ondersteuning voor x86-64.

Een vergelijking tussen de verschillende beschikbare processoren wordt gemaakt in [11].

De belangrijkste vraag is natuurlijk welke verschillen deze twee implementaties bevatten. Wat zeker niet mag gebruikt worden is 3DNow! en SSE3 omdat het eerste alleen door AMD en het tweede voorlopig alleen door Intel geïmplementeerd wordt.

Volgens de faq van Intel in verband met EM64T zal software die geen gebruik maakt van de extra's zoals SSE3, Hyperthreading of 3DNow! vermoedelijk werken op zowel AMD64 als EM64T. Het woordje *likely* wijst er echter op dat er ergens nog een addertje onder het gras kan zitten.

[6] `http://developer.intel.com/technology/64bitextensions/faq.htm`

# Hoofdstuk 3

# De Java Virtuele Machine

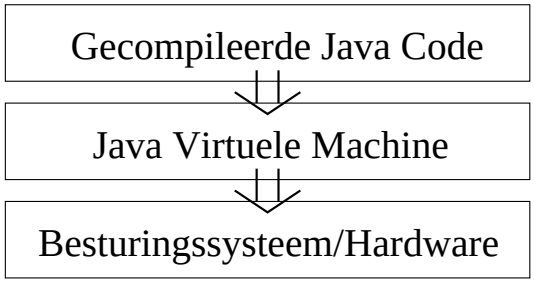
In dit hoofdstuk wordt de Java Virtuele Machine (JVM) beschreven. Allereerst wordt de Java Virtuele Machine geplaatst binnen de wereld van het schrijven en uitvoeren van Java programma's. Vervolgens wordt de opbouw van de JVM bekeken, dit vanuit de Java specificaties. Tenslotte wordt ingegaan op de instructieset van de JVM. Een belangrijk deel van het werk van een virtuele machine bestaat er namelijk in deze instructies om te zetten naar een reeks machine-instructies voor het systeem waarop de virtuele machine draait. Het grootste deel van dit werk draait rond deze omzetting.

## 3.1 Doel

Zoals de naam virtuele machine al verraadt, worden gecompileerde Java programma's niet rechtstreeks op de eigen centrale verwerkingseenheid uitgevoerd - zoals bijvoorbeeld gecompileerde C- of C++ programma's. Java programma's worden uitgevoerd door een ander programma (de virtuele machine) dat het Java programma omzet naar een uitvoerbare vorm voor de fysische processor en die vorm dan ook uitvoert. Men kan dus Java programma's zien als programma's gecompileerd voor een virtuele processor. Die processor moet dan geëmuleerd worden in software, en daarvoor dient de virtuele machine. Figuur 3.1 geeft dit grafisch weer.

Deze aanpak heeft een aantal voordelen:

- Java programma's kunnen zonder hercompilatie op elk platform uitgevoerd worden waarvoor een Java Virtuele Machine beschikbaar is.
- Er kunnen ook virtuele machines geschreven worden voor allerlei kleine apparaten zoals bijvoorbeeld GSM's. Daar kunnen dan lichtgewicht Java programma's op gedraaid worden.




Figuur 3.1: De Java Virtuele Machine

- Er kan binnen de virtuele machine afscherming worden voorzien voor bepaalde delen van het fysieke systeem, en dit afhankelijk van de vertrouwensgraad die een bepaald programma krijgt.

Een programma geschreven in Java wordt gecompileerd tot een class bestand. Dit class bestand wordt dan uitgevoerd binnen de virtuele machine. In Figuur 3.2 wordt dit proces verduidelijkt. Een klein stukje uit een Java programma wordt afgebeeld. Vervolgens wordt het resultaat getoond na compileren met IBM's Blackdown Java Compiler[1]. Als eindresultaat worden de bytes getoond die de virtuele machine hieruit zou kunnen genereren en uitvoeren op een x86-64 platform. Dit eindresultaat is niet geheel correct omdat, bij het met de hand omzetten van de Java byte-codes, de indexering van de locale variabelen niet nagekeken werd.

Voor de leesbaarheid werd, zowel voor de Java byte-code als voor de machinecode, ook in assembler de code weergegeven. Binnen de Java Virtuele Machine wordt de omweg via assembler niet gemaakt. De Java byte-code wordt instructie per instructie ingelezen en omgezet naar machinecode.

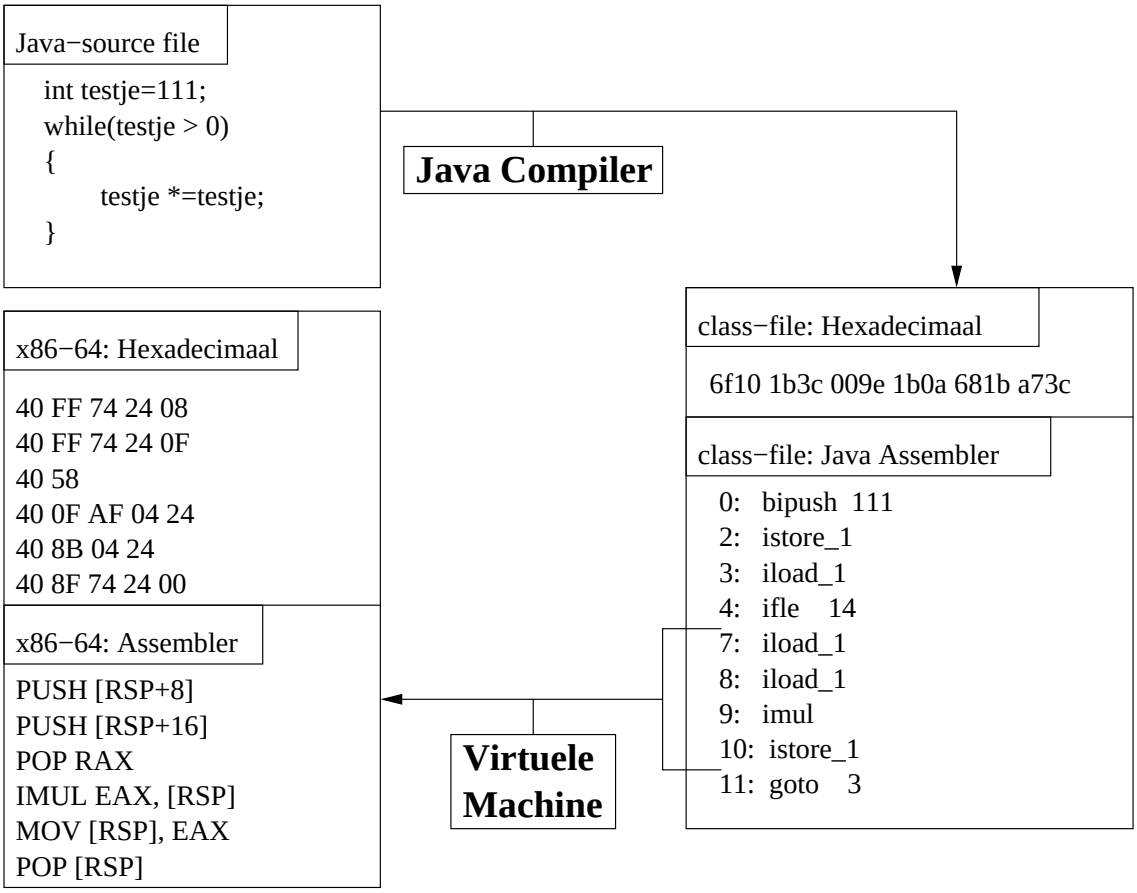
Vanaf nu zal Java byte-code gebruikt worden voor de bytes gegenereerd door een Java broncode compiler en machinecode voor de bytes gegenereerd specifiek voor een fysieke processor.

## 3.2 Opbouw van de JVM

De virtuele hardware van de JVM is opgedeeld in vier stukken:

- De registers
- De stack
- De garbage-collected heap
- Het instructiegedeelte

[1] versie: Blackdown-1.4.2-01




Figuur 3.2: Het Java-programma-model

Sommige van deze onderdelen zijn aanwezig per thread, andere zijn algemeen voor de volledige virtuele machine. De onderdelen specifiek voor de virtuele machine worden aangemaakt bij de creatie van de virtuele machine en vernietigd bij het afsluiten. Hetzelfde geldt voor de threads. Ook daar worden de onderdelen gecreëerd en vernietigd samen met de thread.

De adresbreedte van een adres binnen de JVM is 32-bits. Dit laat toe 4 GiB aan geheugen te adresseren. De stack, de heap en het instructiegedeelte liggen ergens in die 4 GiB geheugen. Elk van de registers kan een adres van 32-bit bevatten.

Het instructiegedeelte is byte-gealigneerd. Dit is het gevolg van het feit dat de Java instructies opgebouwd zijn uit een byte opcode gevolgd door eventueel nog enkele bytes als argument. De stack en de heap zijn 32-bit gealigneerd.

Elke thread heeft een instructiepointer en drie registers om de stack te beheren. De instructiepointer wijst zoals elke instructiepointer naar de huidige instructie. Als de instructie uitgevoerd is, wordt de instructiepointer verzet naar de volgende instructie. De drie andere registers wijzen naar adressen van drie blokken binnen het frame van de huidige methode:

- optop register
- vars register
- frame register

Voor elke java methode wordt op de stack een frame gealloceerd. Binnen dit frame kan men drie delen onderscheiden. Het eerste deel bevat de lokale variabelen die binnen deze methode gebruikt worden. Het begin van dit gedeelte wordt aangewezen door het vars register. Vervolgens is er ook de operand stack. De top van deze stack wordt aangewezen door het optop register. Als laatste is er het frame register. Dit verwijst naar het begin van de code van de methode, in het instructiegedeelte. Het frame register verwijst niet altijd naar de finale code. Omdat Java laat binden ondersteund kan het ook verwijzen naar een virtuele methode.

Vervolgens is er het instructiegedeelte. Dit is een onderdeel gedeeld door de volledige JVM. In het instructiegedeelte wordt alle gecompileerde code opgeslagen.

Tenslotte is er nog de heap. De heap is gemeenschappelijk voor alle threads en wordt dus gecreëerd samen met de JVM. Op de heap worden alle objecten bewaard. Elke keer als een new wordt uitgevoerd wordt een object op de heap bijgemaakt. Op het moment dat er geen referenties meer zijn naar een object op de heap wordt dit object door de garbage collector van de heap verwijderd.

## 3.3 Opbouw van de Instructieset

Om goed te begrijpen wat een virtuele machine dient te doen, is het belangrijk een goed zicht te hebben op de opbouw van de input die ze te verwerken krijgt: de class bestanden. Een class bestand heeft de volgende vaste structuur:

```
ClassFile
{
     u4               magic;
     u2               minor_version;
     u2               major_version;
     u2               constant_pool_count;
     cp_info          constant_pool[constant_pool_count-1];
     u2               access_flags;
     u2               this_class;
     u2               super_class;
     u2               interfaces_count;
     u2               interfaces[interfaces_count];
     u2               fields_count;
     field_info       fields[fields_count];
     u2               methods_count;
     method_info      methods[methods_count];
     u2               attributes_count;
     attribute_info   attributes[attributes_count];
}
```

Niet alle onderdelen van de class bestanden zijn voor dit werk van belang. Voor een diepgaande bespreking van deze onderdelen kan [12] geraadpleegd worden. Voor dit werk is vooral het gedeelte method_info belangrijk. Hierin worden alle gegevens van een bepaalde methode opgeslagen. Voor dit werk zijn vooral de Java byte-codes belangrijk die hierin opgeslagen zitten en bepalend zijn voor de uitvoering van de methode.

Volgende onderdelen worden besproken:

- De primitieve datatypes
- Variabelen binnen een klasse
- Java-bytecodes voor de uitvoering van methodes

Omdat het aanpassen van JikesRVM naar x86-64 vooral op heel laag niveau plaats vindt, met name bij de omzetting van de Java byte-code naar machinecode, wordt vooral daar de nadruk gelegd.

### 3.3.1 Datatypes

De primitieve types die Java ondersteund, zijn verschillende numerieke types, het boolean type en het returnAddress type. Bij de numerieke types zijn er twee groepen: de gehele types en de rationele types.

De gehele types zijn:

**byte** 8-bit 2-complement geheel getal

**short** 16-bit 2-complement geheel getal

**int** 32-bit 2-complement geheel getal

**long** 64-bit 2-complement geheel getal

**char** 16-bit natuurlijk getal, voornamelijk bedoeld om Unicode karakters mee voor te stellen

Voor de char werd in Java niet gekozen voor de standaard ASCII-set, maar voor Unicode omdat die ook ondersteuning biedt voor exotische tekens uit exotische talen.

De reële types zijn:

**float** Getallen uit de float verzameling of uit de float verzameling met uitgebreide exponent

**double** Getallen uit de double verzameling of uit de double verzameling met uitgebreide exponent

De Java specificatie laat, voor de reële types, zowel de standaard 32-bit en 64-bit representatie voor float, respectievelijk double, als de versies met evenveel bits voor teken en mantisse, maar met meer bits voor de exponent, toe. De definitie van float en double volgt op die manier de ANSI/IEEE Standaard 745-1985.

De waarden van het boolean type worden vertaald als waar of vals. De JVM voorziet slechts beperkte ondersteuning voor boolean types. Intern wordt de boolean op de meeste plaatsen behandeld als een byte.

Het type returnAddress wordt gebruikt als pointer naar een opcode van een JVM instructie. Het returnAddress is het enige type dat niet beschikbaar is binnen de Java programmeertaal. Een returnAddress kan vanuit de Java-code dus niet aangepast worden.

Naast de primitieve types zijn er ook nog drie referentie-types. Deze kunnen verwijzen naar objecten van het type class, array en interface. Een referentie kan ook in het niets wijzen. Dan heeft hij de waarde NULL.

### 3.3.2 De operand stack

Voor de uitvoering van instructies maakt de JVM gebruik van een stackmodel. Concreet wil dit zeggen dat elke operatie zijn argumenten van deze stack haalt en dan eventueel een resultaat boven op de stack plaatst.

Zoals elke stack bestaat ook de operand stack uit stackslots. De Java specificatie deelt de types die op deze stack geplaatst kunnen worden in twee categoriën. In Tabel 3.1 wordt voor de verschillende types weergegeven tot welke categorie ze behoren. Uit de tabel kan duidelijk afgeleid worden dat men voor categorie 1, 32-bits in gedachten had en voor categorie 2, 64-bits. Hier komt naar voren dat de specificaties geschreven zijn met een 32-bit systeem in het achterhoofd. In [13] worden de verschillende punten bekeken waar de Java specificaties 32-bit systemen bevoordelen (indien men de specificatie letterlijk volgt).

| **Type** | **Computationeel type** | **Categorie** |
|---|---|---|
| boolean | int | 1 |
| byte | int | 1 |
| char | int | 1 |
| int | int | 1 |
| float | float | 1 |
| reference | reference | 1 |
| returnAddress | returnAddress | 1 |
| long | long | 2 |
| double | double | 2 |

Tabel 3.1: Computationele types en categoriën van de JVM types

### 3.3.3 Bewerkingen

De bewerkingen die de JVM ondersteunt, kunnen in een aantal logische groepen worden onderverdeeld.

- Laden en opslaan van gegevens op de stack
- Arithmetische instructies
- Typeconversie instructies
- Instructies voor het creëren en manipuleren van objecten
- Instructies voor het management van de operand stack

- Transfer controle instructies
- Instructies voor het oproepen van methodes
- Monitor instructies

Omwille van het stackmodel moeten alle gegevens die verwerkt moeten worden eerst op de stack geladen worden. Deze gegevens komen ofwel van de lokale variabelen van de methode ofwel vanuit het instructiegedeelte. Na de verwerking moeten deze gegevens dan ook nog opnieuw kunnen opgeslagen worden, ofwel in een lokale variabele ofwel in het instructiegedeelte. Naast het laden en opslaan van scalaire types, wordt ook ondersteuning geboden voor het laden en opslaan van elementen uit arrays en objecten.

Op de stack worden alle gehele types met minder dan 4 bytes uitgebreid naar 4 bytes. Booleans zijn zowieso al opgeslagen als bytes. Bytes en shorts worden teken-uitgebreid naar 32-bit. Voor chars wordt de nuluitbreiding gebruikt.

Omwille van het feit dat alle gehele types kleiner dan de integer naar integer omgezet worden, kunnen arithmetische operaties op deze types uitgevoerd worden via de arithmetische operaties op integers. Daarom biedt een JVM enkel arithmetische operaties aan op:

- integer
- long
- float
- double

De arithmetische operaties die ondersteund worden zijn de standaard arithmetische bewerkingen.

Verder biedt de JVM ondersteuning voor volgende type conversies:

- Verbredende conversies
  - int naar long, float of double
  - long naar float of double
  - float naar double
- Versmallende conversies
  - int naar byte, short of char
  - long naar int

- float naar int of long
- double naar int, long of float

De andere groepen instructies spreken voor zichzelf en vragen geen verdere bespreking. Voor een overzicht van alle instructies wordt opnieuw verwezen naar [12].

### 3.3.4 Op byte-niveau

Omwille van de eenvoud van de manier waarop de instructieset van de JVM is opgebouwd, kan alle functionaliteit ondersteund worden door slechts een 200-tal opcodes. Aangezien men 256 verschillende opcodes kan creëren met een byte, zijn alle Java opcodes slechts een byte groot. Sommige opcodes hebben ook nog één argument. De lengte van dit argument ligt vast in de opcode.

Al deze byte-codes hebben ook een assemblernotatie. Met het programma javap, meegeleverd met bijvoorbeeld IBM's Blackdown JDK of Sun's JDK, kan de assembler van een methode weergegeven worden. Om meer inzicht te krijgen in de instructieset, kan het heel handig zijn kleine voorbeeldjes te decompileren via `javap KlasseNaam` en te volgen wat er allemaal gebeurd.

# Hoofdstuk 4

# Jikes RVM

De bedoeling van dit hoofdstuk is allereerst een inleiding te geven tot het concept van Jikes RVM. Verder wordt ook uitgelegd hoe Jikes RVM geïnstalleerd en uitgevoerd moet worden. Tenslotte wordt dieper ingegaan op die onderdelen van Jikes RVM die bestudeerd werden om aangepast te worden naar x86-64. Voor een grondiger inleiding wordt verwezen naar [14]. De enige manier om Jikes RVM echt te leren kennen is in de code duiken.

## 4.1 Korte Beschrijving

Jikes RVM (Research Virtual Machine) is een opensource virtuele machine voor de uitvoering van Java programma's. Ze wordt onderhouden en ontwikkeld door verschillende onderzoeksinstellingen en universiteiten. Jikes RVM is vooral van academisch belang, omdat door de open structuur, gemakkelijk veranderingen en uitbreidingen kunnen worden aangebracht, die kunnen helpen bij onderzoek, modelleren en evalueren van implementatietechnieken voor virtuele machines. Voor dagelijks gebruik schiet Jikes RVM nog net iets tekort. Dit komt voornamelijk omdat Jikes RVM nog niet alle klassen van de Java API ondersteund. AWT ondersteuning bijvoorbeeld ontbreekt. In [15] wordt uitgediept hoe Jikes RVM in het verleden al heeft bijgedragen tot verschillende onderzoeken.

Belangrijk is te benadrukken dat Jikes RVM de naam JVM niet mag dragen. Een product mag pas de naam JVM dragen als het slaagt in de officiële Java Test Compatibility Kit (TCK). Omdat Jikes RVM eerder een onderzoeksproject is, voert het deze test niet uit. Het doel van het Jikes RVM project is voornamelijk de gebruikers de mogelijkheid te geven om de huidige mogelijkheden van de virtuele-machine-technologie te verbeteren.

Een bijzondere eigenschap van Jikes RVM is dat deze grotendeels geschreven is in Java. De andere talen die gebruikt worden zijn C, C++, bash en assembler.

### 4.1.1 Ondersteunde platformen

Op het moment van schrijven werkt Jikes RVM op volgende platformen:

- PPC (Linux, Mac OS X en AIX)
- PPC64 (Linux en AIX)
- x86-32 (Linux)

### 4.1.2 Onderdelen

In essentie bestaat Jikes RVM uit volgende onderdelen:

- Core Runtime
- Compilers
- Memory Managers
- Adaptive Optimization System

De Core Runtime is verantwoordelijk voor beheer en onderhoud van alle onderliggende datastructuren en voor de interfaces met libraries. Dit omvat de class loader, de thread sheduler, de verifier, library support, ... De Core Runtime is de kern van Jikes RVM.

De compilers zijn verantwoordelijk voor het genereren van uitvoerbare machinecode uit Java bytecode. Voor PPC/PPC64 zijn er 4 compilers:

- Baseline compiler
- Quick compiler
- Optimizing compiler
- JNI (Java Native Interface)

Voor x86 zijn dezelfde compilers beschikbaar, op de quick compiler na.

De memory manager is verantwoordelijk voor het alloceren van geheugen voor objecten en ook voor de opruiming ervan. Verschillende Memory Managers zijn voorhanden.

Het Adaptive Optimization System is verantwoordelijk voor profiling van uitvoerende programma's en voor het aansturen - op een oordeelkundige manier - van de optimizing compiler.

Bij het bouwen van Jikes RVM uit de broncode moet een type worden opgegeven. Dit wordt gedaan door het meegeven van een string als argument van het jconfigure script. Het gemakkelijkst is hier een voorgedefiniëerd type te gebruiken: prototype, prototype-opt, development, production.

Soms is het nodig beter te specifiëren hoe Jikes RVM moet opgebouwd worden. Men kan zelf een type opbouwen via een string bestaande uit 3 onderdelen:

```
<bootImagecompiler>{ "Base" | "Adaptive" }<garbage collector>
```

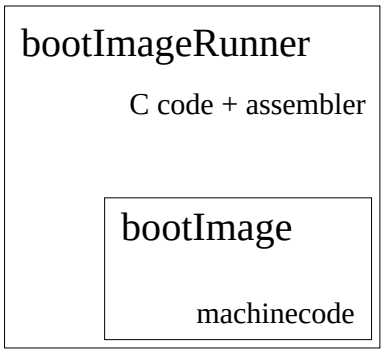
Het eerste gedeelte (bootImagecompiler) bepaalt met welke van de 4 (3) compilers de bootImage moet gecompileerd worden. Het tweede gedeelte bepaalt of in de finale versie het adaptief systeem en de optimizing compiler al dan niet aanwezig zijn. Tenslotte wordt in het laatste deel (garbage collector) gekozen welke memory manager gebruikt moet worden.

Voor details (en meer opties) wordt verwezen naar de Jikes RVM manual ([14]).

### 4.1.3 Een VM geschreven in Java

Een VM geschreven in Java zit met het volgende probleem: om de virtuele machine uit te voeren is een virtuele machine nodig. Het zou natuurlijk onzinnig zijn als Jikes RVM voor de uitvoering altijd op een VM zou moeten rekenen. Om hier een mouw aan te passen wordt tijdens het bouwproces van Jikes RVM, via een andere VM, de bootImageWriter uitgevoerd. Deze schrijft een bootImage weg op de schijf (zie paragraaf 4.3.6). Deze bootImage wordt dan gelanceerd door een programma geschreven in C, de bootImageRunner. Deze bootImageRunner zal de bootImage in het geheugen laden, ervoor zorgen dat alle register goed staan, en dan naar de eerste instructie van de bootImage springen.

In Figuur 4.1 wordt dit principe verduidelijkt.




Figuur 4.1: Een VM in Java

Initiëel is voor de bouw van Jikes RVM dus een andere VM nodig. Hier moet ook opgelet worden. Jikes RVM is heel kieskeurig op het vlak van VM om mee gecompileerd te worden.

Indien een VM niet correct de Java specificaties implementeert en fouten in de implementatie bevat, zal de VM vermoedelijk niet door het compileerproces geraken. Als opensource project zou het ook goed zijn dat Jikes RVM kon gecompileerd worden door een open source VM. Tot nog toe slaagt geen enkele opensource VM erin Jikes RVM in alle gevallen gecompileerd te krijgen. Er moet dus nog altijd terug gegrepen worden naar een niet opensource variant zoals Sun JDK of IBM Blackdown JDK.

Omwille van de grote moeilijkheidsgraad om met een VM door het compileerproces van Jikes RVM te geraken, kan Jikes RVM ook gebruikt worden als stresstest voor een virtuele machine.

## 4.2 Werken met Jikes RVM

### 4.2.1 De Broncode

De broncode van Jikes RVM is te vinden in de Downloads sectie van de website van Jikes RVM: `http://jikesrvm.sourceforge.net`. De broncode wordt aangeboden op twee manieren:

- Officiële Releases
- CVS versie

Voor dit werk werd gekozen te werken met de CVS versie, om niet achter te lopen op veranderingen in de broncode.

### 4.2.2 Installatie

De installatie van Jikes RVM is net iets lastiger dan het downloaden van de broncode en het uitvoeren van een paar make commando's.

Voorbereidend dient nagegaan te worden of de benodigde hulpprogramma's geïnstalleerd zijn. Voor de meeste hulpprogramma's dient gewoon nagekeken worden of een voldoende recente versie aanwezig is. In Tabel 4.1 worden de benodigde programma's voor x86-32, en bij uitbreiding x86-64, weergegeven. Er wordt ook telkens gemeld welke versies gebruikt kunnen worden. Voor de JDK wordt alleen Blackdown vermeld. Andere JDK's zijn mogelijk, maar deze worden niet permanent getest.

Met twee hulpprogramma's moet opgepast worden. Het eerste is Jikes (niet Jikes RVM). Jikes is een Java broncode compiler. Jikes compileert Java broncode naar class bestanden. De CVS heeft een tijdje problemen gehad met de nieuwere versies van Jikes. Op die momenten werd 1.18 of 1.19 aangeraden, latere versies zorgden er vaak voor dat Jikes RVM niet wou compileren. Deze problemen zijn voorlopig verleden tijd en Jikes RVM werkt tegenwoordig met de nieuwste versie van Jikes (1.22).

| **Programma** | **Versie** |
|---|---|
| GNU make | 3.79+ |
| bash | 2.05a+ |
| GNU tar | 1.13+ |
| gcc | 2.95+ |
| unzip | 5.5+ |
| Jikes Compiler | 1.22 |
| Yacc of Bison | elke |
| wget | elke |
| JDK | Blackdown 1.31, 1.41 of 1.42 |
| glibc | 2.2+ |

Tabel 4.1: Benodigde programma's voor het bouwen van Jikes RVM

Daarnaast moet ook opgelet worden met classpath. Ook hier werkt Jikes RVM vaak niet met de laatste versie(s). Daarom is het het gemakkelijkst Jikes RVM tijdens het bouwprocess zelf te laten bepalen welke classpath-versie hij nodig heeft. Het bouwprocess zal dan zelf de correcte versie downloaden en compileren, indien deze nog niet eerder geïnstalleerd was.

Om Jikes RVM te kunnen compileren en achteraf te runnen is het belangrijk enkele omgevingsvariabelen in te stellen. Deze omgevingsvariabelen, samen met een beschrijving, zijn te vinden in Tabel 4.2.

Om deze variabelen niet telkens handmatig te moeten instellen kan het script in Listing 4.1 bijzonder handig zijn. `BASE_DIR` moet wijzen naar de map waar de broncode van Jikes RVM gedownload werd (of zal worden via CVS). De twee CONFIG regels moeten naar de configuratiebestanden voor het HOST- en TARGET- platform wijzen. Het doel hiervan wordt later besproken. Het script wordt als volgt opgeroepen:

```
source settings <type>
```

Hier wordt type vervangen door een string die de opties voor het de te bouwen Jikes RVM omvat. Deze string werd besproken in paragraaf 4.1.2. De instellingen blijven dan geldig in de shell waarin het script werd uitgevoerd tot deze gesloten wordt.

Een goede eigenschap van het script is dat voor ieder type Jikes RVM dat geconfigureerd wordt een aparte map wordt aangemaakt. Zo kan men verschillende versies van Jikes RVM lokaal bewaren. Men hoeft dan voor het gebruik alleen maar het script op te roepen met het gewenste type. Daarna kan dit eerder gecompileerde type gewoon opnieuw gebruikt worden.

Vervolgens kan ingelogd worden op de CVS:

| Omgevingsvariabele | Beschrijving |
|---|---|
| RVM_ROOT | Map van de broncode |
| RVM_BUILD | Map waar Jikes RVM wordt gecompileerd en de bootImage terecht komt |
| PATH | rvm toevoegen aan PATH, zodat het van op gelijk welke locatie kan uitgevoerd worden |
| RVM_HOST_CONFIG | Configuratiebestand voor het systeem waar de image wordt opgebouwd |
| RVM_TARGET_CONFIG | Configuratiebestand voor het systeem waar Jikes RVM moet op uitgevoerd worden |
| CVSROOT | Variabele voor het gemakkelijk gebruiken van de CVS |

Tabel 4.2: Omgevingsvariabelen van Jikes RVM

```
cvs login
```

Een paswoord wordt gevraagd, gewoon enter volstaat hier. Nu kan de laatste versie gedownload worden:

```
cd $RVM_ROOT
cvs co jikesrvm
```

Ofwel - indien Jikes RVM eerder al van de CVS werd gehaald - kan een oudere versie ge-update worden;

```
cd $RVM_ROOT
cvs -q update -dP
```

Dan moet het compileerproces voorbereid worden:

```
rvm/bin/jconfigure <type>
```

Afhankelijk van de instellingen in de configuratiebestanden [1] en van het type JikesRVM worden nu scripts gegenereerd (in `RVM_BUILD`) om het compileerproces te begeleiden. Het compileren wordt als volgt gestart:

[1] RVM_HOST_CONFIG en RVM_TARGET_CONFIG

Listing 4.1: Script voor het instellen van de omgevingsvariabelen

```
BASE_DIR=/path/to/src/dir
export RVM_ROOT=${BASE_DIR}
export RVM_BUILD=${BASE_DIR}/build-cvs/${1}
export PATH=${BASE_DIR}/rvm/bin/:$PATH
export RVM_HOST_CONFIG=${BASE_DIR}/rvm/config/x86-64-pc-linux-gnu
export RVM_TARGET_CONFIG=${BASE_DIR}/rvm/config/x86-64-pc-linux-gnu
export CVSROOT=:pserver:anonymous@cvs.sourceforge.net:/cvsroot/
    jikesrvm

echo "Settings:"
echo "RVM_ROOT                $RVM_ROOT"
echo "RVM_BUILD               $RVM_BUILD"
echo "RVM_HOST_CONFIG         $RVM_HOST_CONFIG"
echo "RVM_TARGET_CONFIG       $RVM_TARGET_CONFIG"
echo "CVSROOT                 $CVSROOT"


  cd $RVM_BUILD
  ./jbuild
```

### 4.2.3 Uitvoeren

Als het bouwprocess is afgelopen, kan Jikes RVM opgeroepen worden net zoals een andere virtuele machine:

```
rvm HelloWorld
Hello World!
```

Indien men zich niet in dezelfde directory bevindt als het class-bestand dient volgend commando gebruikt te worden:

```
rvm -cp /path/to/classfile ClassFileName
```

Tal van andere argumenten kunnen meegegeven worden aan rvm om het gedrag te wijzigen. Hiervoor kan [14] geraadpleegd worden.

## 4.3 Belangrijke onderdelen

In dit onderdeel worden specifieke stukken van Jikes RVM besproken die een belangrijke invloed hebben op het platformafhankelijk gedrag ervan. Indien verwezen wordt naar bepaalde bestanden wordt hun locatie in een voetnoot vermeld. De locatie wordt relatief opgegeven ten opzicht van RVM_ROOT.

### 4.3.1 De configuratiebestanden

Zoals in paragraaf 4.2.2 werd aangegeven, moeten voor de installatie van Jikes RVM één of meerdere installatiebestanden worden opgegeven. Deze configuratiebestanden moeten alle instellingen voor het HOST- en TARGET-platform bevatten. Indien het HOST- en TARGET-platform een en hetzelfde platform zijn, kunnen de twee omgevingsvariabelen naar hetzelfde configuratiebestand verwijzen.

Een aantal voorbeeldbestanden zijn te vinden in de map `rvm/config`. De naam van deze bestanden bevat telkens de naam van het type processor en het type besturingssysteem waarvoor het bedoeld is. Hiermee wordt al aangegeven wat een eerste doel is van zo een configuratiebestand: duidelijk maken voor welk systeem Jikes RVM moet gebouwd worden (of waarop de bootimage gebouwd wordt). Voor x86-32, linux bijvoorbeeld staat in het configuratiebestand:

```
# The target architecture is Intel x86 (IA32)
export RVM_FOR_IA32=1
export RVM_FOR_32_ADDR=1

# The target OS kernel is Linux
export RVM_FOR_LINUX=1
```

Naast processor en besturingssysteem dient ook opgegeven te worden welke de adresbreedte is op het platform. Voor x86-32 is dit inderdaad 32-bit.

In een tweede deel van het configuratiebestand staan de instellingen van alle randprogramma's die tijdens het bouwproces nodig zijn. Voor het grootste deel van de programma's is dit gewoon de locatie van de executable. Voor de virtuele machine die moet gebruikt worden om Jikes RVM te bouwen zijn iets meer instellingen vereist. Daarnaast moeten ook de compilervlaggen voor gcc en g++ worden gedefiniëerd.

Alvorens de installatie op een systeem te starten is het dus van belang het configuratiebestand goed te bekijken en na te kijken of alle instellingen goed staan. Zoniet zal het in een later stadium zeker mis gaan.

### 4.3.2 jconfigure

Het jconfigure[2] script zal, afhankelijk van de instellingen, in RVM_TARGET_CONFIG en RVM_HOST_CONFIG enerzijds, en het type dat als argument wordt meegegeven aan jconfigure anderzijds, de verschillende scripts schrijven om het eigenlijke compileerproces uit te voeren. Deze scripts worden in de map RVM_BUILD opgeslagen. Bij het oproepen van het hierdoor gevormde jbuild script, in RVM_BUILD, zullen deze scripts in volgorde worden uitgevoerd.

Het jconfigure script heeft de belangrijke taak ervoor te zorgen dat de onderdelen die platformafhankelijk zijn correct geladen worden. Dit gebeurt, door afhankelijk van de variabelen die ingesteld zijn in de configuratiebestanden, zoals aangegegeven in paragraaf 4.3.1, de juiste mappen te gebruiken in de jbuild scripts.

Een andere belangrijke taak is de ondersteuning voor cross-compiling. Hierbij kan de bootImage gecompileerd worden op het HOST-platform, en de bootImageRunner op het TARGET-platform. Dit kan bijzonder handig zijn als het TARGET-platform een niet zo zware machine is. Het compileren van de bootImage is namelijk het zwaarste gedeelte. Men kan dan een zware machine als HOST-platform gebruiken om deze bootImage te maken. Het compileren van de bootImageRunner kan dan gebeuren op het TARGET-platform.

Bij de overgang tussen het eerste en het tweede deel wordt het commando 'uname -m' uitgevoerd. Dit geeft als output het platform waarop dit commando wordt uitgevoerd. Als de output verschillend is van wat verwacht wordt voor het TARGET-platform, wordt de uitvoering stopgezet en gevraagd naar het TARGET-platform te verhuizen. Als na het verhuizen het jbuild script opnieuw wordt uitgevoerd, wordt gewoon op de plaats waar gestopt was verder gegaan.

De laatste belangrijke taak van jconfigure, die belangrijk is voor dit werk, is het doorgeven van variabelen uit de configuratiebestanden. Zowel bij het compileren van de bootImage als bij het compileren van de bootImageRunner moeten omgevingsvariabelen worden meegegeven. Het is niet voldoende deze gewoon aanwezig te hebben binnen het jconfigure script. De belangrijke omgevingsvariabelen moeten bij de uitvoering van bepaalde commando's expliciet worden meegegeven.

### 4.3.3 Baseline compiler

Het doel van de BaselineCompiler is, efficiënt en snel, code genereren die duidelijk correct is. De machinecode wordt gegenereerd onmiddellijk na het inlezen van de Java byte-code. De baseline compiler is in de eerste plaats aanwezig om een relatief eenvoudige migratie naar

[2] rvm/bin/jconfigure

nieuwe platformen mogelijk te maken. Verder wordt die ook als hulpmiddel gebruikt voor andere compilers.

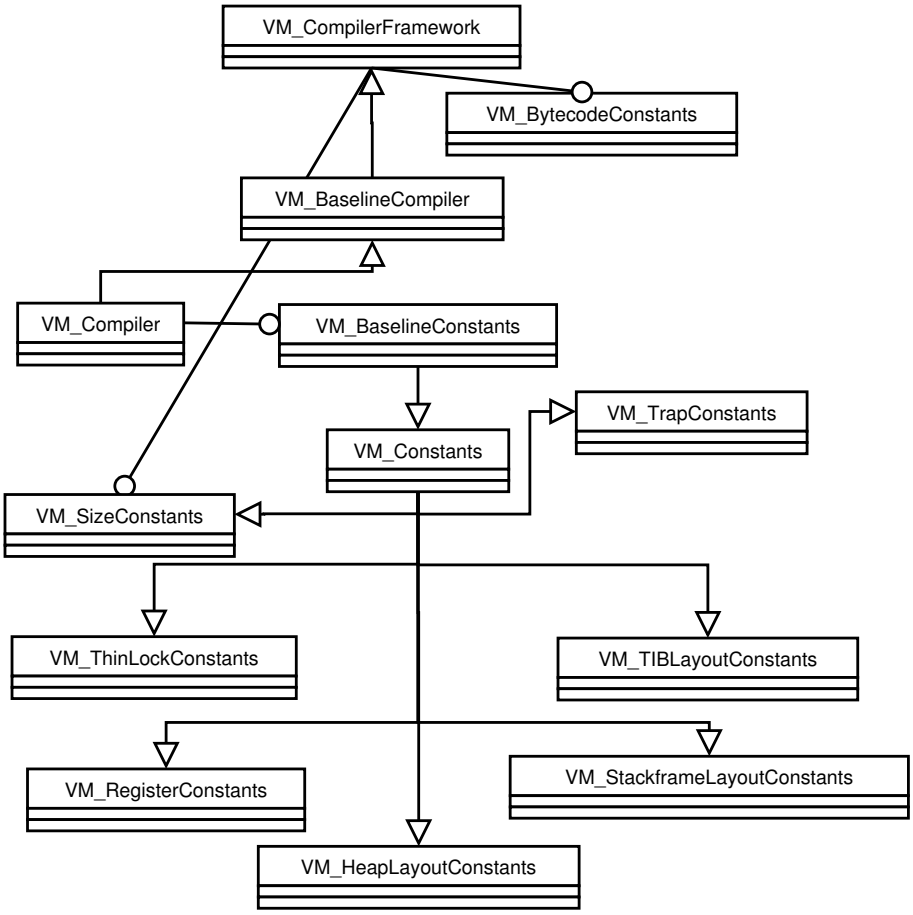
Een vereiste, die Jikes RVM aan de baseline compiler stelt, is dat deze baseline compiler de Java specificatie exact volgt. Dit maakt het opnieuw iets eenvoudiger om deze te bestuderen. De Java specificaties zijn namelijk goed gedocumenteerd. Met die documentatie bij de hand zijn grote delen van de baseline compiler een stuk gemakkelijker te begrijpen.

De BaselineCompiler is opgebouwd uit twee delen:

**Platformonafhankelijk stuk** rvm/src/vm/compilers/baseline/

**Platformafhankelijk stuk** rvm/src/vm/arch/{arch}/compilers/baseline

Figuur 4.2 geeft een overzicht van de klassestructuur van de BaselineCompiler.




Figuur 4.2: Overzicht van de BaselineCompiler

De platformafhankelijke onderdelen van dit schema zijn:

- VM_Compiler.java[3]
- VM_BaselineConstants.java[4]
- VM_TrapConstants.java[5]
- VM_RegisterConstants.java[6]
- VM_StackframeLayoutConstants.java[7]

Deze bestanden zullen het best bekeken moeten worden. De bestanden eindigend op Constants.java bevatten voornamelijk bestanden, die dan in de andere bestanden gebruikt kunnen worden. Twee bestanden doen het echte werk. Het eerste bestand, VM_BaselineCompiler.java, is platformonafhankelijk en zal, afhankelijk van de Java byte-code die ingelezen wordt, de nodig platformafhankelijke functies van het tweede bestand, VM_Compiler.java, oproepen. Het zal dan ook het laatste bestand zijn waar heel veel werk zal aan zijn. Waar nodig zullen meer details later besproken worden.

### 4.3.4 Magic

Een belangrijk nadeel aan het programmeren van een VM in Java, is dat de programmeertaal Java een aantal belangrijke beperkingen heeft voor het implementeren van een VM. Er kan niet rechtstreeks in het geheugen geschreven worden en de ondersteuning van variabelen om positieve gehele getallen te bevatteb ontbreekt. De enige dergelijke variabele is de char, maar deze is maar beperkt tot 16-bit. Voor het gebruiken van adressen en dergelijke meer is de ondersteuning voor dergelijke variabelen nodig.

Om aan deze beperking te ontsnappen maakt Jikes RVM gebruik van Magic. Voor een magic functie wordt in de Java code geen inhoud voorzien. Als een dergelijke functie dan opgeroepen wordt zal Jikes RVM ze herkennen als Magic, en de inhoud rechtstreeks vervangen door de benodigde machinecode.

### 4.3.5 Java preprocessor

De Java preprocessor is een programma met hetzelfde doel als de C preprocessor. Door het gebruik van `//-#if`, `//-#elif`, `//-#else`, `//-#endif` kan men in de code conditionele elementen aanbrengen. Alvorens de broncode naar de Java compiler te voeren, moet dan eerst

[3] rvm/src/vm/arch/{arch}/compilers/baseline/VM_Compiler.java
[4] rvm/src/vm/arch/{arch}/compilers/baseline/VM_BaselineConstants.java
[5] rvm/src/vm/arch/{arch}/VM_TrapConstants.java
[6] rvm/src/vm/arch/{arch}/VM_RegisterConstants.java
[7] rvm/src/vm/arch/{arch}/VM_StackframeLayoutConstants.java

de preprocessor op de broncode worden losgelaten. De Java compiler kent deze tags namelijk niet, en zou ze gewoon als commentaar behandelen. De preprocessor zorgt dan dat de juiste elementen achterblijven in de broncode en de elementen die niet nodig zijn verdwijnen.

De Java preprocessor dient gebruikt te worden op die plaatsen waar een gewone if-constructie uit de Java programmeertaal niet mogelijk is. Het kan bijvoorbeeld op een bepaald platform nodig zijn een variabele als int te declareren en op een ander als een long. Hiervoor moet dan de preprocessor gebruikt worden.

De preprocessor wordt tijdens het bouwprocess van Jikes RVM losgelaten op de broncode alvorens deze te compileren met de Java compiler.

### 4.3.6 bootImage en bootImageRunner

Een ander probleem met het gebruik van Java om een virtuele machine te implementeren, is dat men vanuit Java heel moeilijk een uitvoerbaar bestand kan maken. Het zou dom zijn dat men altijd een andere virtuele machine nodig heeft om Jikes RVM uit te voeren. Hieraan moet Jikes RVM een mouw passen. Dit wordt gedaan door met de Java code en een van de compilers een bootImage te schrijven. Deze bootImage bevat voor elke methode, nodig voor het opstarten van Jikes RVM, de machinecode.

Een hoop gecompileerde functies achter elkaar kan natuurlijk niet zomaar gestart worden. Om Jikes RVM te starten werd een programma geschreven in C. Dit programma moet deze bootImage in het geheugen laden en dan naar de eerste uit te voeren instructie van deze bootImage springen. De eerste instructie is het begin van de gecompileerde versie van VM.boot[8].

Een deel van de C code van de bootImageRunner is platformafhankelijk. Het is het gedeelte dat zich bezig houdt met de fysieke registers van het platform en met het opvangen van interrupts. Dit stuk zal wel enige aanpassingen vereisen om een bootImage geschreven voor x86-64 te kunnen starten op x86-64.

### 4.3.7 StackframeLayout

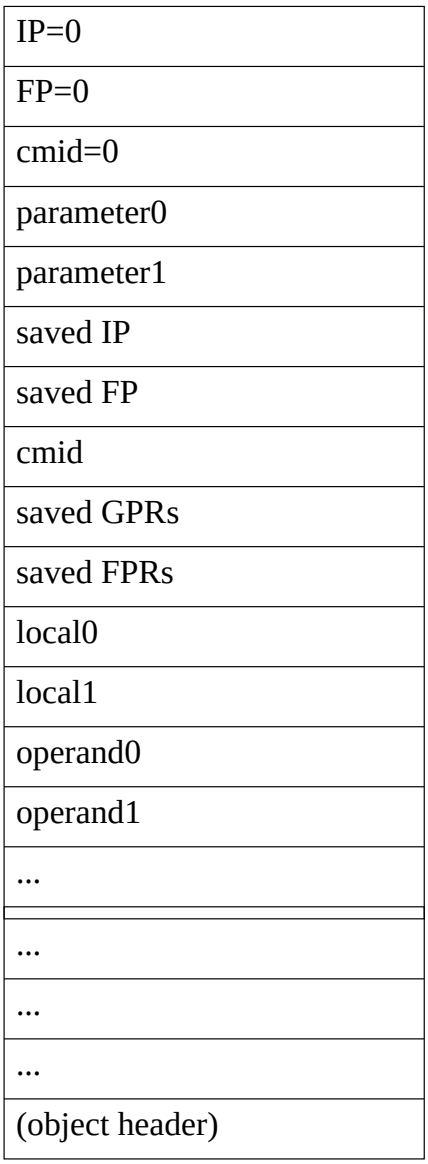
Een stack is een array van slots. Op x86 zijn deze stackslots 32-bit breed, voor x86-64 zal dit 64-bit worden. Een stackslot kan volgende elementen bevatten:

- Primitief element
- Object Pointer

[8] rvm/src/vm/VM.java

- Machine Code pointer
- Frame Pointer

De interpretatie van de inhoud van een stackslot, hangt af van de instructie die ermee omgaat. Het is aan de programmeur om ervoor te zorgen dat de elementen op een correcte manier behandeld worden.




Figuur 4.3: Opbouw Stackframe

De opbouw van een stackframe voor x86 is te vinden in Figuur 4.3. De frame is georiënteerd van high memory (boven) naar low memory (beneden).

# Deel II

# Aanpassingen

# Hoofdstuk 5

# Doel en gevolgde werkwijze

## 5.1 Doel

Het doel van het porteren van Jikes RVM naar x86-64 is alle onderdelen van Jikes RVM in 64-bit mode uitvoerbaar maken. Hierdoor krijg Jikes RVM toegang tot alle uitbreidingen van deze mode. Hiermee kan dan verder onderzoek gedaan worden.

Omwille van de grote omvang van Jikes RVM is het niet haalbaar om binnen deze thesis alle onderdelen besproken te krijgen en aangepast. Een eerste doel van dit werk was het x86-64 platform leren kennen en verschillen met het x86-32 platform aanduiden. Hierop wordt uitgebreid ingegaan in Hoofdstuk 2. Ook diende kennis opgedaan te worden rond Java en Jikes RVM. Dit wordt besproken in Hoofdstuk 3 en Hoofdstuk 4. Met deze kennis kon op zoek gegaan worden naar wat waar dient aangepast te worden.

## 5.2 Werkwijze

### 5.2.1 Voorbereidingen

Als voorbereiding op het programmeerwerk was het nodig na te kijken of alle onderdelen, om Jikes RVM te bouwen en te draaien, beschikbaar zijn voor x86-64. Zonder een 64-bit besturingssysteem en alle nodige tools voor het bouwen van Jikes RVM, was beginnen onmogelijk. Een overzicht van de beschikbare besturingssystemen wordt gegeven in paragraaf 6.1. De benodigde randprogramma's worden besproken in paragraaf 6.2.

Vervolgens werd nagekeken of Jikes RVM aan de praat te krijgen is in Compatibility Mode. Dit was belangrijk om een werkende implementatie bij de hand te hebben om mee te kunnen vergelijken. Op welke manier men Jikes RVM kan gebruiken in Compatibility Mode, wordt beschreven in paragraaf 6.3.

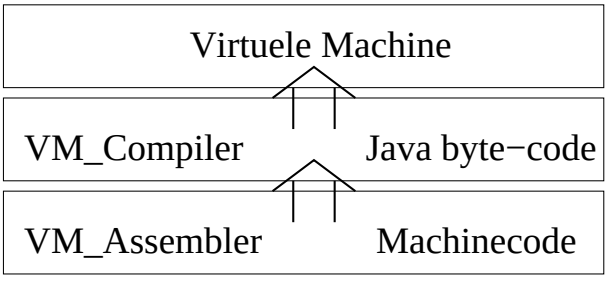
### 5.2.2 $1^{ste}$ werkwijze

Nadat alle voorbereidingen getroffen waren kon het echte werk beginnen. De eerste werkwijze die gevolgd werd, was het volgen van het levensproces van Jikes RVM. Deze werkwijze was gebruikt bij het porteren van PPC naar PPC64. Eerst werd jconfigure nagekeken om te zien wat daar aangepast moet worden om het systeem draaiende te krijgen. Vervolgens werd het jbuild proces gevolgd, en daar waar fouten optraden werden die onderhanden genomen, tot op een bepaald moment het jbuild proces zonder fouten rond geraakte en men dus rvm kon oproepen. Hiervoor diende ook het platformafhankelijk deel van de bootImageRunner worden aangepast. Dit laatste wordt besproken in paragraaf 7.4.

Hier duikt nu de grote moeilijkheid op. Doordat het belangrijkste deel van Jikes RVM de bootImage is, en deze totaal niet correct is na wegwerken van een paar compilatiefouten, gaf het uitvoeren van Jikes RVM onmiddellijk segmentatiefouten. Dit was voorzien. De bedoeling was dan ook Jikes RVM al onmiddellijk bij het opstarten te laten termineren. Ook dit wou echter niet werken. Na een tijd gezocht te hebben langs deze weg, leek het opportuun een nieuwe werkwijze te zoeken.

### 5.2.3 $2^{de}$ werkwijze

Als tweede werkwijze werd gekozen voor een opwaartse aanpak. De opwaartse methode vertrekt bij het aanpassen van de machinecode. Vervolgens wordt dan gekeken naar de omzetting van Java byte-codes in machinecode. Het voordeel van de opwaartse methode is dat men er min of meer kan op vertrouwen dat code van een onderliggende laag aangepast en correct is. Verder bestudeert men via deze weg voornamelijk de belangrijke delen van de code. Het is namelijk een onmogelijke opdracht om de 300.000 lijnen code van Jikes RVM allemaal te bestuderen in een beperkte tijd. De opzet van deze opwaartse werkwijze wordt weergegeven in Figuur 5.1.




Figuur 5.1: Opwaartse Werkwijze

Allereerst werd dus gekeken naar de functies die machinecode genereren. De aanpassingen die hier nodig waren worden besproken in paragraaf 7.5. Met de hier opgedane kennis van de on-

derste laag, kon gekeken worden hoe Java byte-codes omgezet worden in machine-instructies. Hierbij was het belangrijk eerst goed na te kijken hoe men elementen op de stack zou kunnen plaatsen op x86-64. Doordat de stack op x86-64 64-bit breed is, en een stackslot dus 8 bytes breed is, diende goed nagedacht te worden hoe de verschillende types op de stack te plaatsen. De gekozen conventie dient dan verder gevolgd te worden in VM_Compiler.java, waar de eigenlijke omzetting van Java byte-code naar machinecode gebeurd. Hoe dit werd aangepakt wordt besproken in paragraaf 7.6.

Bij het nakijken van VM_Compiler werd duidelijk dat het een goede beslissing was geweest van werkwijze te veranderen. Het gebruik van de stack in VM_Compiler was namelijk verre van correct. Bij het gebruik van PUSH of POP wordt de stack op x86-64 met 8 bytes vergroot, respectievelijk verkleind. Men kan echter ook met de stack werken door elementen te verplaatsen van en naar de stack, en achteraf de stackpointer op een correcte manier te verhogen of te verlagen. Bij het gebruik van de tweede methode liep het erg mis. De stackpointer werd dan telkens verhoogd of verlaagd met een $4x$, met $x$ het aantal stackslots. Op x86-64 moet hier $8x$ gebruikt worden.

Het is duidelijk dat het door elkaar gebruiken van de twee bovenstaande methoden de stack compleet verknoeid. Doordat Java heel sterk via de stack werkt, was het belangrijk dit eerst op te lossen. Hiervoor was het nodig al deze veelvouden van 4 op te zoeken en te vervangen door een veelvoud van WORDSIZE. WORDSIZE heeft op x86-32 de waarde 4 en op x86-64 de waarde 8.

Om dit probleem aan te pakken werd volledig VM_Compiler.java doorlopen om een correct gebruik van de stack te verzekeren. Ondertussen werd ook al gekeken welke stukken van VM_Compiler aangepast moesten worden voor x86-64. Deze aanpassingen werden ondertussen ook doorgevoerd.

Testen van de aanpassingen was echter niet mogelijk tot het volledige bestand was doorlopen, en een consistent gebruik van de stack verzekerd was. Nadat dit bestand volledig was gecontroleerd, samen met de bijhorende bestanden die machinecode genereren, kon opnieuw verder gewerkt worden op de plaats waar bij de eerste werkwijze gestopt werd.

Na het wegwerken van nog enkele fouten (via debuggen in gdb) was het uiteindelijk mogelijk de bootImage op te roepen, en dan op een normale wijze te laten afsluiten. Dit laatste gebeurt door gebruik te maken van VM_SysCall.sysExit. Deze springt naar de C code van de bootImageRunner en stopt Jikes RVM. Deze functie kan gebruikt worden helemaal in het begin van VM.boot, de functie van VM.java[1] die als eerste van de bootImage wordt uitgevoerd. Op die manier wordt de virtuele machine onmiddellijk bij het opstarten terug afgesloten.

Het testen van kleine stukjes Java-code kan dan gebeuren door deze in het begin van VM.boot

[1] rvm/src/vm/VM.java

te plaatsen, net voor een VM_SysCall.sysExit. Als eerste werd op deze manier gekeken naar een paar eenvoudige print functies. Zodra deze werkten konden elementaire types op het scherm gebracht worden. Op die manier konden stukjes Java code getest worden om verschillende Java byte-codes te testen. De Java byte-codes voor bewerkingen op integers, longs en doubles werken nu correct. Ook vergelijkingen tussen longs en integers werken. Verder is het ook mogelijk for en while lussen uit te voeren. Andere byte-codes moeten nog getest worden, om zo VM_Compiler volledig correct te krijgen.

Maar zelfs als dit bestand correct is, is er toch nog wat werk aan de winkel. Een eerste punt waar nog werk nodig is, is de link tussen Java code en platformafhankelijke code, bijvoorbeeld aanwezige gecompileerde C code. Omwille van de veranderde conventies in verband met het doorgeven van parameters loopt dit nog volledig mis.

Verder wordt in de code die hardwaretraps opvangt gebruik gemaakt van een disassembler. Deze bepaalt de lengte van de huidige instructie. Met deze lengte moet de instructiepointer verhoogd worden om met de volgende instructie verder te gaan. Omwille van de veranderingen in de instructieset moet ook deze disassembler worden aangepast.

Als deze aanpassingen zijn gemaakt zou de baselinecompiler correct moeten werken. Daarna kan verder gegaan worden met het porteren van garbage collectors, de optimizing compiler en het adaptief systeem. Al deze bovenstaande dingen waren echter niet meer mogelijk binnen de beschikbare tijd.

## 5.3 Debuggen in Jikes RVM

Het debuggen van Jikes RVM is geen eenvoudige zaak. Doordat bij het bouwproces van Jikes RVM een bootImage geschreven wordt, gaat het verband met de broncode verloren. De bootImageRunner, die in C geschreven is, kan wel gedebugged worden op het niveau van de instructies.

Om het debug-proces iets te vergemakkelijken wordt bij Jikes RVM een script meegeleverd, *gdbrvm*. De bedoeling is gdbrvm op te roepen met dezelfde argumenten waarmee men rvm zou oproepen. Het script roept dan gdb op de correcte manier op, en plaatst ook een breakpoint aan het begin van bootThread.S. Hierdoor kan in de eerste plaats nagekeken worden of correct wordt gesprongen naar het begin van de bootImage.

Indien er nog problemen zouden zijn in de bootImageRunner, ergens in een stuk voor het oproepen van de functie bootThread.S, kan men met het command *break functienaam* nog extra breakpoints plaatsen. Voor het debuggen van het platformafhankelijke deel van de bootImageRunner, libvm.c, kwam dit goed van pas. Zolang de juiste argumenten niet meegegeven zijn aan de functie bootThread is het namelijk nutteloos verder te debuggen.

Bij normaal debuggen wordt het commando step gebruikt. In het geval van bootThread.S gaat die bijzonder goed, omdat hier de instructies expliciet in assembler genoteerd zijn. Als men echter step doet bij de call instructie, heeft gdb geen informatie meer van welke instructie hij aan het uitvoeren is, en daarom voert hij de commando's binnen de bootImage uit (voor zover die juist zijn), tot er ergens iets mis gebeurt. Op dat moment komt er een interupt en deze wordt opgevangen door de hardwaretraphandler van de bootImageRunner. Op dat moment kan gewoon verder gedebugged worden.

Spijtig genoeg is het niet de bootImageRunner die het grootste probleem vormt. Vooral de instructies, uitgevoerd in bootImage, moeten nagekeken worden. Om dit te doen kan men het commando *stepi* gebruiken. Deze voert telkens gewoon de volgende machine-instructie uit. Spijtig genoeg is de output die gdb dan geeft niet zo nuttig:

```
105             call *rAX               // branch to RVM
(gdb) stepi
0x00000000436f96d0 in ?? ()
(gdb)
0x00000000436f96d4 in ?? ()
(gdb)
0x00000000436f96d8 in ?? ()
```

Er dient opgemerkt te worden, dat men *stepi* niet altijd opnieuw moet typen. Als men gewoon enter duwt wordt het laatste commando opnieuw uitgevoerd. Met de pijltjes omhoog (en eventueel weer omlaag) kan men door de geschiedenis van de commando's gaan, en eventueel vroegere commando's terug ophalen.

Om het probleem van hierboven op te lossen kan men het commando *display /i $pc* uitvoeren alvorens men *stepi* doet:

```
105             call *rAX               // branch to RVM
(gdb) display /i $pc
1: x/i $pc  0x40a754 <bootThread+20>:   callq  *%eax
(gdb) stepi
0x00000000436f96d0 in ?? ()
1: x/i $pc  0x436f96d0: rex pushq  0x68(%rsi)
(gdb)
0x00000000436f96d4 in ?? ()
1: x/i $pc  0x436f96d4: mov    %rsp,0x68(%rsi)
(gdb)
0x00000000436f96d8 in ?? ()
1: x/i $pc  0x436f96d8: movq   $0x20fd,0xfffffffffffffff8(%rsp)
```

Het resultaat van het ingeven van dit commando is dat gdb na het uitvoeren van een instructie (of een reeks instructies), de volgende instructie inleest en zelf naar assembler vertaalt. Dit geeft al iets meer informatie.

Door zonder meer deze instructies te volgen kan men echter niet bijzonder veel leren. Tenslotte is het bijzonder lastig om zelf, met de hand, over te gaan van de Java broncode waarin Jikes RVM geschreven is, via een eigen omzetting naar Java byte-code, naar de machinecode die uitgevoerd wordt.

Hier kan Jikes RVM zelf hulp bieden. Door enkele instellingen te veranderen, kan Jikes RVM tijdens het compileerproces van de bootImage, output genereren waarbij per methode de gegenereerde machinecode wordt weergegeven, zowel in machinecode als in assembler. Ook worden telkens de Java byte-codes weergegeven die aan de grondslag liggen van de gegenereerde machinecode.

Om deze informatie te genereren moeten in SharedBooleanOptions.dat[2] twee waarden op true gezet worden:

```
PRINT_METHOD -1 true verbose
Print method name at start of compilation

PRINT_MACHINECODE -1 true mc
Print final machine code
```

Deze output kan men dan tijdens het bouwproces naar een bestand sturen.

```
./jbuild > mc 2>&1
```

De output kan men, als het bouwproces afgelopen is, vinden in het bestand mc. Zonder de `2>&1` zal de nodige inhoud op het scherm belanden, en niet in het bestand mc. Een groot nadeel aan deze methode is wel dat, door het genereren van deze output, het bouwproces een stuk langer duurt.

Nu kan men het werk van de bootImage volgen door in het bestand te zoeken naar VM.boot(). Het is namelijk naar het begin van deze methode dat er gesprongen wordt in bootThread.S.

In mc ziet dit er dan als volgt uit:

```
baseline Start: Final machine code for method com.ibm.JikesRVM.VM boot ()V
000000| prologue for com.ibm.JikesRVM.VM.boot ()V
000000|      PUSH            104[rsi]                | 40FF7668
000004|      MOV             104[rsi]          rsp   | 48896668
000008|      MOV              -8[rsp]         8445   | 48C74424F8FD200000
```

[2] rvm/src/vm/compilers/utility/SharedBooleanOptions.dat

Op die manier kan dus getest worden of de code gegenereerd door VM_Compiler correct is, en ook of de eigen gevormde instructies uit VM_Assembler correct zijn.

Nog een aantal andere instructies kunnen bijzonder handig zijn tijdens het debuggen. Als eerste kan *info registers* gebruikt worden. Zoals de naam al doet vermoeden brengt deze de inhoud van de registers op het scherm. Op momenten waar met rationele getallen wordt gewerkt, kan *info all-registers* goed van pas komen. Dan wordt ook de inhoud van de xmm registers weergegeven. Het controleren van rationele gegevens is niet zo gemakkelijk. Deze worden enerzijds hexadecimaal en anderzijds alsof het gehele waarden zijn, weergegeven. Voor de conversie van hexadecimaal naar decimale notaties kan de website `http://babbage.cs.qc.edu/courses/cs341/IEEE-754.html` gebruikt worden.

Tenslotte is *info stack* ook bijzonder handig. Deze toont de top van de stack, en alles wat eronder zit. Om na te kijken of de frame layout correct gevolgd wordt, en de lokale variabelen op de stack op de correcte plaats komen kan dit commando goed van pas komen. Ook het correct gebruik van de operand stack kan hiermee gecontroleerd worden.

# Hoofdstuk 6

# Voorbereidingen en benodigdheden

In dit hoofdstuk wordt nagekeken of alle benodigdheden om Jikes RVM op een x86-64 besturingssysteem te draaien aanwezig zijn. Allereerst is een besturingssysteem nodig. Vervolgens zijn ook verschillende hulpprogramma's vereist. Tenslotte moet ook een configuratiebestand voor het nieuwe platform gemaakt worden.

## 6.1 Besturingssysteem

Omdat voor x86-32 Jikes RVM alleen maar linux ondersteund, zal voor de x86-64 versie ook linux gebruikt worden. Om toegang te krijgen tot de 64-bit mode is een besturingssysteem nodig, voor Jikes RVM dus een x86-64 linux distributie. Verschillende distributies bieden een versie aan voor x86-64.

Op de huidige datum[1] hebben volgende populaire distributies een versie voor x86-64, waarvan vermeld wordt dat hun distributie gebruikt kan worden voor AMD64 en EM64T.

- Mandriva[2]
- Fedora
- Red Hat
- SuSE
- Ubuntu

Gentoo biedt een versie aan voor AMD64. Deze zou ook moeten werken voor EM64T, maar daarvoor bieden ze geen garantie. Op het forum staan wel al verschillende berichten van

[1] 31 mei 2005
[2] Vroeger Mandrake

werkende systemen steunend op de AMD64 installatiebestanden. In hun nieuwsbrief van 31 januari 2005[3] vragen ze ontwikkelaars voor EM64T om pakketten te testen voor EM64T. Op het forum van de distributie hebben verschillende gebruikers wel al gemeld met succes de AMD64 versie geïnstalleerd te hebben op hun processor met EM64T ondersteuning.

De enige die niet mee doet is Debian. Op hun site wordt x86-64 niet vermeld. De laatste berichten, in nieuwsbrieven van Debian, over x86-64 dateren van 2003 dus vermoedelijk is de port daar stilgelegd.

Voor dit werk werden twee testsystemen gebruikt:

- Dual Opteron 244 met 64-bit Red Hat Linux
- Athlon64 3000+ met 64-bit Gentoo Linux

Alhoewel Gentoo geen expliciete support voor EM64T biedt, kan deze distributie toch als testplatform gebruikt worden, zolang er maar rekening mee gehouden wordt dat geen AMD64 specifieke dingen gebruikt worden.

Wegens een gebrek aan een systeem met ondersteuning voor EM64T kon hierop niet getest worden.

## 6.2 Randprogramma's

De benodigde randprogramma's worden weergegeven in Tabel 4.1. Op beide platformen zijn al deze programma's aanwezig. Ze behoren ook allemaal tot de stabiele pakketten. Hierdoor kan er vanuit gegaan worden dat ze voldoende getest zijn geweest, en hun gedrag dus idem is als op x86-32. Het zal dus ook geen probleem zijn deze randprogramma's draaiende te krijgen op andere x86-64 distributies.

Alleen classpath behoord niet altijd tot de stabiele pakketten van de distributie, maar door deze door Jikes RVM te laten installeren kan nagekeken worden of alles hiermee goed gaat. Met classpath zijn nooit problemen gebleken.

## 6.3 Jikes RVM in Compatibility Mode

Aangezien code gegenereerd voor een x86-32 systeem kan uitgevoerd worden op een 64-bit besturingssysteem in Compatibility Mode, moet ook Jikes RVM in deze mode uitvoerbaar zijn. Opdat dit mogelijk zou zijn op een 64-bit linux distributie moet er wel op gelet worden dat enerzijds ondersteuning van 32-bit binaries in de kernel gebakken is en anderzijds de benodigde

[3] `http://www.gentoo.org/news/en/gwn/20050131-newsletter.xml`

32-bit bibliotheken aanwezig zijn. Bij distributies waar de kernel niet zelf gecompileerd wordt, is het eerste normaal geen probleem. Dan is de kernel algemeen genoeg gebouwd. Indien men zelf de kernel compileert, mag men deze optie niet vergeten aan te zetten. De 32-bit bibliotheken zijn pakketten die gewoon aan het systeem moeten toegevoegd worden. Dit is geen moeilijke stap, maar mag wel niet vergeten worden.

Een probleem dat hier opduikt is dat Jikes RVM niet als binair pakket kan worden gedownload. Het moet op het systeem zelf gecompileerd worden. Hiervoor heeft men, naast de normale benodigdheden voor JikesRVM, een compiler op het 64-bit besturingssysteem nodig die ook 32-bit x86 code kan genereren. Standaard wordt voor Jikes RVM gcc gebruikt. Door deze te bouwen met ondersteuning voor 32-bit code (multilib support), kan ook gcc blijven gebruikt worden. Het enige dat dan veranderd moet worden zijn de compiler vlaggen voor gcc en g++ in het TARGET configuratiebestand. Hier dient *-m32* aan toegevoegd te worden.

Een volgend punt waar op gelet dient te worden is dat ook classpath gecompileerd is met de nieuwe vlaggen. Indien dit niet het geval is, zal Jikes RVM zonder problemen gecompileerd worden, maar de uitvoering zal falen wanneer hij de bibliotheken van classpath probeert te laden. Deze zijn dan namelijk voor 64-bit mode gecompileerd en kunnen niet gebruikt worden in Compatibility Mode. Indien men probeert zowel 64-bit als 32-bit versies in eenzelfde broncodemap te maken kan dit tot problemen leiden, omdat er niet nagekeken wordt met welke vlaggen classpath gecompileerd werd. In een andere map werken, waar de submappen rvm, MMTk in gelinkt worden kan hier voor een oplossing zorgen.

Een laatste probleem duikt op bij het compileren van JikesRVM. Voor de ondersteuning van crosscompiling wordt nagekeken of het doelplatform overeenkomt met het resultaat van *uname -m*. Voor Jikes RVM op x86-32 wordt dit i686. Voor x86-64 wordt dit x86_64. Om dit te verhelpen wordt in het configuratie-bestand een extra variabele aangebracht, RVM_FOR_X86_64=1. Voor het configureren van JikesRVM wordt het configuratiescript jconfigure gebruikt. Door dit bestand te patchen, zodat voor RVM_FOR_X86_64=1 het gecontroleerd wordt of *uname -m* x86_64 geeft, kan dit probleem verholpen worden.

Op eenvoudige wijze kan op deze manier JikesRVM ook op een 64-bit x86 besturingssysteem uitgevoerd worden. In een dergelijke versie kan men echter geen gebruik maken van de grotere adresruimte en de extra registers.

Om niet altijd een bestaand configuratiebestand te moeten aanpassen, werd een nieuw configuratiebestand gemaakt. Dit kreeg de logische naam *x86-64-pc-linux-gnu-32*.

Om dit niet altijd manueel te moeten doen werd een patch ontwikkeld. Deze patch is te downloaden op `www.frocksii.be` onder de sectie Downloads. De installatie van deze patch dient als volgt te gebeuren (na downloaden van de patch naar RVM_ROOT):

```
$ cd $RVM_ROOT
```

```
$ tar -xzvf JikesRVMx86-64_32bit.tar.gz
x86-64-pc-linux-gnu-32
x86-64_32bit
README
$ cp x86-64-pc-linux-gnu-32 rvm/config/
$ patch -p0 < x86-64_32bit
patching file rvm/bin/jconfigure
```

## 6.4 Configuratiebestand

Naar analogie met het configuratiebestand voor Jikes RVM in Compatibility Mode krijgt het configuratiebestand voor 64-bit mode de naam *x86-64-pc-linux-gnu-64*.

In een eerste gedeelte van het configuratiebestand moet ingesteld worden voor welke type systeem Jikes RVM gebouwd wordt. Voor het nieuwe configuratiebestand ziet dit er als volgt uit:

```
# The target architecture is Intel x86
export RVM_FOR_IA32=1
# The target platform has x86-64 extensions enabled
export RVM_FOR_64_ADDR=1
export RVM_FOR_X86_64=1
# The target OS kernel is Linux
export RVM_FOR_LINUX=1
```

Twee dingen zijn verschillend ten opzichte van x86-32. Ten eerste dient aangegeven te worden dat Jikes RVM gebouwd zal worden voor 64-bit adressen. Dit wordt gedaan door *RVM_FOR_64_ADDR=1* in te stellen. Deze variabele was al aanwezig sinds de PPC64 versie. Op x86-32 staat er *RVM_FOR_32_ADDR=1*. Verder wordt, net zoals voor Jikes RVM in Compatibility Mode, de variabele *RVM_FOR_X86_64=1* geïntroduceerd. Deze zal een eerste maal gebruikt worden om de *uname -m* test correct te laten verlopen. Een tweede maal wordt deze gebruikt om in het platformafhankelijk gedeelte van de bootImageRunner een onderscheid te maken tussen code specifiek voor x86-32 en code specifiek voor x86-64.

Ook hier moeten per platform de specifieke instellingen van de verschillende randprogramma's worden nagekeken. Voor gcc en g++ moeten de vlaggen niet veranderd worden ten opzichte van x86-32. Als geen speciale vlaggen worden gebruikt zal gcc/g++ deze gewoon x86-64 code genereren.

Een laatste aanpassing hier is aan de variabele *MAXIMUM_MAPPABLE_ADDRESS=0xc0000000l*. Deze diende als long geïnitialiseerd in plaats van integer geïnitialiseerd te worden.

# Hoofdstuk 7

# Aanpassingen aan Jikes RVM

## 7.1 Algemene principes

In [13] wordt uiteengezet, hoe de Java specificatie 32-bit platformen bevoordeeld. Wat hierin vooral belangrijk is, is dat er toch twee stackslots dienen gebruikt te worden om een long of double op de stack te plaatsen. In theorie kunnen deze types in een stackslot. Het gebruik van meerdere stackslots voor long en double wordt verder uitgewerkt in paragraaf 7.2.

Daarnaast dient het gebrek aan een unsigned int vermeld te worden. Hierdoor moeten alle adressen in de implementatie als een eigen type gezien worden. Voor 64-bit ontbreekt ook de unsigned long. Zowel voor adressen als offsets zijn er binnen Jikes RVM eigen types aanwezig. Ondersteuning voor de 64-bit adressen is, omwille van de 64-bit PPC versie, in de platformonafhankelijke delen, al aanwezig.

Naast een verschillende omgang met de stack moet nu ook verschillend omgegaan worden met adressen. Deze moeten nu behandeld worden als 64-bit breed, en niet als 32-bit breed zoals op x86. Verder kunnen bewerkingen op longs rechtstreeks uitgevoerd worden en niet via een reeks instructies, zoals berekeningen op longs op een 32-bit platform uitgevoerd worden.

Aangezien x86-64 een uitbreiding is op x86-32, zal ook de code als een uitbreiding op de x86 code worden gezien. Er zal gebruik gemaakt worden van de precompiler en van de variabelen uit VM_Configuration, om die plaatsen waar aanpassingen nodig zijn, het verschil te maken tussen x86-32 en x86-64. Hierdoor moeten geen nieuwe bestanden worden toegevoegd en kan bij het schrijven van aanpassingen gemakkelijk zowel de x86-32 als de x86-64 aangepast te worden.

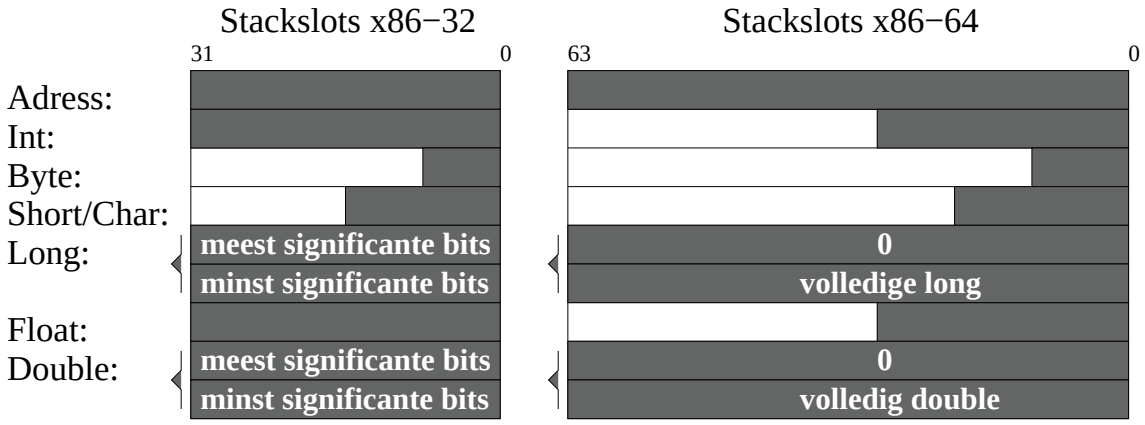
In de volgende onderdelen worden de verschillende veranderingen nodig binnen de broncode van Jikes RVM besproken in iets meer detail.

Om te tekst niet te overladen, worden de implementatie details van de veranderingen verschoven naar Appendix A. Vooral de ontwerpsbeslissingen worden hieronder besproken.

## 7.2 Nieuw stackmodel

Een gevolg van het werken op een 64-bit platform is een 64-bit brede stack. Hierdoor kunnen alle primitieve datatypes van Java in een stackslot gestoken worden. De specificaties voor een Java virtuele machine houden echter in dat doubles en longs 2 stackslots moeten innemen. Dit kan opgelost worden door hier een leeg stackslot in te schuiven en daar dan later rekening mee te houden. Om de code duidelijk te houden wordt ervoor gekozen hier telkens een 0 op de stack te plaatsen. Indien de specificaties niet op de letter gevolgd worden, kan men longs en doubles gewoon in een stackslot plaatsen. Binnen de baselinecompiler geldt voor Jikes RVM echter de conventie, dat de specificatie op de letter gevolgd wordt.

De plaats van de verschillende types op de stack is weergegeven in Figuur 7.1.




Figuur 7.1: Plaats van types op de stack

Het op de stack plaatsen van 8-, 16, 32- en 64-bit gegevens kan nu altijd op dezelfde manier gebeuren. Men gebruikt PUSH om de gegevens op de stack te plaatsen. Op PPC is dit enigszins anders, omdat hier een big-endian architectuur gebruikt wordt. Hierdoor komt de waarde van bijvoorbeeld een byte links op de stack, en moet deze byte eerst naar links geschoven worden met een shift alvorens deze op de stack kan geplaatst worden. Bij little endian is dit gemakkelijker, de minst significante byte zit rechts. En bij het pushen komt de waarde ook perfect rechts te zitten.

Bij de operand stack moet wel enigszins opgelet worden. Bytes en shorts moeten tekenuitgebreid worden naar integer, en chars nul-uitgebreid. Dit omdat bewerkingen op deze types via bewerkingen op integer worden uitgevoerd.

Naast het consequent gebruik van dit nieuwe stackmodel op de plaatsen waar machinecode wordt gegenereerd moet slechts een bestand aangepast worden:
VM_StackframeLayoutConstants.java[1]. Op zich komt het hier vooral op neer de waarde van de constanten met twee te vermenigvuldigen, hetgeen het gevolg is van de verdubbeling van de adresbreedte.

De enige uitzondering hierop is de constante FPU_STATE_SIZE. Die geeft aan hoeveel ruimte de x87-stack inneemt in het geheugen. Deze constante wordt dan gebruikt om de stack in orde te houden na het uitvoeren van commando's, zoals FNSAVE, die deze x87-stack op de gewone stack plaatst. De plaats nodig op de stack voor de uitvoering van deze commando's is zowel op 32- als op 64-bit 108, de grootte van deze stack. Op 32-bit blijft de stack na gebruik van deze constante mooi gealigneerd op 4-bytes. 108 is echter geen veelvoud van 8, en als dus geen extra actie ondernomen wordt, is bij gebruik van deze constante de stack niet mooi gealigneerd meer. Dit kan opgelost worden door deze constante een veelvoud van 8 te maken - 112 is dan de beste keuze. Instructies als FNSAVE vergroten namelijk de stack niet zelf, zodat bij correct gebruik die extra 4-bytes alleen maar vulling zullen zijn om de stack gealigneerd te houden.

## 7.3 jconfigure

De aanpassingen nodig aan jconfigure zijn dezelfde als de aanpassingen die nodig waren om Jikes RVM in compabiliteitsmode te laten draaien. Deze worden dus besproken in paragraaf 6.3.

In tegenstelling tot de versie van Jikes RVM in Compatibiliteitsmode, is het dieper in de Java code van Jikes RVM ook belangrijk te weten dat het om een versie gaat voor x86-64. Hiervoor diende ook VM_Configuration.java te worden aangepast. Deze bevat een reeks statische variabelen (booleans) die waar of vals zijn, afhankelijk van de instellingen van de configuratiebestanden. Hier werd zo een nieuwe variabele ingevoerd: BuildForX86_64. Achteraf gezien werd deze variabele maar op een paar plaatsen gebruikt. Als in de toekomst blijkt dat het bij die paar plaatsen blijft, kan misschien op de plaatsen dat ze gebruikt wordt, voor een andere oplossing gezorgd worden. Dan kan Jikes RVM verder met zo een variabele minder.

## 7.4 BootImageRunner

De taak van de bootImageRunner wordt besproken in paragraaf 4.1.3. Het belangrijkste doel is dus het correct starten van de bootImage. Sommige andere taken, die eerder moelijk zijn

[1] rvm/src/vm/arch/intel/VM_StackframeLayoutConstants.java

in Java, worden ook door deze bootImageRunner op zich genomen. Op platformafhankelijk gebied is dit het afhandelen van interrupts.

De BootImageRunner[2] bestaat uit een platformonafhankelijk en een platformafhankelijk deel. Het is in het platformafhankelijk deel dat enkele wijzigingen nodig waren[3]. Het belangrijkste bestand is daar libvm.C[4]. Op een bepaald moment wordt in libvm.C een functie, geschreven in assembler, opgeroepen. Deze is te vinden in bootThread.S[5].

### 7.4.1 libvm.C

In libvm.C wordt de bootImage geladen. Bij het laden van de bootImage wordt ook nog allerlei randvariabelen correct gezet. Daarnaast worden ook enkele testen uitgevoerd om na te kijken of de bootImage correct geladen is.

Deze testen gebeuren aan de hand van de inhoud van de stack. Hierbij worden ook afstanden op deze stack bekeken. Op x86-32 is een stackslot 4 bytes, maar op x86-64 is dit 8 bytes. Daarom werd er besloten via de C precompiler een aantal constanten te definiëren afhankelijk de vlag `RVM_FOR_X86_64`. Hierdoor kon de bestaande libvm.C behouden blijven mits enkele aanpassingen.

Naast het gebruik van 8 voor de breedte van een stackslot, was het ook nodig constanten in de voeren die overeen komen met de namen van de registers. In de interface die gebruikt wordt voor de registers wordt op x86-64 namelijk gebruik gemaakt van de benaming met een R (RAX, RBX, ...) en op x86-32 is dit nog altijd de benaming met E (EAX, EBX, ...). Verder vallen enkele segmentregisters weg, dus mogen ook de aanroepen naar deze registers niet meer voor komen in de code.

Verder werd er voor gezorgd dat de inhoud van registers in variabelen komt met een correcte lengte. Op 32-bit is dit een int, op 64-bit een long. Door opnieuw via de precompiler te werken, en overal ADDRESS te vervangen door int/long werd dit opgelost.

Tenslotte werd er ook voor gezorgd dat voor een 32-bit adres altijd 4 bytes worden afgedrukt, en voor een 64-bit 8 bytes. Dit is vooral belangrijk als er fouten optreden. In die gevallen worden, afhankelijk van het type fout, een of meer register op het scherm afgeprint. Indien slechts 4 bytes zouden worden afgedrukt voor een 64-bit register, kan nuttige informatie verloren gaan.

[2] rvm/src/tools/bootImageRunner
[3] rvm/src/vm/bootImageRunner/IA32
[4] rvm/src/vm/bootImageRunner/IA32/libvm.C
[5] rvm/src/vm/bootImageRunner/IA32/bootThread.S

### 7.4.2 bootThread.S

bootThread.S heeft als taak een aantal registers correct te initialiseren en dan naar de eerste instructie van de bootImage te springen. Hiervoor moeten het JTOC, PR en SP geladen worden met de waarde die meegegeven wordt als argument bij het aanroepen van de functie bootThread.

De eerste aanpassing die hier nodig was, was de assembler correct te maken voor x86-64. Een aantal instructie werden ongeldig, zoals het aanpassen van de stackpointer door middel van een 32-bit functie. Een aantal instructie diende naar 64-bit te worden omgezet. Verder is ook de manier, waarop parameters worden doorgegeven naar een methode, verschillend. Op x86-64 gebeurt dit, zover mogelijk, via parameters. Ook deze aanpassingen werden doorgevoerd aan bootThread.S, zodat deze volledig correct werkt.

Initiëel werden de parameters voor bootThread.S vervangen door longs, omdat het om waarden voor 64-bit registers gaat. Achteraf werd deze verandering ongedaan gemaakt. De reden hiervoor is dat de parameters allemaal adressen zijn binnen het bereik van Jikes RVM. Deze vallen echter allemaal mooi binnen 32-bit. Het veranderen van de oproep van bootThread.S had ook gevolgen voor het platformonafhankelijk deel, en het leek dus eenvoudiger voorlopig integer te blijven gebruiken voor deze parameters. Als in de toekomst de adresruimte voor 64-bit versies van Jikes RVM breder zou gemaakt worden, moet dit natuurlijk wel aangepast worden.

### 7.4.3 Andere aanpassingen

Om de bootImageRunner een bepaalde graad van platformonafhankelijkheid te geven, wordt het header bestand InterfaceDeclartions.h dynamisch gegenereerd, afhankelijk van de instellingen. Dit wordt gedaan met GenerateInterfaceDeclartions.java[6]. Oorspronkelijk dacht dit bestand dat het een bestand diende te genereren voor PPC in plaats van voor x86-64. Dit werd aangepast. Verder werd hier ook een adres gezien als een int. Dit werd aangepast zodat voor x86-64 een adres een long wordt gebruikt.

## 7.5 Machinecode

Om de omzetting van de Java byte-code naar machinecode een beetje overzichtelijk te houden, voorziet Jikes RVM een klasse met Java functies die ongeveer alle machine-instructies representeren die nodig zijn. Deze klasse is VM_Assembler.java. VM_Assembler.java bevat voor elke opcode verschillende functies die de verschillende adresseermodes en operandbreedtes voor verschillende opcodes aanbieden.

[6] rvm/src/tools/bootImageRunner/GenerateInterfaceDeclarations.java

Omdat de structuur van de functies voor instructies met eenzelfde operandbreedte en eenzelfde adresseermode zeer gelijklopend is, wordt een groot deel van VM_Assembler.java gegenereerd door een script: genAssembler.sh [7]. Een ander gedeelte van VM_Assembler.java is meer specifiek en staat statisch in VM_Assembler.in [8]. genAssembler.sh wordt tijdens het build proces opgeroepen om, met VM_Assembler.in als extra input, VM_Assembler.java te vormen.

Op machinecode niveau biedt de code ondersteuning voor het genereren van instructies op operandbreedtes van 8-, 16-, en 32-bit. Het uitbreiden naar x86-64 behelst het invoegen van ondersteuning voor 64-bit operandbreedtes.

Indien men in een later stadium zou willen gebruik maken van de extra registers, moeten deze ook op dit niveau al ingevoerd worden. Verder zal het op bepaalde plaatsen nodig zijn SSE instructies te gebruiken. Ook hiervoor moet ondersteuning op dit niveau worden ingevoerd. Tenslotte moet ook de ondersteuning voor 64-bit constanten aangeboden worden.

In de volgende onderdelen worden de grove lijnen van de gedane veranderingen besproken. In paragraaf A.1 wordt in groter detail uitgelegd op welke punten wat werd veranderd.

### 7.5.1 Opbouw van de functies in VM_Assembler.java

De functies VM_Assembler.java hebben de volgende vorm:

```
emit<opcode>_<operand1>_<operand2>_<operandbreedte>(<lijst van params>)
```

Het enige verplichte gedeelte is `<opcode>`. Dit is de opcode van de instructie uit de x86 instructieset. Afhankelijk van het aantal operands dat de opcode verwacht, zijn `<operand1>` of `<operand2>` aanwezig. De operandbreedte wordt aangegeven door `<operandbreedte>`. Als deze niet opgegeven is, is de operandbreedte 32-bit. Voor 8-, 16- en 64-bit komt hier respectievelijk Byte, Word en Quad.

Indien aanwezig nemen `<operand1>` of `<operand2>` een van volgende vormen aan:

**Imm** De operand is een constante

**Reg** De operand is een register

**Abs** De operand is [displacement]

**RegInd** De operand is [register]

**RegDisp** De operand is [register + displacement]

**RegOff** De operand is [index<<scale+disp]

[7] rvm/src/vm/arch/intel/assembler/genAssembler.sh
[8] rvm/src/vm/arch/intel/assembler/VM_Assembler.in

**RegIdx** De operand [base+index<<scale+disp ]

**Cond** Het argument is een conditie (gelijk, ongelijk, groter en dergelijke).

De argumenten aanwezig in de lijst van argumenten komen altijd overeen met de gegevens nodig om de adresmode de encoderen.

Verder moet ook nog vermeld worden dat `<operand1>` altijd overeen komt met de plaats waar het resultaat terecht komt.

Een klein voorbeeldje:

```
emitMOV_Reg_RegDisp_Quad(byte dstReg, byte srcReg, Offset disp)
```

Met deze functie zal de machinecode worden gegenereerd voor het laden van 64-bit van op de geheugenlocatie [srcReg + disp] in het register dstReg.

### 7.5.2 Verdwenen instructies

Dit is een geheel triviaal onderdeel. Er dient gewoon verzekerd te worden dat de instructies die ongeldig geworden zijn in 64-bit mode niet meer gebruikt worden in 64-bit mode. Een lijst van deze instructies werd gegeven in paragraaf 2.6.3.

Na inspectie van VM_Assembler.in en genAssembler.sh blijkt dat van deze lijst alleen INC (0x40) en DEC (0x48) in een paar configuraties gebruikt worden. Deze instructies (op x86-32) dienen om een register de incrementeren. 0x40 + reg (met reg een van de 8 x86 registers verhoogt de waarde van dat register met 1. De 0x48 variant verlaagt de waarde. Als alternatief zal hier de opcode 0xFF/0 respectievelijk 0xFF/1 gebruikt moeten worden. De instructie zal daardoor wel iets langer worden.

Dit is trouwens een mooi voorbeeld van een opcode die een verschillende betekenis krijgt afhankelijk van de waarde van het reg gedeelte van de ModRM-byte.

### 7.5.3 Nieuwe Registers en nieuwe operandbreedte

In paragraaf 2.6.2 werd de invloed besproken van de REX-prefix op het gebruik van de nieuwe registers en de nieuwe operandbreedte.

Omdat het handig zou zijn dat iedere instructie toegang heeft tot de nieuwe registers, zal bij iedere instructie die gebruik maakt van registers deze REX-prefix ingevoegd worden. Om dit proces te vergemakkelijken werd een hulpfunctie geschreven die uit de vier meegeleverde argumenten een REX-prefix vormt.

```
private byte REXprefix(boolean W,  byte R, byte X, byte B) {
  byte bW = (byte) (W ? 0x8 : 0x0);
  byte bR = (byte)((R>>3)<<2);
  byte bX = (byte)((X>>3)<<1);
  byte bB = (byte)((B>>3));

  return (byte) (0x40 | bW | bR | bX | bB);
}
```

In elke instructie waar deze prefix van belang zou kunnen zijn, moet de REX-prefix, net voor de opcode, aan de instructie toegevoegd worden. Eventueel zou een test kunnen uitgevoerd worden om na te gaan wanneer de REX-prefix de waarde 0x40 heeft. Met deze waarde zou hij eventueel kunnen worden weggelaten. Er werd echter besloten de code eenvoudiger te houden en gewoon overal deze REX-prefix te laten staan.

Het gebruik van de functie is bijzonder eenvoudig. Afhankelijk van de gebruikte adresseermode worden de passende registers als argumenten gebruikt. Indien in de adresseermode bepaalde argumenten van geen belang zijn, wordt gewoon 0x0 meegegeven. Afhankelijk van de gewenste operandbreedte moet ook true of false worden meegegeven. Bij true wordt de operandbreedte 64-bit, bij false 32-bit of 16-bit (indien de prefix voor 0x66 wordt gebruikt).

Om de programmeur te ontlasten van het van buiten leren van de waarde toegekend aan een register, worden in VM_RegisterConstants.java[9] constanten gedefinieerd die met de naam van een register zijn waarde laten overeen komen. Voor de 8 eerste general purpose registers worden de namen zoals op x86-32 (beginnend met een E) verder gebruikt. Voor de 8 nieuwe registers worden de constanten R8, R9,. . . , R15 ingevoerd, met bijhorende waarde 0x8, 0x9,. . . ,0xF.

Een probleem duikt nog op bij het encoderen van de ModRM- en SIB-bytes. Het 4-de bit van de nieuwe registerwaarden mag hier niet in rekening gebracht worden. Deze bytes worden namelijk gevormd door een combinatie van SHIFT- en OF-operaties. In de finale samenstelling van de ModRM- en SIB-bytes moeten de registers dus tot hun 3 minst significante bits beperkt worden alvorens ze te gebruiken. Gelukkig moet dit maar op een paar plaatsen aangepast worden.

Verder was het ook nodig een functie om een 64-bit constante (long) aan een instructie toe te voegen. Op x86-32 is is alleen ondersteuning voor 8-, 16- en 32-bit constanten aanwezig.

[9] rvm/src/vm/arch/intel/VM_RegisterConstants.java

### 7.5.4 SSE ondersteuning

Ondersteuning voor de SSE/SSE2 instructies is in JikesRVM op 32-bit x86 niet aanwezig. Op oude 32-bit x86 platformen zijn niet altijd SSE-eenheden aanwezig, zodat SSE hier niet algemeen kan gebruikt worden. De nieuwe 64-bit x86-64 platformen hebben allemaal SSE/SSE2 ondersteuning. Zonder deze ondersteuning zal de x86-64 versie ook niet kunnen werken, omdat reële parameters naar C functies moeten worden doorgegeven langs xmm registers.

In wat volgt zal geen onderscheid meer gemaakt worden tussen SSE en SSE2. Er zal altijd gewoon over SSE gesproken worden. Verder is ondersteuning bieden voor alle SSE instructies zeker niet nodig. Enkel ondersteuning voor instructies die nuttig zijn voor de implementatie zal worden voorzien. Met een paar kleine aanpassingen kan de voorziening voor SSE ook beschikbaar gesteld worden voor x86-32 platformen waar SSE op aanwezig is.

Een eerste type instructies dat nodig is, zijn de instructies om de gegevens in xmm registers te laden en om gegegevens vanuit deze registers naar het geheugen te schrijven. Omdat SSE zal gebruikt worden om vlottende komma ondersteuning te bieden, worden de instructies, om float en double te verplaatsen, van het geheugen of vanuit een general purpose register, naar een xmm register, en omgekeerd, geïmplementeerd. De instructies die hiervoor zorgen zijn:

**MOVSS** Verplaatsen van een single precision floating point

**MOVSD** Verplaatsen van een double precision floating point

Vervolgens zijn er instructies nodig om gehele waarden te converteren naar reële waarden. Omdat voorlopig slechts een beperkt aantal Java bytes via SSE geïmplementeerd zal worden, volstaat hier ondersteuning voor omzetting van floating point naar gehele waarden. De twee volgende instructies worden gebruikt voor f2i, f2l, d2i en d2l:

**CVTSS2SI** Omzetten van float naar integer/long

**CVTSD2SI** Omzetten van double naar integer/long

Het leuke aan deze instructies is dat men rechtstreeks vanuit het geheugen naar een general purpose register kan omzetten. Men moet geen omweg nemen langs een xmm register. Dit is bijzonder handig bij het gebruik van de operand stack.

Omdat de conversiefuncties uit de vorige paragraaf niet op elk vlak hetzelfde resultaat genereren als de Java specificaties verwachten, zijn een paar vergelijkingen nodig om tot het goede resultaat te komen. Om deze vergelijkingen uit te voeren zijn volgende instructies vereist:

**UCOMISS** Vergelijking van een single precision floating point

**UCOMISD** Vergelijking van een double precision floating point

Deze vergelijken een xmm register met een andere xmm register of een geheugenlocatie. Het resultaat van de vergelijking past de rflags aan, zodat met de gewone spronginstructies kan gewerkt worden.

Om voor deze instructies functies aan VM_Assembler.java toe te voegen, werd aan genAssembler.sh een stuk toegevoegd. De naamgeving van de gevormde functies volgt het algemene principe. Ook de verschillende adresseermodes werden voorzien voor bovenstaande functies.

Verder dienden ook de constanten voor XMM0, XMM1,. . . ,XMM15 toegevoegd te worden aan VM_RegisterConstants.java.

### 7.5.5 Andere aanpassingen

Het enig andere bestand dat nog diende aangepast te worden was VM_Lister.java[10]. Dit wordt gebruikt om, indien gevraagd wordt de inhoud van de bootImage op het scherm te brengen, per instructie een regeltje tekst te genereren met daarin zowel de assembler als de machinecode van de instructie.

In VM_Lister.java werden enkele wijzigingen aangebracht opdat het mogelijk zou zijn ook constanten van het type long als operand van een opcode te hebben. Anders wou de code voor deze constanten niet werken.

Een verandering die nog nodig is, is ondersteuning voor xmm instructies toevoegen. Op dit moment worden deze afgedrukt alsof ze met general purpose registers werken.

## 7.6 Baseline Compiler

Het doel van de compiler is de Java byte-code om te zetten naar platformafhankelijke machinecode. Het is dus de bedoeling de door een Java byte-code compiler gegenereerde byte-codes om te zetten naar uitvoerbare x86_64 code.

Het doel en de opzet van de baseline compiler wordt besproken in paragraaf 4.3.3. Het grootste werk was nodig in VM_Compiler.java[11]. Deze bevat hulpfuncties, die door het platformafhankelijk VM_BaselineCompiler.java[12] worden opgeroepen, om de verschillende Java byte-codes om te zetten in een reeks machine-instructies. Een groot deel van de functies uit VM_Compiler.java liggen qua naam heel dicht bij een Java byte-code. Deze functies zullen dan ook de functionaliteit van een dergelijke byte-code in machinecode vertalen.

Opnieuw worden hier alleen de belangrijke ontwerpsbeslissingen besproken. De details van de aanpassingen zijn te vinden in paragraaf A.2.

[10] rvm/src/vm/arch/intel/assembler/VM_Lister.java
[11] rvm/src/vm/arch/intel/compilers/baseline/VM_Compiler.java
[12] rvm/src/vm/compilers/baseline/VM_BaselineCompiler.java

### 7.6.1 Algemene regels voor het gebruik van assembler

In dit onderdeel zal de gevolgde conventie voor het gebruik van assembler worden toegelicht. Met assembler wordt in dit geval het gebruik van de functies uit VM_Assembler.java bedoeld.

Een eerste probleem binnen VM_Compiler.java was het rechtstreeks gebruik van constanten. Tegen alle principes van overzichtelijk programmeren in, werden overal constanten rechtstreeks gebruikt. Het was hier veel beter geweest statische variabelen te gebruiken gedefinieerd in een van de vele bestanden met statische constanten. In VM_Compiler ging het meestal om het gebruik van een veelvoud van 4 in plaats van een veelvoud van WORDSIZE. Soms wordt ook 2 in plaats van LG_WORDSIZE gebruikt. Deze constanten waren nochtans gedefinieerd in VM_BaselineConstants.java[13]. Alvorens het aanpassen van VM_Compiler.java werd aangevat, was dit bestand al aangepast, en de baseline compiler had al heel wat dichter bij een werkende versie geweest na gewoon aanpassen van VM_BaselineConstants.java, indien consequent gebruik was gemaakt van deze variabelen. Overal waar een dergelijke constante werd ontdekt werd deze ook veranderd door de correcte variabele.

Om dit te verduidelijken wordt een klein voorbeeld besproken. De code voor x86-32 is te vinden in Listing 6. Het toont de code voor het laden van een element uit een array van integers. Indien men deze laat lopen op x86-64 zal deze niet werken. Het probleem schuilt hier in het gebruik van 8 in plaats van 2*WORDSIZE. Indien men deze verandering doorvoert, zal dit stuk code ook werken op x86-64. Men gebruikt hier voor het poppen van argumenten van de stack gewoon een optelling om de stackpointer te veranderen. Dit mag, maar dan moet de stackpointer ook met de correcte waarde vermeerderd of verminderd worden, en die waarde is altijd het aantal slots vermenigvuldigd met de WORDSIZE. Het is meestal in deze context dat een dergelijke verandering diende te worden doorgevoerd.

Alhoewel deze code correct zal uitgevoerd worden, voldoet deze nog niet aan de conventies

[13] rvm/src/vm/arch/intel/compilers/baseline/VM_BaselineConstants.java

```
1  asm.emitMOV_Reg_RegDisp(T0, SP, NO_SLOT);      // T0 is array
       index
2  asm.emitMOV_Reg_RegDisp(S0, SP, ONE_SLOT);       // S0 is the
       array ref
3  genBoundsCheck(asm, T0, S0);                  // T0 is index, S0 is
        address of array
4  asm.emitADD_Reg_Imm(SP, 8);       // complete popping the 2 args
5  asm.emitPUSH_RegIdx(S0, T0, asm.WORD , NO_SLOT); // push
       desired int array element
```

Listing 7.1: x86-32 codefragment

die bij het aanpassen werden gevolgd. Er werd bij het aanpassen altijd rekening gehouden met de types die men aan het gebruiken was in de instructie. Op de operand stack en in de registers worden gelijke types altijd op gelijke wijze opgeslagen:

**byte** Teken-uitgebreid naar 32-bit, dan verder nul-uitgebreid naar 64-bit

**short** Teken-uitgebreid naar 32-bit, dan verder nul-uitgebreid naar 64-bit

**char** Nul-uitgebreid naar 32-bit, dan verder nul-uitgebreid naar 64-bit

**integer** Nul-uitgebreid naar 64-bit

**long** Op de stack: opgeslagen in twee stackslots, eerst een stackslot met een 0 erin, dan een stackslot met de long erin. In een register: alleen de long waarde zelf, de 0 wordt genegeerd.

**referentie** Zoals ze is, 64-bit breed.

**float** Idem integer

**double** Idem long

Deze conventie is alleen belangrijk voor het gebruik van de operand stack en de registers. Op andere plaatsen in het geheugen, bijvoorbeeld in de stack van de lokale variabelen wordt alles opgeslagen zoals in Figuur 7.1. De grijze gedeeltes zijn daar onbelangrijk.

Het uitbreiden van byte, short en char gebeurt op die manier, omdat de bewerkingen op deze types via de integer bewerkingen van de virtuele machine worden geïmplementeerd. De manier waarop dan de integer wordt opgeslagen volgt uit de manier waarop x86-64 met 32-bit gegevens omgaat. In een register wordt het resultaat van een 32-bit bewerking altijd nul-uitgebreid naar 64-bit. Voor het werken op 32-bit zullen dan ook altijd de functies uit VM_Assembler gebruikt worden waar `<operandbreedte>` niet gedefinieerd is, en de standaard operandbreedte voor x86-64, 32-bit, wordt gebruikt. Het resultaat van een dergelijke bewerking zal dan de conventie met het nul-uitbreiden volgen. Voor bewerkingen op 64-bit moeten de functies met `<operandbreedte>` Quad gebruikt worden. Naast longs moet dus ook voor doubles en voor referenties `<operandbreedte>` Quad gebruikt worden.

Als deze regels worden toegepast op de code uit Listing 6, zijn nog enkele aanpassingen nodig. Regel 1 is correct omdat het daar om een integer gaat. Regels 2 en 4 moeten aangepast worden omdat het referenties zijn. Regel 5 is nog een apart probleem. Indien PUSH gebruikt wordt, zal 64-bit op de stack gepushed worden. De bovenste 32-bit zullen echter overeen komen met het volgende element uit de array. Daarom moet eerst een 32-bit MOV en dan een 64-bit PUSH gebruikt worden. Het resultaat van de aanpassingen aan dit stuk code zijn te vinden in Listing 7.

```
1  asm.emitMOV_Reg_RegDisp(T0, SP, NO_SLOT);          // T0 is array
       index
2  asm.emitMOV_Reg_RegDisp_Quad(S0, SP, ONE_SLOT);    // S0 is
       the array ref
3  genBoundsCheck(asm, T0, S0);                       // T0 is index, S0 is
        address of array
4  asm.emitADD_Reg_Imm_Quad(SP, WORDSIZE*2);          // complete
       popping the 2 args
5  asm.emitMOV_Reg_RegIdx(T1,S0,T0, asm.WORD, NO_SLOT); //there's
       no 32-bit push available in 64-bit mode, first MOV then PUSH
6  asm.emitPUSH_Reg(T1);
```

Listing 7.2: x86-64 codefragment

Het volgen van deze conventies is niet altijd strikt nodig. Na het aanpassen van de constante zou het stuk code ook al gewerkt hebben, behalve in het geval dat SP groter dan 31-bit zou worden. Dit laat Jikes RVM echter niet toe. Het volgen van de conventies maakt echter een stuk duidelijker wat de intenties van de code zijn, en correctheid is dan ook gemakkelijker te controleren. Ook bij bewerkingen waar overflow kan optreden is het belangrijk de conventies te volgen.

De grote moeilijkheid in het aanpassen van VM_Compiler was dus goed na te kijken met welke types in welke instructies wordt gewerkt. Bij de meeste types is dit niet zo een groot probleem. De conventies voor het uitbreiden van byte, char en short naar integer werd op 32-bit in VM_Compiler ook gevolgd. Het herkennen van het gebruik van long, double en float is omwille van hun natuur (twee stackslots) ook niet bijzonder moeilijk. Alleen het verschil tussen integer en referentie (op 32-bit) vormt een probleem. Daar is op zich geen onderscheid te merken (op 32-bit). Er moet dus telkens uit de context afgeleid worden of het om een integer of een referentie gaat, en dat is niet altijd even gemakkelijk.

### 7.6.2 Structurele aanpassingen

VM_Compiler bestaat uit drie onderdelen:

- Java byte-codes
- Hulpfuncties
- Magic

Naast de veranderingen zoals besproken in paragraaf 7.6.1 werden nog enkele meer structurele veranderingen aangebracht. Bij de hulpfuncties bleek dit nergens nodig te zijn, bij de Java byte-codes en de ondersteuning voor Magic wel.

### Java byte-codes

Een eerste structurele verandering die werd ingevoerd, was voor de bewerkingen op longs. Hiervoor diende op 32-bit telkens een algoritme gebruikt te worden om in meerdere stappen deze bewerkingen uit te voeren. Deze bewerkingen kunnen op 64-bit met een instructie uitgevoerd worden. De lange algoritmes werden dan ook overal vervangen door de passende instructie.

Een tweede verandering werd ingevoerd bij de Java byte-codes voor conversie: f2i, f2l, d2i, d2l. Deze zetten floats en doubles enerzijds om naar integer en longs anderzijds. Om deze Java byte-codes om te zetten werd, op x86-32, gebruik gemaakt van een functie geschreven in C, meegecompileerd met de bootImageRunner. Alvorens men echter zo een functie kan oproepen moet men alle te bewaren registers op de stack pushen. Ook de argumenten van de functie moeten op de stack komen. Achteraf moet dan het resultaat van de stack gehaald worden. Daarna moeten de registers hersteld worden.

Het op 32-bit uitvoeren van dergelijke acties, gebeurt met x87 instructies. Het aantal instructies om deze conversies uit te voeren, gebruik makende van x87 instructies, is echter tamelijk groot, zodat de overhead voor het oproepen van een C functie niet al te groot is. Op x86-64 kan men hier echter gebruik maken van SSE. Hiermee kunnen deze conversies uitgevoerd worden in ongeveer 10 instructies. De overhead wordt hier dan wel heel groot. Daarom werd van de SSE ondersteuning gebruik gemaakt om zelf de machinecode te genereren voor deze conversies, en niet meer rond te gaan langs een C functie.

### Magic

Bij het porteren naar 64-bit PPC waren al enkele plaatsen aangeduid waar de instructies voor de magic niet meer zouden werken. Deze dienden natuurlijk nagekeken te worden en waar nodig aangepast.

Het probleem bij deze aanpassingen was vooral de correcte semantiek van de magic vinden. Dit is niet altijd eenduidig af te leiden uit de naam. Door iets dieper in de code te duiken, kan meestal de betekenis wel achterhaald worden.

De enige echt structurele wijziging werd uitgevoerd aan de magic voor het implementeren van een syscall. Een dergelijke call komt overeen met het aanroepen van een van de C functies uit Sys.c[14]. Zoals in paragraaf 2.7 wordt besproken, kan dit niet op dezelfde wijze gebeuren

[14] rvm/src/tools/bootImageRunner/Sys.c

als op x86-32. Het op deze manier doorgeven van parameters vanuit de Java code is een niet zo eenvoudige zaak. Het probleem is dat het laatste argument eerst op de stack moet geplaatst worden en dan het voorlaatste en zo verder. Het probleem dat zich stelt is dat de eerste 6 gehele en de eerste 8 reële parameters langs registers, en niet op de stack moeten worden doorgegeven. De lijst met parameters moet daarom twee keer doorlopen worden. De eerste maal wordt aangeduid welke parameters in registers moeten doorgegeven worden en welke niet. De eerste maal wordt de lijst hiervoor van het eerste naar het laatste argument doorlopen. Een tweede maal wordt de lijst van achter naar voren doorlopen en met de opgedane kennis kan dan besloten worden het resultaat ofwel in een register ofwel op de stack te plaatsen.

Er werd ook voor gezorgd dat voldaan wordt aan de randvoorwaarden:

- De stack is correct gealigneerd op $16n + 8$ bytes
- AL bevat een bovengrens voor het aantal gebruikte xmm registers (8).

### 7.6.3 Andere aanpassingen

Zoals de naam VM_BaselineConstants.java[15] al doet vermoeden, bevat deze constanten die gebruikt worden door de baseline compiler. Aan dit bestand waren twee types aanpassingen nodig. Op de eerste plaats zijn enkele constanten gedefiniëerd voor het gebruik van de stack, specifiek binnen de baseline compiler. Zoals gewoonlijk dienden deze voor x86-64 in grootte verdubbeld worden, afhankelijk van de afstand in stackslots waarvoor de constante bedoeld is.

Verder worden hier ook de namen van de registers vastgelegd voor het gebruik in VM_Lister. Door deze aan te passen aan de benamingen met een r, volgens de x86-64 conventie, worden ook deze benamingen gebruikt in de output van de bootImage.

[15] rvm/src/vm/arch/intel/compilers/baseline/VM_BaselineConstants.java

# Hoofdstuk 8

# Besluit

Ondanks een paar uitbreidingen is de x86-64 instructieset heel gelijklopend aan de x86-32 instructieset. Desondanks is het werk aan een project, dat bijzonder veel gebruik maakt van machinecode (en bij uitbreiding assembler) tamelijk groot. Verder wordt bij ontwerp van software nog altijd te weinig rekening gehouden met onderhoud en uitbreidbaarheid. Het is dan ook soms heel moeilijk bepaalde problemen te voorzien.

Bij het schrijven van de platformafhankelijke code voor x86-64 had waarschijnlijk niemand ooit gedacht dat deze nog ging uitgebreid worden om ook te werken op een 64-bit systeem. Bij de uitbreiding werd meer rekening gehouden met mogelijke uitbreidingen, onderhoud en duidelijkheid van de code. Dit werd gedaan door voornamelijk via variabelen te werken in plaats van constanten. Ook het vervangen van types van variabelen via de precompiler werd hiervoor gebruikt.

De moeilijkheid van het aanpassen van een component die machinecode genereert, mag zeker niet onderschat worden. Een zeer grondige kennis van de architectuur is hiervoor vereist. Een verkeerde veronderstelling is al te rap gemaakt, en instructies moeten nu eenmaal correct geëncodeerd worden. Vooral de REX-prefix in combinatie met de ModRM- en SIB-bytes is een lastig stuk in de x86-64 instructieset. De manier waarop de REX-prefix is opgevat maakt het encoderen van een instructie nog moeilijker, dan het als was op x86-32 met diezelfde ModRM- en SIB-bytes. Een eenvoudiger oplossing was er vermoedelijk echter niet, toch niet als men het systeem van ModRM- en SIB-bytes gewoon wou behouden. Bytes maken met meer dan 8-bit is dan ook niet echt mogelijk. Het compleet herwerken van de x86 instructieset was een andere optie geweest. Dit had echter op veel minder succes bij de consument kunnen rekenen. De compabiliteitsmode en de legacy modes zijn zeker voor industriële toepassingen van groot belang, want men zal daar een goed werkend product echt niet herschrijven.

Verder is het aanpassen en proberen snappen van assemblercode, of semi-assemblercode zoals in VM_Compiler niet altijd even gemakkelijk. Zo lang het aantal instructies van een functie

beperkt blijft is het meestal goed te doen om, samen met de functionele naam, en eventuele commentaar, de functie te begrijpen. Naarmate het aantal instructies oploopt, loopt ook de tijd nodig om de exacte bedoeling en werkwijze van de functie te doorgronden sterk op. Dit maakte het controleren en aanpassen van VM_Compiler een bijzonder lastige job.

Alhoewel er nog wel wat werk aan de winkel is om Jikes RVM op x86-64 op hetzelfde niveau te tillen als de x86-32 versie, is een goed fundament al gelegd. Het grootste deel van de baseline compiler is aangepast. Enkele stukjes binnen VM_Compiler moeten nog nagekeken worden, maar voor een groot aantal Java bytes werden al testen uitgevoerd. De meeste geteste Java byte-codes bleken te werken, en daar waar fouten ontdekt werden, werden deze weggewerkt. Het is nu een kwestie van de resterende, niet geteste, Java byte-codes te testen, en indien ze niet blijken te werken te kijken waar in VM_Compiler het mis loopt. Dit laatste is wel een lastig werkje. Met de werkwijze uitgewerkt in 5.3 moet men dan zoeken naar de plaats waar het mis loopt. Opnieuw geldt dat het niet altijd even eenvoudig is de fout te zoeken in een reeks assembler instructies.

Met de versie zoals ze op dit moment is, is het al mogelijke verschillende demo's te draaien. Delers bepalen van gehele getallen, genereren van Fibonaci getallen, allemaal algoritmen die al perfect mogelijk zijn. Voor bewerkingen op longs bleek de x86-64 versie al 10% sneller te lopen. Bij de optimizing compiler zal het verschil vermoedelijk nog groter zijn.

Daarnaast is ook het begin van ondersteuning voor SSE een goede zaak. Vanuit de Jikes RVM community, is op de website een aanvraag gekomen om SSE te ondersteunen. Hiervoor werd al een grond gelegd. Met een klein aantal aanpassingen, waaronder het invoeren van een nieuwe variabele - bijvoorbeeld RVM_WITH_SSE - in het configuratiebestand, kan men ervoor zorgen dat SSE beschikbaar is voor andere platformen dan x86-64. Indien een dergelijk platform dan SSE ondersteuning wil gebruiken, hoeft het deze variabele maar op 1 te zetten in het configuratiebestand. Er moet dan wel op gelet worden dat op deze platformen geen REX-prefix meer gebruikt wordt voor het encoderen van de SSE instructies, zoals nu wel standaard gebeurd. De aanpassing hiervoor is echter ook minimaal van zodra de variabele RVM_WITH_SSE bestaat.

Met de verschillen tussen x86-32 en x86-64 indachtig, zou men zich kunnen afvragen of het uiteindelijk resultaat - een Jikes RVM op x86-64 - nuttig zal zijn. Op het vlak van de grotere adresruimte is het antwoord hier zeker neen. Dit komt voornamelijk omdat Jikes RVM voorlopig maar werkt met adressen tot 32-bit. Het is zeker geen slecht idee om te onderzoeken in hoeverre een grotere adresruimte binnen Java mogelijk is, en vooral wat het effect hiervan dan is op de performantie. Vermoedelijk zal ook het geheugenbeheer dan soms ook anders moeten opgevat worden. Gelukkig zorgt de mogelijkheid om bewerkingen op longs uit te voeren wel voor een voordeel. Dit komt doordat deze bewerkingen rechtstreeks kunnen uitgevoerd worden in plaats van via een algoritme van een aantal instructies, zoals op x86-32. Een nadeel is dan weer het encoderen van de REX-prefix in elke instructie. Hierdoor is het

iets ingewikkelder een instructie te encoderen, en zal dit ook niet ietsje langer duren.

Uiteindelijk zal verder onderzoek moeten uitwijzen hoe het gedrag tussen x86-32 en x86-64 nu uiteindelijk verschilt. Vermoedelijk zal dit, zoals bij elk platform, afhankelijk zijn van applicatie tot applicatie. Interessant aan het x86-64 platform is, dat indien men weet van een applicatie dat ze beter in 32-bit draait, men dit ook gewoon kan doen. Een leuke extra zou dan kunnen zijn een rvm32 en rvm64 te voorzien. Hierbij is de eerste dan een Jikes RVM versie gebouwd in Compatibility Mode en de tweede een echte x86-64 versie. Het is dan kwestie te kiezen voor de juiste Jikes RVM op het juiste moment.

# Bijlage A

# Details Aanpassingen

Dit deel bevat een log van de veranderingen aangebracht aan volgende bestanden:

- VM_Assembler.java
- VM_Compiler.java

Niet alle veranderingen zijn opgenomen, alleen de meeste belangrijke. Verder worden de beweegredenen voor de veranderingen ook opgegeven. Het moet gezien worden als een geschiedenis van de beslissingen tijdens de eerste keer dat deze bestanden werden doorlopen. Nadat de bootImage kon geboot worden, zijn nog aanpassingen gemaakt en fouten verbeterd die hier niet meer opgenomen zijn.

## A.1 VM_Assembler.java

VM_Assembler.java bestaat uit twee delen. Een eerste deel is statisch en opgeslagen in VM_Assembler.in[1]. Een tweede deel wordt gegenereerd door genAssembler.sh[2] en achter de inhoud van VM_Assembler.in in VM_Assembler.java geplaatst.

### A.1.1 VM_Assembler.in

Voor het verkleinen van de registers op 3 bit werd een constante REGTOP gedefiniëerd:

```
private static final byte REGTOP = 0x7;
```

Ox7 komt overeen met 111, en dus zorgt

[1] rvm/src/vm/arch/intel/assembler/VM_Assembler.in
[2] rvm/src/vm/arch/intel/assembler/genAssembler.sh

`reg & REGTOP`

voor het gewenste resultaat.

Het gebruik van geheugen en registers is, in de meeste instructies, vooral afhankelijk van de opbouw van de ModRM- en (eventueel) de SIB-byte. Daarom is in VM_Assembler.in ondersteuning voor de verschillende adresmodi voorzien:

**emitRegRegOperands(reg1,reg2)** BEW reg1, reg2

**emitRegDispRegOperands(reg1,disp,reg2)** BEW [ reg1 + disp ] reg2

**emitRegIndirectReg(reg1,reg2)** BEW [reg1] reg2

**emitRegOffRegOperands(index,scale,disp,reg)** BEW [index<<scale+disp] reg

**emitRegAbsRegOperands(disp,reg)** BEW [disp] reg

**emitSIBRegOperands(base,index,scale,disp,reg)** BEW [base+index<<scale+disp] reg

Afhankelijk van de opcode is het complexe gedeelte het eerste argument en het register het tweede , of omgekeerd. Voor ADC heeft men bijvoorbeeld ADC reg/mem reg met opcode 0x11 en ADC reg reg/mem met opcode 0x13. Het reg/mem gedeelte dient dus vervangen te worden door één van deze bovenstaande 'complexe modi'. Deze modi kunnen met bepaalde instructies ook met een constante gebruikt worden. ADC reg/mem imm is 0x81/2. De /x slaat hier op de (decimale) waarde van het reg-gedeelte van het ModRM-byte.

Aan deze functies moest niet veel veranderd worden. Op bepaalde plaatsen werd een registervariabele vergeleken met een registerconstante (ESP meestal), omdat bepaalde speciale gevallen op die manier geëncodeerd worden. Hier diende REGTOP gebruikt te worden omdat de vergelijking ook opging voor overeenkomstige register met het $4^{de}$ bit op 1.

In emitAbsRegOperands werd een speciale mode gebruikt die niet meer dezelfde betekenis had in 64-bit mode. Hier diende een omweg geïmplementeerd te worden via een SIB-byte om hetzelfde gedrag als op x86-32 te behouden.

Deze instructies maken allemaal gebruik van 5 functies om zo alle gevallen van ModRM- en SIB-bytes te kunnen vormen:

- byte regIndirectRegModRM(reg1,reg2)
- byte regDisp8RegModRM(reg1,reg2)
- byte regDisp32RegModRM(reg1,reg2)
- byte regRegModRM(reg1,reg2)

- byte sib(scale,baseReg,indexReg)

De eerste 4 gevallen zijn ModRM-bytes met het mod-gedeelte op 00,01,10,11. Omdat de registers die doorgegeven worden uit 4 ipv 3 bits kunnen bestaan, en het 4de bit in de REX-prefix geëncodeerd moet zitten, worden al de registers op 3 bits verkleind, want de vorming van deze 5 bytes is een combinatie van OR en SHIFT en dus zou het 4-de bit een deel van het byte kunnen beïnvloeden waar het niet aan mag komen.

Er werd gekozen het verkleinen van de registers alleen maar helemaal op het einde uit te voeren, omdat anders in alle instructies deze actie zou moeten uitgevoerd worden, en dit niet alleen veel werk, maar ook een grote kans op fouten met zich mee zou brengen.

Vervolgens werd een functie createREX gedefiniëerd, om via genAssembler.sh, alle 16 mogelijke REX-prefixen te creëren, in een vorm REXWRXB, met W,R,X,B aanwezig of afwezig afhankelijk van de waarde van het gelijknamige bit in de REX-prefix.

```
private static byte createREX(boolean W, boolean R, boolean X, boolean B) {
  byte bW = (byte) (W ? 0x8 : 0x0);
  byte bR = (byte)((R>>3)<<2);
  byte bX = (byte)((X>>3)<<1);
  byte bB = (byte)((B>>3));

  return (byte)(0x40 | bW | bR | bX | bB);
}
```

Omdat het in genAssembler.sh in vele gevallen niet mogelijk bleek, de REX-prefix op voorhand vast te leggen, werd nog een tweede functie REXprefix geschreven, die afhankelijk van de waarde van de registers de correcte REX-prefix genereert.

```
private byte REXprefix(boolean W,  byte R, byte X, byte B) {
  byte bW = (byte) (W ? 0x8 : 0x0);
  byte bR = (byte)((R>>3)<<2);
  byte bX = (byte)((X>>3)<<1);
  byte bB = (byte)((B>>3));

  return (byte) (0x40 | bW | bR | bX | bB);
}
```

In genAssembler.sh werd bij de meeste general purpose instructies deze prefix geplaatst. Als alleen aanpassing voor de nieuwe registers nodig is, mag W false zijn. Wanneer 64-bit operandbreedte vereist is dient deze op true te staan.

Op x86 was de grootst mogelijke constante 32-bit. Op x86_64 is dit 64-bit. Een functie om 64-bit constante in instructie te encoderen ontbrak dus. Deze werd dan ook toegevoegd.

Vervolgens is er ondersteuning aanwezig voor patches in de code en voor table switches. Op deze plaats diende niet echt iets gewijzigd te worden.

Het laatste deel van VM_Assembler.in bestaat uit hand-gecodeerde instructies. Bij deze dient dus na gegaan te worden, welke uitgebreid kunnen worden naar 64-bit operandbreedtes en welke ook naar de nieuwe registers.

De eerste groep instructies waren de CMOVcc instructies. Op 32-bit werden deze alleen maar voor 32-bit operandbreedtes geïmplementeerd. Dit wordt ook zo verder gezet. Indien een CMOVcc voor 64-bit nodig zou blijken in de toekomst kan dit nog altijd uitgebreid worden.

Vervolgens is er de groep instructies voor SETcc. Deze zijn 8-bit instructies, dus 64-bit operandbreedte komt hier niet ter sprake. Alleen uitbreiding naar de nieuwe registers is nodig.

Dan is er een reeks van handgecodeerde IMUL-instructies. Dit zijn de instructies van de vorm IMUL reg reg/mem. Deze werden op 32-bit alleen voor 32-bit geïmplementeerd. Voor de implementatie van de java byte codes zijn zowel de 32-bit als de 64-bit vermenigvuldiging nuttig. Deze worden dus beiden voorzien.

Vervolgens is er een functie om interrupts op te roepen. Deze verandert niet in 64-bit mode.

Dan zijn er enkele functies voor conditionele sprongen. Deze dienen niet aangepast te worden, daar ze alleen met constanten werken.

Vervolgens is er een groep functies die de verschillende gevallen van de LEA instructie bevatten. Ook voor LEA werd alleen de 32-bit versie geïmplementeerd, aangezien in 64-bit mode adressen 64-bit zijn, hoeven we nu alleen nog maar de 64-bit versie te implementeren. Natuurlijk moet ook ondersteuning voor de nieuwe registers voorzien worden.

Dan is er een functie die gebruik maakt van de onmiddellijke versie van MOV om een constante in een register te laden. 0xB8 + reg, gevolgd door een constante, laadt de constante in dat register. Dit is de enige instructie in de instructieset om een 64-bit constante op te laden. Voor verder gebruik binnen de virtuele machine zijn alleen de 32-bit en 64-bit nodig, en alleen deze worden dus voorzien.

Vervolgens wordt er ondersteuning geïmplementeerd voor RET en ENTER. Hier is geen REX-prefix nodig, en er zijn geen registers als argumenten, dus moet hier niets veranderd worden.

Dan is er ondersteuning voor CDQ, dat EAX teken-uitbreid naar EDX (op 32-bit). Op 64-bit noemt deze functie CDO. Er wordt ondersteuning voorzien zowel voor CDQ en CDO omdat deze onder andere nodig zullen zijn bij de 32-bit, respectievelijk 64-bit, deling.

Vervolgens is er ondersteuning voor NOP. Het hoeft geen betoog dat voor deze instructies geen aanpassingen nodig zijn.

Daar onder bevindt zich ondersteuning voor x87 floating point operaties. Het goede nieuws is dat daar niets aan veranderd moet worden.

### A.1.2 genAssembler.sh

Volgende fucnties maken gebruik van de adresmodes hierboven beschreven:

**emitBinaryReg** Deze creëert voor een bepaalde opcode alle bovenstaande opgenoemde adresmogelijkheden. En dit in beide richtingen. Verder biedt deze ook ondersteuning om deze functies te laten genereren voor 8-bit, 16-bit en 32-bit grootte.

**emitBinaryImmWordOrDouble** Hier gaat het niet om 2 registers, maar om een registermode en een constante. Dit in 16-bit en 32-bit grootte. Ook wordt de ondersteuning voor het rechtstreekse bewerkingen op een constante en rAX. Men krijgt dan een instructie van de vorm: opcode constante.

**emitBinaryImmByte** Deze biedt dezelfde functionaliteit als emitBinaryImmWordOrDouble, maar dan voor bewerkingen met een byte.

Bewerkingen op bytes worden vaak apart beschouwd. Deze bezitten een aparte opcode en kunnen niet via een prefix gegenereerd worden zoals de 16-bit en de 64-bit versies van de standaard 32-bit operandbreedte.

Er dient wel onthouden te worden dat constanten maximaal 32-bit mogen zijn. De enige instructies die werken met 64-bit constanten zijn de MOV-instructies om 64-bit constanten naar een register te laden.

Deze 3 emit-functies worden dan nog eens herhaaldelijk opgeroepn in emitBinaryAcc om alle verschillende mogelijke gevallen te genereren. emitBinaryReg wordt hierin 4 keer opgeroepen, eenmaal voor een van de vier operandbreedtes. Dan nog eens 3 keer de emitBinaryImmWordOrDouble, voor 3 grootste operandbreedtes, en één maal de emitBinaryImmByte, voor de byte.

Instructies die door emitBinaryAcc gedraaid worden moeten dus beschikken over opcodes om volgende bewerkingen mogelijk te maken:

- BEW imm16-32
- BEW imm8
- BEW reg/mem16-32-64 reg

- BEW reg/mem8 reg
- BEW reg reg/mem16-32-64
- BEW reg reg/mem8
- BEW reg8 imm8
- BEW reg16-32-64 imm16-32-32
- BEW reg16-32-64 imm8

Bovenstaande functies werden dus aangepast om enerzijds 64-bit operandbreedtes te ondersteunen en anderzijds toegang te bieden tot de 8 nieuwe registers.

Volgende instructies maken gebruik van emitBinaryAcc:

- ADC
- ADD
- AND
- CMP
- OR
- SBB
- SUB

Vervolgens wordt emitBinaryReg nog eens toegepast op MOV en CMPXCH. Bij deze laatste duikt nog een probleem op ten opzichte van de x86-code. CMPXCH is een opcode bestaand uit 2 bytes. Om dit probleem op x86-32 te vermijden hebben ze het eerste byte als een prefix opgevat. De plaats van een prefix is echter altijd voor de REX-prefix, en dat van een eerste byte van een twee-byte opcode is na de REX-prefix. De functie emitBinaryReg moet dus nog verder aangepast worden om dit probleem op te lossen. Gelukkig beginnen alle opcodes die uit 2 bytes bestaan met 0x0F. Hiervoor wordt in de functie emitBinaryReg een nieuwe variable toegevoegd (twobyteop). Als het laatste argument van emitBinaryReg, normaal de operandbreedte of een ander prefix, 0x0F is, wordt twobyteop ingesteld om 0x0F toe te voegen aan de instructie na het rexprefix, anders gebeurt er gewoon niets.

Vervolgens is er de functie emitCall. Deze genereereert de mogelijke gebruiken voor JMP (Near) en Call (Near). Deze kunnen opgeroepen worden ofwel met een offset ofwel met een adresseercombinatie. De opbouw van de instructies komt heel sterk overeen met emitBinaryWordOrDouble. Er wordt echter alleen ondersteuning geboden voor 8- en 32-bit sprongen.

Operandbreedte prefixen zijn hier niet nodig, enerzijds omdat geen gebruik gemaakt wordt van 16-bit, en anderzijds omdat de x86-64 instructieset geen sprongen groter dan 32-bit ondersteunt via deze methode. Er wordt wel rechtstreeks ten opzichte van van RIP gewerkt, dat is de 64-bit versie van de instructiepointer. Aan deze functie moest dus alleen ondersteuning worden toegevoegd voor de 8 nieuwe registers.

Dan is er de functie emitUnaryAcc. Deze is bestemd voor bewerkingen die inwerken op 1 argument. Voor emitUnaryAcc zijn dit

- DEC
- INC
- NEG
- NOT

In 32-bit en legacy mode hebben DEC en INC een vorm die toelaat om in één byte een register te decrementeren, respectievelijk te incrementeren. Zoals eerder besproken is deze mode weg. De instructies hebben hier de vorm BEW reg/mem. Er moet dus zowel ondersteunig voor 64-bit operandbreedte als voor de nieuwe registers ingebakken worden.

Vervolgens is er de functie emitMD. Deze is er voor vermenigvuldiging en deling:

- DIV
- IDIV
- MUL
- IMUL

In de 32-bit versie werd alleen de 32-bit vermenigvuldiging en deling geïmplementeerd. Net zoals de MUL-instructies in VM_Assembler.in wordt hier een 32- en 64-bit versie voorzien.

Dan is er ook de functie emitMoveSubWord. Deze dient om alle gevallen van

- MOVSX
- MOVZX

te genereren. Het doel is om een waarde uit een 8- of 16-bit register te plaatsen in een register dat groter is. MOVSX doet dit met teken-uitbreiding, MOVZX doet dit met nuls-uitbreiding. In de 32-bit versie van Jikes wordt alleen de uitbreiding van 8- en 16-bit naar 32-bit gebruikt. Deze instructies worden gebruikt om bytes, shorts en chars op correcte wijze naar 32-bit uit te

breiden. Dit moet gedaan worden omdat bewerkingen op deze types niet ondersteund worden door de Java specificatie en via bewerkingen op integers moet gewerkt worden.

Omdat het gedrag van deze instructies behouden moet worden, wordt ondersteuning voorzien voor de 32-bit versie van deze instructies. Het resultaat hiervan wordt dan nul-uitgebreid opgeslagen in een 64-bit register.

Dan is er ook nog de functie emitShift. Deze dient om de verschillende gevallen van

- SAL
- SAR
- SHL
- SHR
- RCL
- RCR
- ROL
- ROR

instructies te geneneren. Deze gevallen zijn

- shift reg/mem 1
- shift reg/mem CL
- shift reg/mem imm8

Hier is CL het 8-bit register van RCX, de minst significante bits van RCX dus. Hier werd extra ondersteuning voor 64-bit operandbreedtes en ondersteuning voor de nieuwe registers voorzien.

Vervolgens is er de functie ShiftDouble. Deze dient om de verschillende gevallen van volgende instructies te genereren:

- SHLD
- SHRD

Deze hebben volgende syntax:

- shift reg/mem reg imm8

- shift reg/mem reg CL

Op 32-bit Jikes RVM werd alleen ondersteuning geboden voor de 32-bit versie van deze functie. Vermoedelijk om long's te shiften. Omdat dit, op x86-64, gaat met de gewone 64-bit shift operaties, zullen we deze functie ook uitbreiden naar twee keer 64-bit. Natuurlijk wordt ook ondersteuning geboden voor de nieuwe registers. De instructie wordt in de x86-64 versie niet meer gebruikt, dus veel maakt het op zich niet uit.

Vervolgens is er emitStackOp. Deze regelt de gebruiken van

- PUSH
- POP

Volgende acties worden ondersteund:

- stackop reg
- stackop reg/mem
- stackop imm8
- stackop imm32

De laatste twee operaties zijn alleen maar toegankelijk voor POP. Ook hier werd, op x86-32, alleen de 32-bit versie geïmplementeerd. Deze instructies worden automatisch 64-bit in 64-bit mode. Alleen ondersteuning voor de nieuwe registers dient dus toegevoegd te worden. Een probleem dat hier opduikt, is dat er geen ondersteuning is voor het pushen van 64-bit constanten aanwezig is. Indien men een 64-bit constante op de stack wil plaatsen, moet een omweg via een MOV genomen worden.

Vervolgens zijn er nog een heleboel functies om met de x87 instructies om te gaan. Deze moeten echter niet aangepast worden voor 64-bit mode.

Er is dan nog slechts één functie op general purpose instructions: emitMoveImss. Deze dient voor de MOV reg/mem imm te coderen. De mogelijkheden zijn:

- MOV reg/mem8 imm8
- MOV reg/mem16 imm16
- MOV reg/mem32 imm32
- MOV reg/mem64 imm32

Hier diende dus uitbreiding naar de nieuwe registers en naar de 64-bit operandbreedtes geïmplementeerd te worden.

Onderaan genAssembler.sh werd ook ondersteuning voor SSE ingevoerd.

Eerst wordt ondersteuning voor MOVSS en MOVSD voorzien. De mogelijke adresmodes komen overeen met de gewone x86 adresmodes. De rechtstreekse registermodes zijn nu wel vervangen door rechtstreekse XMM-registers.

- MOVSS xmm1, xmm2/mem32
- MOVSS xmm1/mem32, xmm2

Voor MOVSD bekomt men hetzelfde maar met mem64 in plaats van mem32.

Om deze instructies te laten genereren, wordt gebruik gemaakt van een aangepaste versie van emitBinaryReg, emitSSEMOV. Aanpassingen waren nodig omdat MOVSS en MOVSD een bytecode hebben die uit 3 delen bestaat. Verder moet er geen ondersteuning voorzien worden voor verschillende formaten (byte,word,...) van de instructie. Natuurlijk wordt ook ondersteuning voor de volledige 16 GPRS en XMM-registers voorzien via de REX-prefix.

Vervolgens is er een functie voorzien om voor de conversies van vlotten komma getallen naar gehele getallen te verzorgen. Deze functie is opnieuw een aangepaste versie van emitBinaryReg. emitSSECONV werkt ook met 3-delige byte-codes. Hier werd de ondersteuning voor verschillende operandbreedtes behouden, omdat afhankelijk van de REX-prefix, naar integer of naar long, wordt geconverteerd.

Met emitSSECONV werd ondersteuning voorzien voor volgende opcodes:

- CVTTSS2SI
- CVTTSD2SI

Deze dienen om floats of doubles om te zetten naar integers of longs.

Een aangepaste versie van emitSSECONV zorgt voor UCOMISS en UCOMMISD. Een aanpassing was nodig omdat UCOMISS een 2-delige instructie is en UCOMISD een 3-delige. UCOMISS en UCOMMISD worden gebruikt voor het vergelijken van floats respectievelijk, doubles.

## A.2 VM_Compiler.java

### A.2.1 Java byte-codes

Op veel plaatsen in de code werd, in de plaats van de constante WORDSIZE, zijn waarde (4 of 32-bit en 8 op 64-bit) gebruikt. Daar waar de 4 door WORDSIZE vervangen kon worden,

zodat op slag de code ook voor 64-bit in orde was, werd dit gedaan, zonder dit hier expliciet te vermelden.

Verder worden uitbreidingen waar bewerkingen op adressen van 32-bit naar 64-bit werden uitgebreid, door het achtervoegsel _Quad aan de functie toe te voegen niet als structurele veranderingen gezien. Deze werden dus ook niet altijd vermeld.

In paragraaf A.2.1 wordt een opsomming gegeven van de de bytecodes waar geen structurele aanpassingen voor nodig waren. In paragraaf A.2.1 worden de bytecodes vermeld waar dit wel het geval was. Er wordt ook telkens vermeld welke structurele aanpassingen precies vereist waren.

### Niet aangepaste Java byte-codes

In Tabel A.1 worden de niet aangepaste Java byte-codes opgesomd.

### Aangepaste Java byte-codes

**dconst_1** Op 32-bit zit de double in 2 stackslot's. Op 64-bit zit die in 1 stackslot, maar is er ook nog een stackslot met een O om aan de specificaties te voldoen. Het onmiddellijk pushen van de 64-bit constante is hier niet mogelijk. Omdat de instructieset alleen maar pushen tot 32-bit constanten ondersteunt, moet via MOV de constante in een register geladen worden en dan gepushed.

**ldc** De werkwijze op 32-bit PUSHED rechtstreeks 32-bit vanuit het geheugen. Op 64-bit is geen 32-bit PUSH beschikbaar. De correcte waarde dient dus eerst in een register geladen te worden en dan gepushed te worden.

**ldc2** Met ldc wordt een long of double geladen vanuit de 'runtime constant pool'. Op 32-bit zit dit in 2 stackslots. Op 64-bit, moet eerst een 0 gepushed worden en dan de geladen constante.

**lload** Idem als ldc2 maar dan vanuit het huidige frame.

**dload** Idem als lload maar dan voor een double in plaats van een long.

**iastore** Op 32-bit kon, met de referentie naar het begin van de array en de index binnen de array, rechtstreeks het correcte element op de stack gepushed worden. PUSH is echter niet beschikbaar voor 32-bit in 64-bit mode. De oude methode zou dus het gevolg hebben dat er 64-bit uitgelezen wordt, een long dus. Via een MOV kan wel 32-bit geladen worden, met nul-uitbreiding, in een 64-bit register en dan correct op de stack gepushed worden.

| | | | |
|---|---|---|---|
| aconst_null | iconst | lconst | fconst_0 |
| fconst_1 | fconst_2 | dconst_0 | iload |
| fload | aload | istore | lstore |
| fstore | dstore | astore | laload |
| daload | baload | caload | saload |
| iastore | fastore | aastore | bastore |
| castore | sastore | pop | pop2 |
| dup | dup_x1 | dup_x2 | dup2_x1 |
| swap | idiv | irem | fadd |
| fsub | fmul | fdiv | frem |
| fneg | dadd | dsub | dmul |
| ddiv | drem | dneg | i2f |
| l2f | l2d | d2f | i2b |
| fcmpl | fcmpg | dcmpl | ifeq |
| ifne | iflt | ifge | ifgt |
| ifle | if_icmpeq | if_icmpne | if_icmplt |
| if_icmpge | if_icmpgt | if_icmple | if_acmpeq |
| if_acmpne | ifnull | ifnonnull | goto |
| goto_w | ret | tableswitch | lookupswitch |
| ireturn | lreturn | freturn | dreturn |
| areturn | return | multianewarray | arraylength |
| athrow | checkcast | instanceof | monitorenter |
| monitorexit | imul | | |

Tabel A.1: Niet aangepaste Java byte-codes

**faload** Analoog iastore

**aaload** Alhoewel referenties 64-bit zijn, kan hier de oude structuur behouden worden. Men kan rechtstreeks 64-bit pushen. Alleen de schaal binnen de SIB-byte moet aangepast worden van 4 naar 8.

**lastore** Op 32-bit moet het opslaan van een 64-bit waarde in 2 keer gebeuren. Op 64-bit kan dit in één keer. Er mag echter ook niet vergeten worden het lege stackslot te verwijderen.

**dastore** Analoog lastore

**dup2_x2** De implementatie voor 32-bit maakt gebruik van 4 registers. Er zijn echter maar 3 registers beschikbaar. Om hier een mouw aan te passen, wordt het register dat gebruikt wordt om de JTOC in te bewaren als 4de register gebruikt, en daarna via een omweg

terug op zijn correcte waarde gesteld. In 64-bit mode zijn er nu echter 8 nieuwe registers. R8 kan nu gebruikt worden in plaats van JTOC.

**iadd** Om correct te werken volgens de voorgestelde methode, worden de argumenten eerst van de stack gehaald, vervolgens wordt de bewerking uitgevoerd, en dan op de stack gepushed. Het zou kunnen zijn dat dit niet strict noodzakelijk is. Dit moet in de toekomst dan eens nagekeken worden.

**isub** Analoog iadd

**ineg** Analoog iadd

**ishl** Analoog iadd

**ishr** Analoog iadd

**iushr** Analoog iadd

**iand** Analoog iadd

**ior** Analoog iadd

**ixor** Analoog iadd

**iinc** Hier wordt, ongeveer analoog als bij iadd, rechtstreeks op de stack gewerkt. Alleen wordt hier helemaal niets van de stack gehaald. iinc incrementeert rechtstreeks een argument. Zoals bij iadd kan dit tot problemen leiden. Hier zal dus de te incrementeren waarde eerst naar een register gecopiëerd moeten worden. Vervolgens kan dit register dan 32-bit geïncrementeerd worden. Als laatste moet het dan gewoon terug gekopiëerd worden naar zijn oorspronkelijke locatie. Vermoedelijk kan dit ook correct uitgevoerd worden rechtstreeks. Dit dient echter getest te worden.

**ladd** Op 32-bit moeten de 2 helften van de long apart opgeteld worden. Bij de hoge helft moet dan het carry-bit van de lage helft opgeteld worden. Op 64-bit kan dit in één keer. Er mag alleen niet vergeten worden het tweede stackslots te verwijderen van de stack.

**lsub** Analoog ladd

**lmul** Om lmul op 32-bit uit te voeren moet in een 10-tal stappen gewerkt worden. Op 64-bit kan dit rechtstreeks. De methode om de imul-vermenigvuldiging op 32-bit uit te voeren, kan hier gebruikt worden.

**ldiv** Op 32-bit wordt voor de long deling gebruikt gemaakt van een routine geschreven in C. Omdat er nu een 64-bit deling is, is deze routine op 64-bit niet meer nodig. De deling voor 32-bit int's kan gebruikt worden. Een kleine aanpassing is nodig omdat het lege stackslot voor een long nog altijd in de weg zit.

**lrem** Analoog ldiv

**lneg** Op 32-bit moet dit in 3 stappen gebeuren. Met de aanwezigheid van een 64-bit NEG kan de methode voor de 32-bit ineg overgenomen worden.

**lshl** Analoog lmul

**lshr** Analoog lmul

**lushr** Analoog lmul

**land** Analoog ladd

**lor** Analoog ladd

**lxor** Analoog ladd

**f2i** Voor f2i wordt in de 32-bit versie naar een C-routine gesprongen die aanwezig is in sys.C, een onderdeel van de BootImageRunner. De staat van alle registers moet opgeslagen worden alvorens een dergelijke C-routine kan gebruikt worden. Dit is een enorme overhead voor het uitvoeren van enkele instructies, althans op 64-bit. Op 32-bit moet voor de implementatie een groot aantal x86 en x87 gebruikt worden. Op 64-bit kan dit in 10 instructies door gebruik te maken van SSE/SSE2.

**f2l** Analoog f2i

**d2i** Analoog f2i

**d2l** Analoog f2i

**i2l** Op 32-bit wordt een integer omgezet naar een long door de integer in EAX met het tekenbit uit te breiden in EDX. Op 64-bit moet EAX naar RAX uitgebreid worden. Verder moet een 0 gepushed worden om de stack correct te maken.

**i2d** Hier moet alleen de stack in orde gezet worden door een extra 0 te pushen.

**l2i** Op 32-bit moet hier gewoon het bovenste deel van de stack verwijderd worden. Op 64-bit moet het stackslot met de nul verwijderd worden en het stackslot moet een 32-bit waarde krijgen. Dit kan door een 32-bit MOV uit te voeren op het register waar die waarde zich in bevindt. De bovenste 32-bit worden dan 0 gemaakt.

**f2d** Analoog i2d

**lcmp** Bij afwezigheid van een instructie om 64-bit waarden te vergelijken diende op 32-bit het vergelijken van twee longs in verschillende stappen te gebeuren. In 64-bit mode kan dit rechtstreeks gebeuren door de twee waarden van elkaar af te trekken en vervolgens, afhankelijk van het resultaat hiervan, de juiste stappen te ondernemen. Het

is wel belangrijk na te kijken, of de vlaggen die getest worden bij het vergelijken, wel aangepast werden, door - in dit geval SUB - de gebruikte operatie. Aangezien na de SUB verschillende mogelijkheden tot springen worden gebruikt, is het ook belangrijk te weten dat Jcc de vlaggen achteraf niet veranderd.

### Gecontroleerde hulpfuncties

In VM_Compiler worden ook verscheidene hulpfuncties gebruikt. Aangezien alles nagekeken dient te worden waar assembler in wordt gebruikt, dienen ook deze functies gecontroleerd te worden.

**genEpilogue** Deze hulpfunctie zorgt voor de overgang van een opgeroepen functie naar de functie waarin genEpilogue werd opgeroepen. Omdat hier voornamelijk met adressen wordt gewerkt moet alles hier naar 64-bit gehaald worden.

**emit_unresolved_getstatic** Geen aanpassingen

**emit_resolved_getstatic** Analoog emit_unresolved_getstatic

**emit_unresolved_putstatic** Analoog emit_unresolved_getstatic

**emit_resolved_putstatic** Analoog emit_unresolved_getstatic

**emit_unresolved_getfield** Getfield gebruikt een werkwijze gelijkaardig aan getstatic of putstatic. Alleen wordt hier eerst een MOV en dan een PUSH gedaan. Deze MOV moet voor de references en de 64-bit types een 64-bit MOV zijn. Hier werd de code voor aangepast.

**emit_resolved_getfield** Analoog emit_unresolved_getfield

**emit_unresolved_putfield** Analoog emit_unresolved_getfield

**emit_resolved_putfield** Analoog emit_unresolved_getfield

**emit_unresolved_invokevirtual** Analoog emit_unresolved_getfield

**emit_resolved_invokevirtual** Analoog emit_unresolved_getfield

**emit_unresolved_invokespecial** Analoog emit_unresolved_getfield

**emit_resolved_invokespecial** Analoog emit_unresolved_getfield

**emit_unresolved_invokestatic** Analoog emit_unresolved_getstatic

**emit_resolved_invokestatic** Analoog emit_unresolved_getstatic

**emit_invokeinterface** Voor het eerste werd hier ook de constante LG_WORDSIZE gebruikt, althans in letterlijke vorm, namelijk 2. Op 64-bit is dit 3. De constante werd vervangen door haar naam.

**emit_resolved_new** Geen aanpassingen

**emit_unresolved_new** Geen aanpassingen

**emit_resolved_newarray** Geen aanpassingen

**emit_unresolved_newarray** Geen aanpassingen

**emit_checkcast_resolvedClass** Geen aanpassingen

**emit_checkcast_final** Geen aanpassingen

**emit_instanceof_resolvedClass** Geen aanpassingen

**emit_instanceof_final** Geen aanpassingen

**genPrologue** Naast gewone aanpassingen voor het correct gebruik van adressen duikt hier nog een nieuwe probleem op: FPU_STACK_SIZE. Dit probleem wordt verder uitgewerkt in paragraaf 7.6. Vermoedelijk zal de code hier wel werken, maar dit moet toch zeker deftig nagekeken worden.

**emit_deferred_prologue** Aanpassingen voor het correct gebruik van adressen.

**genMonitorEnter** Aanpassingen voor het correct gebruik van adressen.

**genMonitorExit** Aanpassingen voor het correct gebruik van adressen.

**genBoundsCheck** Aanpassingen voor het correct gebruik van adressen.

**genCondbranch** Geen aanpassingen.

**genParameterRegisterLoad()** Geen Aanpassingen

**genParameterRegisterLoad (int params)** Stack slots zijn nu 64-bit. Om deze correct te verplaatsen moeten dezelfde aanpassingen als voor adressen gemaakt worden.

**genParameterRegisterLoad(VM_MethodReference method,bool hasThisParam)** Een probleem hier is dat ervanuit wordt gegaan dat er maar 2 vrije GPRS zijn. Op 64-bit zijn dit er echter nog eens 8 meer. De methode werd uitgebreid om te werken met de extra registers. Een ander probleem is dat een referentie gelijk wordt gezien aan een integer, hier moet dit lichtjes aangepast worden. Een extra if is nodig om referenties van integer-achtigen (ook bytes, shorts en chars worden als integer behandeld) te onderscheiden.

**genParameterCopy** Analoog vorige item.

### A.2.2 Magic

Het laatste stuk van VM_Compiler bevat de ondersteuning voor Magic. Deze wordt opgevangen door een serie van if's:

```
if (methodName == VM_MagicNames.wordToLong){
  //do someting magical
}
```

Het is vaak moeilijk om na te gaan wat deze magic's nu precies moeten doen, vooral omdat moet afgegaan worden op de naam en er nergens een beschrijving beschikbaar is.

Een eerste reeks magic's die aangepast diende te worden was deze die omzetting deed van int of long naar word. Op een 32-bit platform is een WORD 32-bit en op een 64-bit platform 64-bit. Met bepaalde instructies kan hier een probleem opduiken. De instructies van dit type worden hieronder besproken.

**wordFromInt** Het omzetten van een Int naar een Word. Aangezien zowel een Word als een Int in één stackslot wordt opgeslagen, dient, net zoals op een 32-bit platform, hier niets gedaan te worden.

**wordFromIntZeroExtend** Als ervan uit kan gegaan worden dat een Int altijd het resultaat is van een bewerking op 32-bit registers, dient hier niets gedaan te worden. In 64-bit mode worden 32-bit resultaten automatisch nul-uitgebreid opgeslagen in een 64-bit register. Dit is dus geen probleem.

**wordFromIntSignExtend** Zoals vermeld in [1] moet men, alvorens men een 32-bit register 64-bit teken-uitgebreid kan gebruiken, het expliciet naar 64-bit uitbreiden. Hiervoor dient de instructie CDQE.

**wordFromObject** Zowel de referentie naar een Object als een Word hebben dezelfde semantiek. Hier moet dus niets gedaan worden.

**wordToInt** Omdat 32-bit registers ZeroExtended opgeslagen worden in 64-bit registers, moeten de meest significante 32-bit van het Word op 0 geplaatst worden. Dit kan door reg = Reg & 0x00000000FFFFFFFF uit te voeren.

**wordToAddress** Word en Address zijn dezelfde types. Hier moet dus niets gedaan worden.

**wordToOffset** Idem wordToAddress.

**wordToObject** Idem wordToAddress.

**wordToObjectReference** Idem wordToAddress.

**wordToExtent** Idem wordToAddress.

**wordToWord** Triviaal.

**wordFromLong** Hier dient het lege stackslot dat altijd eerst wordt gepushed verwijderd te worden. Op 32-bit werd hier een fout ontdekt. Deze operatie werd als een operatie gedefinieerd die niets moest doen. Ook hier moest echter het eerste stackslot van de long verwijderd worden.

**wordToLong** Hier dient een extra leeg stackslot toegevoegd te worden. Dus eerst wordt een nul gepushed, dan het oorspronkelijke word.

Een volgende groep van magic die aangepast dient te worden is deze die shift's op word's uitvoert. Deze worden op 32-bit-registers uitgevoerd, en op 64-bit registers moeten dus de 64-bit shift-methodes gebruikt worden. Dit kon eenvoudig gebeuren door de in VM_Assembler uitgebreide methoden te gebruiken. Gewoon *_Quad* achter de juiste instructie voegen is voldoende. Volgende magic's werden aangepast:

- wordLsh
- wordRshl
- wordRsha

# Bijlage B

# Inhoud CD

**JikesRVM-x86-64_64bit.tar.gz** De aangepaste versie van Jikes RVM voor x86-64 in 64-bit mode

**JikesRVM-x86-64_32bit.tar.gz** De aangepaste versie van Jikes RVM voor x86-64 in 32-bit mode

**JikesRVM-x86-64_64bit.diff** Een patch om de originele broncode om te zetten naar deze voor Jikes RVM voor x86-64 in 64-bit mode

**JikesRVM-x86-64_32bit.diff** Een patch om de originele broncode om te zetten naar deze voor Jikes RVM voor x86-64 in 32-bit mode

**VM_Examples** Een map met een aantal versies van VM.java. Deze hebben telkens een naam `VM_<test>.java`. Door VM.java te vervangen door een van deze bestanden kunnen een aantal demonstraties gegeven worden van de 64-bit versie.

**Java_Examples** Een map met een aantal Java bestanden. Deze hebben telkens een naam `test<test>.java`. Deze testbestanden kunnen gebruikt worden om aan te tonen dat de werking van de VM.java versies met gelijke `<test>` correct is. Ze bevatten namelijk dezelfde code (op de printfunctie na).

**JikesRVM_x86-64.pdf** Dit document

**x86-64-pc-linux-gnu-64** Het configuratiebestand voor Jikes RVM voor x86-64 in 64-bit mode

**x86-64-pc-linux-gnu-32** Het configuratiebestand voor Jikes RVM voor x86-64 in 32-bit mode

# Bibliografie


[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume I: Application Programming*, 3.09 edition, September 2003.

[2] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume II: System Programming*, 3.09 edition, September 2003.

[3] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume III: General Purpose and System Instructions*, 3.09 edition, September 2003.

[4] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume IV: 128-Bit Media Instructions*, 3.05 edition, September 2003.

[5] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume V: 64-Bit Media and x87 Floating Point Instructions*, 3.05 edition, September 2003.

[6] Intel Corporation. *Intel Extended Memory 64 Technology Software Developer's Guide Volume 1 of 2*, 1.1 edition.

[7] Intel Corporation. *Intel Extended Memory 64 Technology Software Developer's Guide Volume 2 of 2*, 1.1 edition.

[8] Christof Windeck Axel Vahldiek. Overstappen a.u.b.:de 64-bit pc-techniek gaat nu pas echt van start. *c't magazine voor computer techniek*, (5):100–103, Mei 2005.

[9] Christof Windeck. Quo vadis?: De 64-bit plannen van amd en intel. *c't magazine voor computer techniek*, (5):116–120, Mei 2005.

[10] *System V Application Binary Interface AMD64 Architecture Supplement*, 0.95 edition, January.

[11] Benjamin Benz. Motor op maat: Processors van amd en intel voor elke toepassing. *c't magazine voor computer techniek*, (5):110–115, Mei 2005.

[12] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition.* Addison Wesley, 2nd edition, 1999.

[13] Kris Venstermans and Koen De Bosschere. Jvm spec favours 32-bit platforms. 2005.

[14] *The Jikes Research Virtual Machine User's Guide*, 2.3.4 edition, February 2005.

[15] S.M. Blackburn M. Butrico A. Cocchi P Cheng J. Dolby S. Fink D. Grove M. Hind K.S. McKinley M. Mergen J.E.B. Moss T. Ngo V. Sarkar B. Alpern, S. Augart and M. Trapp. The jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.