Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Eine Implementation des Python Compilers aus dem Buch Essentials of Compilation für Risc-V

Selina Koch

Bachelorarbeit

Beginn der Arbeit:	31.01.2023
Abgabe der Arbeit:	01.05.2023
Gutachter:	Dr. J. Witulski
	Dr. C. Bolz-Tereick

Ehrenwörtliche Erklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 30. April 2023

Selina Koch

Zusammenfassung

Das Buch *Essentials of Compilation* von Jeremy Siek [Sie23] ist ein gutes praktisches Lehrbuch für die Implementation eines Compiler für die x86 Architektur. Ein solches Pendant für die RISC-V Architektur gibt es hingegen nicht. Da RISC-V in seiner Bedeutung und Beliebtheit stark wächst ist es interessant zu sehen ob das oben genannte Lehrbuch auch für diese Architektur geeignet ist. Daher werden in dieser Arbeit die erste Kapitel des Buches für RISC-V als Ziel-Architektur implementiert. Damit wird geprüft ob sich die Inhalte des Buches und der zugehörige Support-Code ohne großartige Änderungen auch auf andere Architekturen übertragen lassen. Diese Arbeit legt nahe, dass sich die Theorie des Buches gut auf die RISC-V Architektur übertragen lässt. Der vorgegebene Support-Code hingegen ist sehr nah an die x86 Architektur gebunden und benötigt daher grundlegender Anpassungen um für RISC-V geeignet zu sein.

Danksagung

Ich danke meinem Freund Jonas Braß für seine Emotionale Unterstützung und Geduld mit mir während der Erstellung dieser Arbeit.
Außerdem danke ich meinen Betreuern Carl Friedrich Bolz-Tereick und John Witulski für ihre Unterstützung mittels fachlicher Kompetenz und dem motivieren in frustrierenden Zeiten. Und natürlich danke ich auch Jeremy Siek für die Bereitstellung der x86 Musterlösung und der Vorabversion des Buches Essentials of Compilation.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **2**

2.1 Grundlagen der x86- und RISC-V-Architekturen 2

2.2 Calling Conventions und Stack Frames der RISC-V Architektur 3

2.3 Einführung in den Python-Compiler 6

2.4 Teststrategie 9

2.5 Testumgebung für RISC-V 10

3 Implementierung des Python-Compilers **11**

3.1 Integer und Variablen 11

3.2 Register Allocation 14

3.3 Booleans und Bedingungen 16

3.4 Schleifen 19

3.5 Funktionen 20

3.6 Interpreter Anpassungen 21

4 Evaluation **23**

4.1 Testen der implementierten Funktionalitäten 23

4.2 Testen der Leistung des Python-Compilers 24

4.3 Bewertung der Ergebnisse 25

5 Verwandte Arbeiten **27**

6 Zusammenfassung und Ausblick **28**

Abbildungsverzeichnis **29**

Tabellenverzeichnis **29**

Literatur **31**

1 Einleitung

Das Themengebiet Compilerbau aus der Informatik beschäftigt sich mit der Implementierung und Optimierung von Compilern. Ein Compiler ist ein Programm, welches Quellcode einer höheren Programmiersprache in eine ausführbare Maschinensprache übersetzt, damit es auf einem Computer ausgeführt werden kann. Der Übersetzungsprozess beinhaltet normalerweise mehrere Phasen, wie die Analyse des Quellcodes, die Optimierung des generierten Codes und die Generierung des Maschinencodes. Compilerbau ist ein anspruchsvolles Feld der Informatik, das hohe Einstiegshürden aufweist.
Das Buch *Essentials of Compilation* von Jeremy Siek [Sie23] versucht Bachelor Studenten dieses Thema nahe zu bringen und ihnen den Einstieg zu erleichtern. Dabei bedient es sich der modernen Programmiersprache Python und einem praktischen Ansatz. Den Studenten wird mit dem Buch die Theorie eines Compilers vermittelt. Mittels Aufgaben im Buch und unter Zuhilfenahme des zur Verfügung gestellten Support-Codes sollen die Studenten innerhalb eines Semesters ihren eigenen Compiler schreiben und Stück für Stück verbessern. Das Buch hat x86 als Ziel-Architektur, allerdings nutzen viele moderne Systeme nutzen heutzutage eher RISC-Architekturen, wie zum Beispiel ARM oder RISC-V. Daher wollen wir betrachten ob das Buch auch mit anderen Ziel-Architekturen, hier im Hinblick auf RISC-Architekturen, umsetzbar ist.
Da RISC-V als open source Befehlssatzarchitektur (Instruktion set architecture, ISA) zur Verfügung steht, wählen wir in dieser Arbeit dieses als Ziel-Architektur. Bei der Umsetzung wird allerdings nicht der Ausgangspunkt der Studenten eingenommen, welche das Lehrbuch und den Support-Code [Sie20] zur Verfügung haben, sondern zusätzlich wird mit der x86 Musterlösung gestartet, welche ebenfalls von Jeremy Siek geschrieben wurde. Um den entstehenden Code zu testen wird der C basierende Support Code durch die RISC-V toolchain cross-compiliert. Der durch den Compiler generierte RISC-V Assemblercode wird in einer durch den Emulator QEMU zu Verfügung stehenden Umgebung ausgeführt und getestet, da so zur Entwicklung des Compilers ein x86 basierender Laptop verwendet werden kann.
Am Ende der Arbeit messen wir die Güte des entstandenen Compiler einerseits anhand der Tests die im Support-Code zur Verfügung stehen und Vergleichen seine Laufzeiten mit denen des nativen Python Interpreters. Außerdem wird am Ende der Arbeit beurteilt ob sich das Lehrbuch [Sie23] auch für andere Architekturen, speziell RISC-Architekturen eignet und in wieweit der zur Verfügung stehende Support Code diese Änderung der Zielsprache unterstützt oder behindert.

2 Grundlagen

2.1 Grundlagen der x86- und RISC-V-Architekturen

Da wir einen Compiler, der auf x86 compiliert, umbauen wollen zu einem Compiler, der zu RISC-V compiliert, betrachten wir zuerst die konzeptionellen Unterschiede beider Architekturen.
Die x86 Architektur ist eine Computerarchitektur die erstmals 1978 mit den Prozessoren 8086 und 8088 von Intel eingeführt wurde [[aut23] S.2-1] und bis heute, in weiter entwickelter Form, viel Anwendung findet. Sie gehört zu der Klasse der CISC-Architekturen (Complex Instruction Set Computer), die im Gegensatz zu RISC-Architekturen (Reduced Instruction Set Computer) eine große Anzahl von komplexen Befehlen unterstützt. Der durch die x86-Architektur definierte Befehlssatz besteht aus einer großen Anzahl von Befehlen, welche alles von einfachen mathematischen Operationen bis hin zu komplexen Anweisungen zur Steuerung von Ein-/Ausgabe umfassen. Auch die Registerorganisation, die von der CPU zur Verarbeitung von Daten verwendet wird, wird durch die Architektur festgelegt. Sie unterstützt eine große Anzahl von Registern, einschließlich Allzweckregistern, Segmentregistern und Flags-Registern, die für die Steuerung des Prozessorverhaltens verwendet werden [[Jam95] S.13].
Die RISC-V-Architektur hingegen gehört, wie der Name bereits besagt, zu den RISC-Architekturen und stellt somit einen komprimierten Befehlssatz zur Verfügung. Eine RISC-V Instruktion besteht in den meisten Fällen aus einem Drei-Adressen-Code, also einem Namen und drei Operanden, z.B. *add t0, t1, x0* [[ULSA08] S.440]. Der Name gibt an um welche Instruktion es sich handelt. Falls der Name auf ein *i* endet gibt dieses an, dass die Instruktion als letzten Operanden einen Immediate, also eine ganze Zahl, erwartet. Angewendet wird die Instruktion auf den vorletzten und letzten Operanden, hier also *t1 + x0*. Das Ergebnis wird im ersten Operanden *t0* gespeichert. Es gibt auch Außnahmen dieser Regel, z.B. Instruktionen zum Laden und Speichern von Werten aus und in den Speicher haben lediglich zwei Operanden.
In dieser Arbeit wird die 64-Bit Variante von RISC-V verwendet, da diese am ehesten Kompatibel ist zu der im Buch verwendeten 64-Bit Variante von x86. In diese Version von RISC-V sind alle Register 64-Bit groß und der Speicher wird mit 64-Bit Adressbreite adressiert.
Ein Vorteil der generellen RISC Architektur ist es, dass man sich einfacher in diese einlesen und mit ihr arbeiten kann. Die Instruktionen sind alle sehr ähnlich und einfach aufgebaut und es gibt wenige Funktionalitäten die in mehreren Instruktionen vorkommen [[Jam95] S.14]. Während bei RISC Architekturen einzelne Funktionalitäten (wie zum Beispiel das Inkrementieren[1]) durch möglichst wenig Instruktionen implementiert werden, gibt es bei CISC Architekturen verschiedene Instruktionen die dies realisieren, da alle einen etwas anderen Zweck erfüllen, um die Anzahl an benötigten Instruktionen pro Programm gering zu halten. So lässt sich das Inkrementieren einer Zahl in x86 durch die

[1] erhöhen einer Zahl um 1

Instruktionen `add`, `adc`, `inc`, `xadd` und `adx` realisieren. In RISC-V hingegen ginge dies nur mittels der Instruktion *addi*.[2]
Diese Reduziertheit an Befehlen kann auch zu einer höheren Leistung und Energieeffizienz führen, da RISC-Architekturen versuchen die Taktzyklen pro Instruktion zu minimieren, häufig auf Kosten der Anzahl der benötigten Instruktionen pro Programm. CISC hingegen bedient sich dem genau gegenteiligen Ansatz, die Anzahl an Instruktionen pro Programm zu minimieren, auf Kosten der Anzahl an Taktzyklen pro Instruktion [CCS00].
Wir haben nun also gesehen, dass RISC und CISC unterschiedliche Ansätze bei der Umsetzung von Programmen haben, welche Einfluss nehmen auf den Compiler, den wir zu verstehen und zu verändern versuchen.

2.2 Calling Conventions und Stack Frames der RISC-V Architektur

Da sich die grundlegende Struktur des Assemblercodes zwischen x86 und RISC-V ändert, ist es auch interessant sich anzusehen wie die Interaktion von Funktionen und Programmen der Quellsprache im Assemblercode realisiert werden. Daher sehen wir uns im Folgenden die Calling Conventions von RISC-V an, welche wir in den Kapiteln Register Allocation und Funktionen implementieren werden.
Die RISC-V Calling Convention besagt, dass der Stackpointer abwärts wächst, auf die erste freie Stelle im Stack zeigt und immer 16-Bytes bündig ist [[WLPA14] S.109]. Da der Stack abwärts wächst werden auf dem Stack gespeicherte Werte über positive Offsets des Stackpointers erreicht. Allerdings unterstützen RISC-V Instruktionen keine automatische Verwaltung des Stacks. Es existiert keine `push`, `pop`, `call` oder `return` Instruktionen. RISC-V kann aber durchaus auch auf den Stack schreiben. So kann *push wert* über die speicher-Instruktion *sd wert, offset[sp]*[3] erreicht werden. *offset[sp]* steht hierbei dafür, dass auf den Speicher an der Adresse von *sp* plus den Wert von *offset* zugegriffen wird. Das Verwalten des Offsets ist eine Aufgabe die der Compiler im Vergleich zu x86 zusätzlich behandeln muss. *pop register* kann äquivalent über die Lade-Instruktion *ld register, offset[sp]*[4] erreicht werden. `return` wird über ein *jr ra*[5] erreicht, hierbei wird einfach zu der im Register *ra* gespeicherten Adresse gesprungen. Vorher muss allerdings der Stack aufgeräumt werden, da dies nicht in *jr* enthalten ist. Dies ist Aufgabe des Callees, also der aufgerufenen Funktion. Ihr Gegenstück ist der Caller, die Funktion, welche die aktuelle Funktion aufgerufen hat. Ebenfalls muss der Callee den Stackpointer wieder auf seinen vorherigen Wert zurücksetzen, da dieses Register Callee-Save ist. Die `call` Funktion ist in RISC-V nicht nativ implementiert, wird aber von vielen Assemblern als Pseudo-Instruktion unterstützt.
Bei Aufruf einer Funktion mit Argumenten werden diese in den Registern *a0* bis *a7* übergeben [[III18] Kapitel 5]. Sollte es mehr als acht Argumente geben werden diese auf dem

[2] addi = add Immediate
[3] sd = store double word
[4] ld = load double word
[5] jr = jump register

Stack übergeben. Bei Objekten die größer als 64-Bit sind werden Zeiger auf diese Objekte als Argumente übergeben. Die Rückgabewerte der Funktion liegen dann im Anschluss in den Registern *a0* und *a1*.
In der Tabelle 1 ist zu sehen welche RISC-V Register mit welchen x86 Registern vergleichbar sind und wer bei einem Funktionsaufruf jeweils zuständig ist diese Register zu speichern.

Register	Name	Beschreibung	Saver	x86-Äquivalent	Saver
x0	zero	ist immer 0			
x1	ra	Rücksprungadresse	Caller		
x2	sp	Stack-Pointer	Callee	rsp	Caller
x3	gp	Global-Pointer			
x4	tp	Thread-Pointer			
x5-x7	t0-t2	Temporaries	Caller		
x8	s0/fp	Frame-Pointer	Callee	rbp	Callee
x9	s1	Saved Register	Callee	rbx	Callee
x10	a0	1. Funktionsargument/Rückgabewert	Caller	rdi/rax	Caller
x11	a1	2.Funktionsargument/Rückgabewert	Caller	rsi/-	Caller
x12-x17	a2-a7	Funktionsargumente	Caller	rdx, rcx, r8, r9	Caller
x18-x27	s2-s11	Saved Registers	Callee	r12-r15	Callee
x28-x31	t3-t6	Temporaries	Caller	r10,r11	Caller

Tabelle 1: RISC-V Register und ihre x86 Äquivalente in diesem Compiler inklusive der zugehörigen Savern [[WLPA14] S.137, [III18] S.147].

Es folgt ein Beispiel wie sich der Stack und die Register bei einem Beispielprogramm verändern:
Das Programm in Abb. 1 ruft in der *main*-Funktion die Funktion *add* mit den Werten 40 und 2 auf und gibt das Ergebnis anschließend mittels *print_int* auf der Konsole aus.

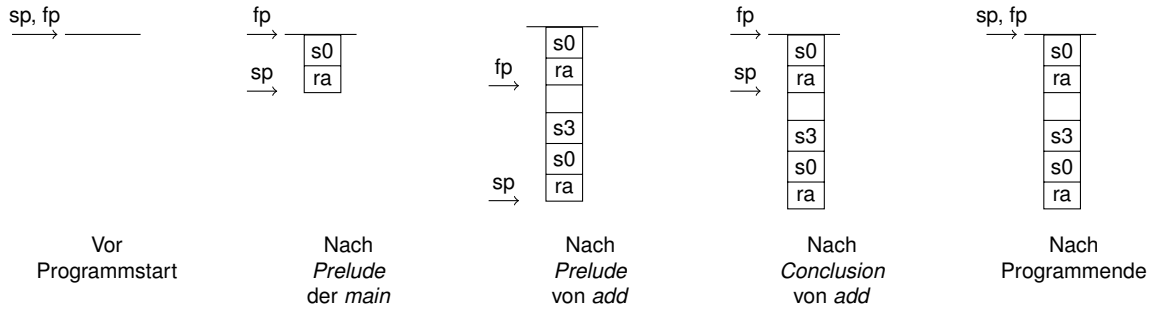

Tabelle 2: Der Verlauf des Stack für das in Abb. 1 abgebildete Programm.

Der Stack verändert sich dabei nach Start des Programms wie in Tabelle 2 zu sehen. In den ersten vier Zeilen der *main* werden die Werte für *ra* und *s0* auf den Stack gelegt und sowohl der Stackpointer als auch der Framepointer angepasst. Diesen Part des Programms nennen wir *prelude* [[Wil19]]. In den nächsten fünf Zeilen bereiten wir den Sprung in die Funktion *add* vor, indem wir die Adresse der Funktion in *t1* und die Imme-

```
main:
prelude:
    addi sp, sp, -16
    sd ra, 8(sp)
    sd s0, 16(sp)
    addi s0, sp, 16
    la t1, add
    li t0, 40
    mv a0, t0
    li t0, 2
    mv a1, t0
    jalr ra, t1, 0
    call print_int
conclusion:
    ld s0, 16(sp)
    ld ra, 8(sp)
    addi sp, sp, 16
    li a5, 0
    mv a0, a5
    jr ra
```

```
add:
prelude:
    addi sp, sp, -32
    sd ra, 8(sp)
    sd s0, 16(sp)
    sd s3, 24(sp)
    addi s0, sp, 32
    mv s3, a0
    mv a5, a1
    add a0, s3, a5
conclusion:
    ld s3, 24(sp)
    ld s0, 16(sp)
    ld ra, 8(sp)
    addi sp, sp, 32
    jr ra
```

Abbildung 1: RISC-V Assembler Code für den Aufruf einer einfachen Addier-Funktion aus der *main*.

diates 40 und 2 in die Argument-Register *a0* und *a1* laden[6,7,8] . In Zeile zehn springen wir nun in die Funktion *add*.[9]
Hier werden erneut in den ersten fünf Zeilen die Werte von *s3*, *s0* und *ra* auf dem Stack gespeichert. *s3* wird hier gespeichert, da wir dieses Register innerhalb der Funktion verwenden werden und der Callee bei Benutzung für die Sicherung zuständig ist. Da wir drei Werte auf dem Stack speichern, welche jeweils einen Platz von 8-Byte einnehmen, der Stackpointer aber ein vielfaches von 16-Byte sein muss entsteht hier eine Lücke auf dem Stack.
In den nächsten drei Zeilen berechnet die Funktion die Summe beider Eingaben und speichert das Ergebnis im Rückgabe-Register *a0*. Nun folgt die *conclusion*. Hier werden die ursprünglichen Werte von *s3*, *s0*, *ra* und *sp* wieder hergestellt und zur in *ra* gespeicherten Rücksprungadresse gesprungen.
Wir befinden uns nun zurück in Zeile elf in der *main*. Hier wird nun das Ergebnis mittels der Funktion *print_int* ausgegeben und im Anschluss in der *conclusion* alle Werte von *sp*,

[6] la=load address
[7] li=load Immediate
[8] mv=move
[9] jalr=jump and link register

ra und *s0* wieder auf die anfänglichen Werte gesetzt und das Programm beendet.

2.3 Einführung in den Python-Compiler

Da wir uns nun grundlegend die wichtigsten Unterschiede zwischen den beiden Architekturen angesehen haben, können wir uns im Folgenden mit dem vorhandenen Compiler und Interpreter, die in [Sie23] gebaut wurden, beschäftigen.
Der Compiler besteht grundlegend aus einem Front- und einem Backend. Das Frontend ist dafür zuständig den eingegebenen Pythoncode zu analysieren und aus ihm einen abstrakten Syntaxbaum (AST) zu erstellen. Diesen Syntaxbaum nimmt das Backend als Ausgangspunkt und erstellt daraus in verschiedenen Passes den Assemblercode.
Das Frontend wird im Buch in Kapitel 3 *Parsing* erklärt. Dieses Kapitel ist allerdings keine Voraussetzung für die folgenden Kapitel, da der dort entwickelte Parser im Compiler auch durch den Python-Parser und die Python-AST-Klassen ersetzt werden kann. Der Fokus des Buches liegt auf der Entwicklung und Erklärung des Backends. Daher wird auch nur das Backend in dieser Arbeit behandelt und das Frontend so belassen wie es in der Musterlösung realisiert wurde.
Das Backend hingegen wird so verändert, dass der resultierende Assemblercode von einer RISC-V Architektur ausgeführt werden kann.
Da [Sie23] als angewandtes Lehrbuch konzipiert ist wird der Compiler dort in verschiedenen Stufen gebaut. Jedes Buchkapitel stellt eine Stufe dar und erstellt einen funktionsfähigen Compiler. Was sich ändert ist die Komplexität und Größe der Teilsprache die diese Compiler in einzelnen Stufen compilieren können. Jeder Compiler baut auf die vorherigen auf. Um das Verständnis für die einzelnen Vorgänge zu vereinfachen wurden die einzelnen zu bewältigenden Aufgaben des Backends in sogenannte *Passes* unterteilt. Ein Pass der in einer vorherigen Stufe eingeführt wurde wird in den weiteren Stufen immer mit durchlaufen, aber teilweise nicht verändert. In Tabelle 3 ist dargestellt welche Passes von den in dieser Arbeit behandelten Compilern verändert werden.

Es folgt eine Übersicht über die einzelnen Passes, die durchlaufen werden, um aus dem AST eines Programms einen ausführbaren RISC-V Assemblercode zu generieren. Die Namen und Funktionen der einzelnen Passes sind in [Sie23] ausführlich behandelt. Hier wird lediglich ein Überblick über die einzelnen Passes gegeben, damit ihre Funktion in den folgenden Kapiteln der Arbeit bekannt sind.
Für die erste Stufe des Compilers stellt der *Remove Complex Operands* Pass den Anfang dar. Dieser Pass generiert aus dem vorhandenen AST einen 3-Adressen-Code.
In den ersten Stufen des Compilers folgt hierauf der Pass *Select Instructions*. Dieser formt den generierten Drei-Adressen-Code zu RISC-V Instruktionen um. Hierbei werden den genutzten Variablen und Immediates allerdings noch keine Speicherplätze zugewiesen. Dies erfolgt im nächsten Pass *Assign Homes*. In der ersten Stufe werden hier alle Variablen auf dem Stack gespeichert. In den folgenden Stufen werden den Variablen Register zugewiesen und nur im Falle von Spills, mehr zu speichernde Werte als freie Register, Variablen auf dem Stack gespeichert.

Passes \ Stufen	Integer und Variablen	Register Allocation	Booleans und Bedingungen	Schleifen	Tuple	Funktionen
Type-based Resolution						✓
Shrink			✓	✓	✓	✓
Reveal Functions						✓
Limit Functions						✓
Expose Allocation					✓	✓
Remove Complex Operands	✓		✓	✓	✓	✓
Explicate Control			✓	✓	✓	✓
Select Instructions	✓		✓		✓	✓
Uncover Live		✓	✓	✓	✓	✓
Build Interference		✓	✓		✓	✓
Allocate Registers		✓	✓		✓	
Assign Homes	✓	✓			✓	✓
Patch Instructions	✓	✓	✓		✓	✓
Prelude and Conclusion	✓	✓	✓		✓	✓

Tabelle 3: Die Passes werden in dieser Reihenfolge in den einzelnen Stufen des Compilers abgearbeitet. Markiert sind hier jeweils die Passes die in den einzelnen Stufen geändert oder hinzugefügt werden.

Der nächste Pass *Patch Instructions* behebt mögliche Fehler im nun vorhandenen Assemblercode und Optimiert diesen teilweise. Wenn Werte auf dem Stack referenziert werden, werden sie hier erst in das Register *t0* geladen, da RISC-V Operationen nicht direkt auf dem Stack agieren können. Außerdem werden *mv* Befehle die das gleiche Ziel wie Quelle haben ausgelassen.
Alle Stufen enden mit dem Pass *Prelude and Conclusion*. Hier werden die Callee-Save Register und die Return Adressen vor Start des Programms auf dem Stack gesichert und für den aktuellen Frame der *main*-Funktion der Stack- und Frame-Pointer angepasst. Am Ende des Programms werden die gesicherten Werte wieder hergestellt. In den ersten Stufen wird dieser Code nur am Anfang und Ende angefügt, ab der Stufe Funktionen geschieht dies für jede einzelne Funktion.
In der Stufe Register Allocation werden die Passes *Uncover Live*, *Build Interference* und *Allocate Registers* eingeführt. Innerhalb dieser drei Passes wird der Pass *Assign Homes* neu aufgesetzt und optimiert. Es wird zuerst in *Uncover Live* eine liveness Analyse durchgeführt, die für jede Instruktion berechnet welche Variablen vor und nach der Instruktion noch live sind, welche Werte also noch aktiv benötigt werden. Im Anschluss werden diese Befunde in *Build Interference* in einen Interferenzgraph gespeichert. Dieser besitzt für alle im Programm verwendeten Variablen einen Knoten. Die Kanten stellen dar ob zwei Variablen zum gleichen Zeitpunkt live sind, und somit nicht im selben Register gespeichert werden können. Dieser aufgebaute Graph kann nun im nächsten Pass *Allocate Registers* mittels eines Graph-Färbungs-Algorithmus eingefärbt werden.[10] Ziel dieses Algorithmus ist es jeden Knoten mit einer Farbe zu färben und dabei benachbarte Knoten mit unterschiedlichen Farben zu versehen. Außerdem ist es Ziel des Algorithmus

[10] [CAC+81] und [Cha82]

die insgesamte Anzahl der verwendeten Farben zu minimieren. Die Farben stellen in unserem Kontext die verschiedenen Speicherplätze dar. Dies bedeutet, dass zwei Variablen die gleichzeitig genutzt werden nicht am selben Ort gespeichert sind und insgesamt so wenig wie möglich verschiedene Speicherplätze genutzt werden. Denn wenn die Anzahl der Farben die Anzahl der Register überschreitet, werden die übrigen Variablen als Spills auf den Stack geschrieben, was zusätzliche Zeit kostet und daher vermieden werden soll. Anschließend werden die Variablen im Programm mit ihrem zugewiesenen Speicherplatz referenziert und dort abgespeichert.
Der Pass *Shrink* ersetzt ausgewählte Operationen im Code mit anderen, um die Anzahl der benutzten Operationen und somit den Aufwand in späteren Stufen zu verringern. In der Stufe Booleans und Bedingungen wird er erstmals verwendet und ersetzt dort *and* und *or* Operationen mit *True\False* Blöcken. In der Stufe Funktionen wird er verwendet um eine explizite *main*-Funktion einzuführen.
Der Pass *Explicate Control* wird ebenfalls mit der Stufe Booleans und Bedingungen eingeführt. In diesem Pass wird ab dieser Stufe zusätzliche die Sprache C_{if} benutzt und L_{if} in diese übersetzt. C_{if} ist ähnlich zu C und unterstützt die meisten Operatoren der Sprache L_{if}, aber die Argumente und Operatoren sind auf atomare Ausdrücke reduziert [[Sie23] S.22]. C_{if} unterstützt auch *if*-Ausdrücke, aber nur in beschränkter Form. Die Bedingung muss ein Vergleich sein und die beiden Branches dürfen nur *goto*-Ausdrücke beinhalten.
In der Stufe Funktionen kommt erstmals der Pass *Reveal Functions* zu Einsatz. Er sorgt dafür, dass Funktions- und Variablenamen unterschiedlich behandelt werden können, indem er alle Funktionesnamen von der Syntax Name(*name*) auf FunRef(*name*, n) setzt, wobei n die Arität der Funktion ist.
Der letzte Pass der in dieser Arbeit relevant ist, ist der Pass *Limit Functions*. Dieser sorgt dafür, dass Funktionen mit mehr als 6 Argumenten nicht vorkommen. Dies erleichtert uns die Implementation von Tail-Calls. Um zu gewährleisten, dass Funktionen dieser Vorgabe entsprechen, obwohl Funktionen in Python kein begrenzte Anzahl von Argumenten haben werden bei Funktionen mit mehr als 6 Argumenten alle überzähligen Argumente in ein Tupel gespeichert, welches das neue sechste Argument darstellt.
Neben dem Compiler wird in [Sie23] auch ein Interpreter gebaut. Dieser nutzt den vom Compiler erzeugten Zwischencode und überprüft diesen nach jedem Pass auf Korrektheit. Hierfür werden die im Zwischencode erzeugten Konstrukte zurück nach Python übersetzt und ihre Wirkung dort simuliert. Er wurde im Rahmen dieser Arbeit ebenfalls auf RISC-V angepasst, da er als Hilfsmittel dient Fehler in der Übersetzung bereits im Zwischencode zu finden. Ohne dieses Hilfsmittel könnte die Fehlerfindung nur im finalen Assemblercode über QEMU stattfinden. Hier würden die meisten Fehler als ein Segmentationfault auftreten, im Interpreter hingegen wird sowohl der genaue Ort, als auch eine Beschreibung des Fehlers gegeben.

2.4 Teststrategie

Da wir jetzt eine grobe Vorstellung davon haben wie der Compiler, der als Vorgabe genutzt wird, funktioniert, können wir uns damit beschäftigen, wie wir überprüfen, ob der Compiler richtig funktioniert. Wir betrachten im folgenden also die Teststrategie, mit welcher in [Sie20] gearbeitet wurde.
Für den umgebauten Compiler wurde die Teststrategie des originalen Compilers übernommen, da die Umgebung hierfür bereits existierte. Für alle behandelten Stufen, bis auf *Register Allocation*, gibt es separate Testdateien. Außerdem existiert ebenfalls für jede Stufe außer *Register Allocation* in dem Ordner *tests* einen Unterordner, in dem die für jede Stufe neuen Tests gespeichert sind. Zusätzlich gibt es noch einen Ordner *int64*, welcher Tests enthält, die die 64-Bit Kompatibilität des Compilers prüfen. Diese Tests sind auch ab der ersten Stufe implementiert. Ein Test besteht aus vier Dateien:

- der Python Datei die compiliert wird,
- eine Input-Datei in der steht welche Eingaben das Programm bekommt, sofern Eingaben erforderlich sind,
- eine Output-Datei in welcher steht was das compilierte und ausgeführte Assembler Programm ausgibt
- eine *golden*-Datei in welcher angegeben ist was die erwartete Ausgabe des Programms ist.

Die einzelnen Dateien eines Tests interagieren dabei wie in Abb. 2 zu sehen.

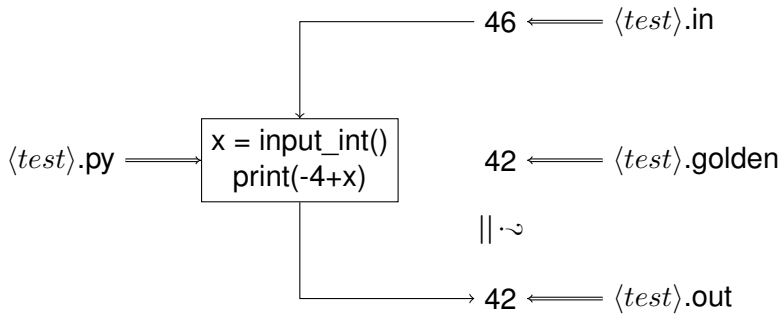

Abbildung 2: Beispiel für den Inhalt der Testdateien.
Die doppelten Pfeile zeigen dabei den Inhalt einer Datei an.
Die einfachen Pfeile zeigen den zeitlichen Ablauf des Tests an, beginnend bei $\langle test \rangle$.in.

Die Testprogramme laufen nun die angegebenen Tests ab, wobei sie Funktionen aus der *utils.py*-Datei verwenden. Das entsprechende Python-Programm wird zunächst für jeden Pass interpretiert und dessen Output mit dem Inhalt der *golden* Datei verglichen. Stimmen beide überein, so gilt der Pass als bestanden. Dies ist insofern interessant, als

dass nicht nur das finale Produkt, sondern auch alle Zwischensprachen getestet werden. Dadurch ist der Code einerseits stark an die Vorgaben gebunden, andererseits erleichtert dies die Fehlersuche, da so leicht der genaue Pass gefunden werden kann in welchem ein Fehler auftritt.
Nachdem alle Passes durchlaufen sind wird mit dem generierten Assemblercode eine ausführbare Assembler Datei erzeugt und deren Output mit dem Inhalt der *golden*-Datei verglichen. Ein Test ist erfolgreich wenn auch hier beide Inhalte übereinstimmen. So wird sowohl der Interpreter als auch der Compiler getestet.
Für die verschiedenen Stufen stehen hierbei unterschiedlich viele Tests zur Verfügung. Für die erste Stufe *Integer und Variablen* und die zweite *Register Allocation* gibt es 26 Tests in der Kategorie *var* und sieben unter *int64*. Für die dritte Stufe *Booleans und Bedingungen* kommen 43 Tests unter *if* hinzu. Für die vierte Stufe *Schleifen* sind es sieben Tests die hinzukommen und für *Funktionen* 14.

2.5 Testumgebung für RISC-V

Diese Tests wollen wir auch für unseren modifizierten Compiler verwenden. Allerdings können wir RISC-V Code nicht nativ ausführen und benötigen zum Testen des nun entstehenden Assemblercodes eine neue Testumgebung. Hierfür wurde eine Kombination aus der RISC-V toolchain und QEMU benutzt.
Die RISC-V toolchain ist ein Cross-Compiler, welcher von C und C++ auf RISC-V-Assembler übersetzt [[Dab14]]. Sie wurde genutzt um die runtime des Compilers zu übersetzen und um den vom Compiler generierten Code in ein Binary zu assemblieren und zu linken.
Als RISC-V Emulator wurde QEMU genutzt, welches ein generischer open source Maschinen Emulator und Virtualisierer ist [[Dev22]]. Hier werden die generierten Programme, inklusive der runtime, laufen gelassen und die Ergebnisse mit den erwarteten Ergebnissen der Python-Test-Programme verglichen.

3 Implementierung des Python-Compilers

Wir haben jetzt eine Testumgebung für den zu generierenden RISC-V Assemblercode zusammengestellt und ein Verständnis dafür wie der x86 Compiler beim Übersetzen eines Python Programms vorgeht. Nun können wir anfangen die einzelnen Stufen des Compilers auf RISC-V umzubauen. In diesem Kapitel geht es darum, wie genau die einzelnen Kapitel des Buches [Sie23] für RISC-V umgesetzt und dabei aufgetretene Probleme gelöst werden. Die Struktur und Titel in diesem Kapitels folgen daher der des Buches. Behandelt werden in dieser Arbeit *Kapitel 2-Integers and Variables*, *Kapitel 4-Register Allocation*, *Kapitel 5-Booleans and Conditionals*, *Kapitel 6-Loops and Dataflow Analysis* und *Kapitel 8-Functions*. *Kapitel 3-Parsing* wird ausgelassen, da es sich als einziges auf das Frontend bezieht und daher ein eigenständiges Thema darstellt. Ebenso wird *Kapitel 7-Tuples and Garbage Collection* ausgelassen. Jeremy Siek gibt in [Sie23] [S.xiii] an, dass die beiden Kapitel 7 und 8 als Ende eines Kurses zur Einführung in den Compilerbau geeignet sind. Daher wird sich entschieden die Arbeit mit Kapitel 7 abzuschließen, da hier der Unterschied zwischen x86 und RISC-V durch die verschiedenen Calling Conventions gut betrachtet werden kann.

3.1 Integer und Variablen

Das Buch steigt mit einer sehr komprimierten und daher einfachen Teilsprache von Python in den Bau des Compilers ein. Dies reduziert die Komplexität der ersten Stufe des Compilers sehr und bricht die zu durchlaufenden Passes auf ein Mindestmaß herunter. Dadurch ist es möglich den ersten Compiler in relativ kurzer Zeit fertig zu stellen und die ersten Python Programme zügig laufen zu lassen. Die Teilsprache $\mathcal{L}_{Var}$ Abb. 3 die in den ersten beiden Kapiteln des Buches behandelt wird umfasst nur die mathematischen Operationen Plus und Minus und lässt als Operanden nur Variablen und Integer zu.

$$
\begin{array}{llll}
\mathcal{L}_{Var} & exp & ::= & \text{int} \mid \text{input\_int()} \mid -exp \mid exp + exp \mid exp - exp \mid (exp) \\
 & stmt & ::= & \text{print}(exp) \mid exp \\
 & exp & ::= & var \\
 & stmt & ::= & var = exp \\
 & \mathcal{L}_{Var} & ::= & stmt^*
\end{array}
$$

Abbildung 3: konkrete Sprache $\mathcal{L}_{Var}$ frei nach [Sie23] S.14.

Programme die hier compiliert werden können bestehen also nur aus Zuweisungen wie in Abb. 4 zu sehen.

Der Compiler durchläuft hier die 4 Passes *Remove Complex Operands*, *Select Instructions*, *Assign Homes* und *Patch Instructions* [Tabelle 3].
Die Grundlegende Struktur des Compilers muss hier nicht verändert werden. Eine relevante Änderung gibt es allerdings bei der Behandlung von großen Immediates im Pass

```
x = input_int()
x = x + 3
y = x - 3
print(input_int() + (-x) + y)
```

Abbildung 4: Python Beispiel Code für die Sprache $\mathcal{L}_{Var}$.

Select Instructions.
In x86 wird nicht zwischen großen und kleinen Integern unterschieden. Hier kann jede ganzzahlige Zahl mit einem einzigen Befehl in ein Register, oder sogar den Stack geladen werden. Die ist bei RISC-V nicht der Fall.
In RISC-V werden Integer mit der Instruktion *li* in ein Register geladen.[11] Allerdings gilt dies nur für 32-bit Werte [[III18] S.58]. Da wir aber einen Compiler auf RISC-V-64 schreiben, wollen wir auch Werte im 64-bit Bereich benutzen. Um dies zu schaffen wird in dem Pass *Select Instructions* eine Funktion *shift* hinzugefügt, welche alle Integer behandelt die im 64-bit, aber nicht im 31-bit Bereich liegen [Abb. 5].

```
0     def shift(self, lhs, s):
1         result = []
2         tobin = format(to_unsigned(s.value), '064b')
3         zero = False
4         if not all([tobin[31] == t for t in tobin[:31]]):
5             result += [Instr('li', [lhs, Immediate(int(tobin[:31], 2))]),
6                        Instr('slli', [lhs, lhs, Immediate(0xb)])]
7         else:
8             zero = True
9         for pos in range(31, 64, 11):
10            if int(tobin[pos:pos+11], 2)!=0:
11                if zero:
12                    result += [Instr('li',
13                    [lhs, Immediate(int(tobin[pos:pos+11], 2))])]
14                else:
15                    result += [Instr('addi',
16                    [lhs, lhs, Immediate(int(tobin[pos:pos+11], 2))])]
17                zero = False
18                if pos<53:
19                    result += [Instr('slli', [lhs, lhs, Immediate(0xb)])]
20            else:
21                if pos<53 and not zero:
22                    result += [Instr('slli', [lhs, lhs, Immediate(0xb)])]
23        return result
```

Abbildung 5: Die Funktion *shift* aus *var.py*, welche den Umgang mit Immediates größer als 31-Bit regelt.

[11] li ist ein Pseudoinstruktion, die je nach größe des Integers in ein *addi* oder ein *lui+addi* übersetzt wird [lui = load upper Immediate].

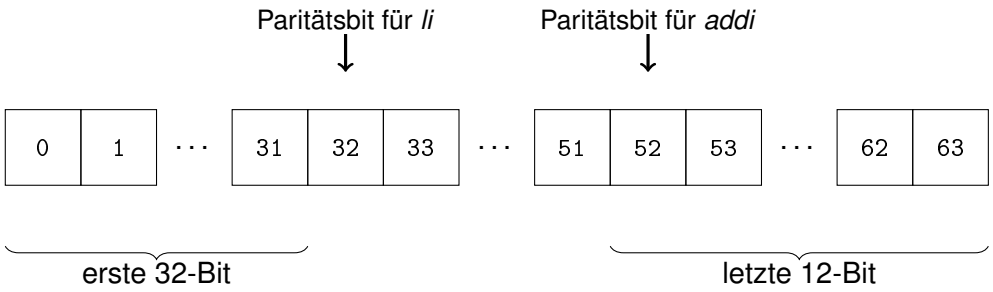

Abbildung 6: Darstellung eines 64-Bits Wertes in Binädarstellung mit Angabe der Paritätsbits für *li* und *addi*.

shift arbeitet mit den Instruktionen *li*,[12] *slli*[13] und *addi*.[14] Bei den Instruktionen *addi* und *li* ist zu beachten, dass diese die zu speichernden Immediates Vorzeichen erweitern (sign-extended), das bedeutet, dass *li* das 32-te und *addi* das 52-te Bit als Paritätsbit interpretieren und die ersten 32 beziehungsweise 52 Bits auf den selben Wert wie dieses Bit setzen [Abb. 6]. Im ersten Schritt der Funktion wird für den übergebene Immediate *s* in Zeile 2 seine 64-Bit Binädarstellung berechnet. Im Anschluss werden in Zeile 5 und 6 die ersten 31 Bits dieser Darstellung mittels *li* nach *lhs* geladen, sofern diese ersten 31 Bits nicht alle gleich sind. Sollten die ersten Bits alle gleich sein bedeutet dies, dass sie nur das Vorzeichen darstellen und ihr Wert null Beträgt. Im Anschluss werden die geladenen Bits um 11 Plätze nach links verschoben, um den fehlenden Bits Platz zu machen. Nun werden die folgenden 33 Bits in drei 11-er Paketen mittels den Instruktionen *slli* und *addi* ebenfalls nach *lhs* geladen. Hier werden immer 11 Bits verwendet, damit garantiert werden kann, dass das Paritätsbit für *addi* 0 ist, damit die addierte Zahl nicht als negativ angesehen und von der bisher geladenen Zahl subtrahiert wird, statt addiert. Am Ende gibt die Funktion die generierten Instruktionen zurück. Diese Funktion kann nun überall dort, wo Immediates auftauchen, aufgerufen werden um diese zu speichern.
Eine weitere Änderung betrifft den Pass *Patch Instructions* und die Speicherung von Variablen. Diese werden in dieser Stufe des Compilers ausschließlich auf dem Stack gespeichert. In x86 ist es möglich mit dort gespeicherten Werten direkt zu interagieren und diese Werte dort zu manipulieren. Dies ist mit RISC-V nicht möglich. Hier können Instruktionen nur auf Registern, in manchen Fällen auch auf Immediates, operieren. Daher müssen Werte erst vom Stack in Register gespeichert werden um diese zu verwenden. Dafür werden die Register *t0* und *t1* verwendet.
Zuletzt muss im Pass *Prelude and Conclusion* angepasst werden, welche Register, also der Stack- und Framepointer, gesichert werden. Der Stack wird also auf- und abgebaut wie in Calling Conventions und Stack Frames der RISC-V Architektur bereits beschrieben.

[12] li = load Immediate
[13] slli = shift left logical Immediate
[14] addi = add Immediate

3.2 Register Allocation

Wir haben in der ersten Stufe gemerkt, dass wir jegliche Werte mit welchen wir arbeiten wollen nur auf dem Stack speichern können. Dies benötigt Instruktionen und Laufzeit, welche wir uns sparen können, da es genau für diesen Zweck Register gibt. Um also nun Werte auch in Registern speichern zu können müssen wir uns überlegen welche Werte wir an welchem Ort wann speichern wollen. Dies ist die Aufgabe des vierten Buchkapitels *Register Allocation*.
In dieser Stufe ist die erste notwendige Änderung die referenzierten Register zu ändern. Anfangs sind unter *Caller_Save* die Register `rax,rcx, rdx, rsi, rdi, r8-r11` gespeichert. In der RISC-V Calling Convention entspricht dies den Registern *ra*, *t0-t6* und *a0-a7*. Dasselbe gilt für die *Callee_Save* Register. Hier werden `rsp, rbp,rbx, r12-r15` durch *sp*, *s0-s11* ersetzt. Die reservierten Register, in x86 `rax, r11, r15, rsp, rsp, rbp, flags` entsprechen im finalen Compiler den Registern *x0*, *ra*, *sp*, *gp*, *tp*, *s0*, *t0*, *t1*, *s1*, und *s2*. Sie werden für folgende Vorgänge benötigt und können daher nicht allgemein genutzt werden: *ra* wird ausgenommen, da hier die Rücksprungadresse gespeichert wird, welche nicht überschrieben werden sollte. *rax* da hier der Rückgabewert liegt, welcher erhalten bleiben soll. `r11, r15`, beziehungsweise *s1*, *s2*, speichern ab der Stufe *Tuple* Heap bezogene Zeiger. `rsp` beziehungsweise *sp* zeigen auf den aktuellen Stack und `rbp` beziehungsweise *s0*/*fp* auf den aktuellen Frame. `flags` wird wie der Name bereits suggeriert für Flags genutzt, die zum Beispiel bei der Instruktion `cmpq` entstehen. *gp* und *tp* werden bei internen Vorgängen der Architektur benötigt. *t0* und *t1* werden an vielen Stellen als Zwischenspeicher verwendet. Und zuletzt sind noch *a0-a7* für Funktions Argumente und Rückgabewerte reserviert.
generell_registers entspricht nun allen Registern die nicht reserviert sind. Im originalen Compiler werden diese übrigen Register in einer Liste aufgezählt und aus dieser und der Menge der reservierten Register die Menge aller Register berechnet. Dies wird hier geändert, da auf diese Weise Register bei der Implementierung des Compilers vergessen werden können. Stattdessen wird nun die Menge aller Register definiert und die Liste der generellen Register als Differenz aller Register mit der Menge der reservierten Register definiert. Diese Vorgehensweise erscheint intuitiver und weniger fehleranfällig, da nun auch im Nachhinein Register reserviert werden können, falls Sie für zukünftige Stufen des Compilers benötigt werden. Hierfür muss man nun nicht darauf achten jene Register aus den allgemeinen Registern aus- und in die reservierten einzutragen, sondern es genügt diese als reserviert zu vermerken.
Ebenfalls werden Änderungen an dem Dictionary *register_color* vorgenommen. Dieses bildet am Anfang des Compilier-Vorgangs ab, dass alle reservierten Register kein Farbe erhalten, also nicht mit Variablen belegt werden können. Hierzu wird diesen Registern im Dictionary eine negative Zahl zugewiesen und dies geschieht per Aufzählung, was bei Änderungen am Compiler zu Fehlern führen kann. In der finalen Version wird dieses Dictionary nun mittels eines Generators erstellt, weshalb mögliche Änderungen an der Menge der reservierten Registern direkt übernommen werden.
Im ersten neuen Pass *Uncover Live* wird für jede Instruktion ausgewertet welche Ope-

randen für diese benötigt werden. Hierbei wird unterschieden ob ein Operand `a` in der k-ten Instruktion gelesen($\texttt{a} \in R(k)$) oder geschrieben ($\texttt{a} \in W(k)$) wird. Die Liste der Instruktionen wird dafür rückwärts durchgegangen, da sicher ist, dass nach dem Durchlauf des Programms keine Variable mehr live ist. Der Endzustand nach einer Instruktion ist somit immer bekannt, da er Anfangs vorgegeben ist und im Verlauf des Programms dem Anfangszustand der vorherigen Instruktion entspricht.

$$L_{after}(k) = L_{before}(k+1) \quad \text{und} \quad L_{after}(n) = \emptyset \quad \text{[[Sie23] S.49]}$$

Der neue Anfangszustand wird nun berechnet indem aus dem Endzustand die Variablen herausgenommen werden, die in dieser Instruktion einen neuen Wert zugewiesen bekommen, da ihr vorheriger Wert somit überschrieben wird und nicht von Relevanz ist. Zu dieser Menge werden nun noch die Variablen hinzugefügt, welche in dieser Instruktion gelesen werden, da ihre Werte mindestens bis zum Anfang dieser Instruktion relevant sind.

$$L_{before}(k) = (L_{after}(k) \setminus W(k)) \cup R(k) \quad \text{[[Sie23] S.49]}$$

In Abb. 7 ist ein Beispiel für die Berechnung dieser Werte angegeben.

	Instruktion	W	R	L_{after}	L_{before}
1	li a, 10	{a}	$\emptyset$	{a}	$\emptyset$
2	mv b, a	{b}	{a}	{b}	{a}
3	sub c, b,c	{c}	{b, c}	{b, c}	{b , c}
4	mv a, c	{a}	{c}	{a, b}	{b, c}
5	add a, a,b	{a}	{a, b}	{a}	{a, b}
6	addi c, a, 5	{c}	{a}	$\emptyset$	{a}

Abbildung 7: Berechnung für L_{before} für ein RISC-V Assembler Programm. W(k) und R(k) sind aus der Instruktion ablesbar, L_{before} wird von unten nach oben schrittweise berechnet.

Zuerst liest man aus den Instruktionen ab welche Werte gelesen und geschrieben werden. In RISC-V Assembler wird, bis auf ein paar Ausnahmen, das linke Register geschrieben und das oder die rechten beiden gelesen. Als nächstes berechnet man L_{before} von unten nach oben. Nach der letzten Instruktion ist L_{after} immer leer, es gilt hier also $L_{before} = (\emptyset \setminus \{\texttt{c}\}) \cup \{\texttt{a}\} = \{\texttt{a}\}$. Nun überträgt, man L_{before} einfach eine Zeile darüber in L_{after} und berechnet das nächste L_{before} über $L_{before} = (\{\texttt{a}\} \setminus \{\texttt{a}\}) \cup \{\texttt{a,b}\} = \{\texttt{a,b}\}$. So geht man weiter vor bis das letzte L_{before} berechnet ist. In diesem Beispiel sieht man gut, dass registriert wird, wenn ein Variablen-Wert nicht mehr benötigt wird, auch wenn die Variable noch einen Wert speichert. In die Variable `a` wird in der ersten Instruktion eine Zehn geladen. Nachdem in der zweiten Instruktion diese Zehn nach `b` kopiert wird, befindet `a` sich nicht mehr in L_{after}, obwohl `a` später im Programm wieder vorkommt. Dies ist der Fall, da `a` in Instruktion vier den Wert von `c` übernimmt und der vorherige Wert überschrieben wird. `a` wird also im späteren Verlauf wieder verwendet, der Wert zehn aber nicht und kann daher als nicht live angesehen werden. Das bedeutet, dass das Register, welches `a` in den ersten beiden Instruktionen repräsentiert, danach anderen Werten

zugewiesen werden kann, und, dass der Variablen `a` im späteren Programmverlauf auch ein anderes Register zugewiesen werden kann, da beide Vorkommen von `a` unabhängig voneinander sind.
Der Graph-Färbungs-Algorithmus selbst ist sowohl auf x86, als auch auf RISC-V Assemblercode anwendbar. Es gibt aber einen Unterschied in x86 und RISC-V welche Variablen innerhalb einer Instruktion geschrieben und gelesen werden. Daher müssen hier Änderungen vorgenommen werden. In x86 wird bei einer Instruktion wie *addq* `a b` die Variablen `a` und `b` gelesen und `b` geschrieben. In RISC-V hingegen werden bei der äquivalenten Instruktion *add* `a b c` die Register `b` und `c` gelesen und `a` geschrieben. Außnahme sind Speicher-Operationen wie *sd* `a imm[b]`. Hier werden `a` und `b` gelesen und der Speicher an Stelle `Memory[b+imm]` geschrieben.
In den Funktionen *read_vars* und *write_vars* muss daher umdefiniert werden welche Operanden zurückgegeben werden. Dabei fallen die Sonderfälle *movq* und *cmpq*[15] weg und die Sonderfälle *sw*,[16] *sh*,[17] *sb*[18] entstehen.

3.3 Booleans und Bedingungen

Im fünften Buchkapitel wollen wir unsere Sprache als nächsten Schritt so erweitern, dass wir auch Bedingungen einbauen können und einen ersten Kontrollfluss in den möglichen Programmen erlauben. Es sollen also nun Programme wie in Abb. 8 möglich sein.

```
y = input_int()
z = y
if not(y==2):
  x = (y==z)
else:
  x = True if (2>=3) else False
w = False
print(z if (x or w) else y)
```

Abbildung 8: Python Beispiel Code für die Sprache $\mathcal{L}_{If}$.

x86 und RISC-V handhaben Vergleiche und bedingte Sprünge unterschiedlich. Daher sind bei der Erweiterung der Sprache $\mathcal{L}_{Var}$ auf $\mathcal{L}_{If}$ um boolesche Werte *True* und *False*, sowie entsprechende Vergleichsoperatoren, und *if-else* Blöcke, einige Änderungen notwendig [Abb. 9].

In x86 wird bei Sprüngen und Vergleichen immer die Instruktion *cmpq* verwendet. Diese vergleicht die zwei Operanden und speichert das Ergebnis im *flags* Register. *cmpq a*

[15] cmpq = compare
[16] sw = store word
[17] sh = store halfword
[18] sb = store byte

$\mathcal{L}_{Var}$	exp	::=	int | input_int() | $-\,exp$ | $exp + exp$ | $exp - exp$ | (exp)
	$stmt$	::=	print(exp) | exp
	exp	::=	var
	$stmt$	::=	$var = exp$
$\mathcal{L}_{If}$	cmp	::=	$== \mid != \mid < \mid <= \mid > \mid >=$
	exp	::=	True | False | exp and exp | exp or exp | not exp
		|	$exp\ cmp\ exp$ | exp if exp else exp
	$stmt$	::=	if $exp:\ stmt^+$ else : $stmt^+$
	$\mathcal{L}_{If}$	::=	$stmt^*$

Abbildung 9: konkrete Sprache $\mathcal{L}_{If}$ frei nach [Sie23] S.66.

b kann auch als *subq a b* angesehen werden, wobei das Ergebnis verworfen wird und lediglich die entsprechenden Flags gesetzt werden, dabei sind das Zero-Flag (ZF) und das Carry-Flag (CF) entscheidend. So wird bei Gleichheit von *a* und *b* das ZF, aber nicht das CF gesetzt. Bei *a<b* wird nur das CF gesetzt und bei *a>b* wird kein Flag gesetzt. Wird *cmpq* für einen bedingten Sprung verwendet, folgt auf die Instruktion eine Instruktion *je*,[19] *jne*,[20] *jl*,[21] *jle*,[22] *jg*[23] oder *jge*,[24] welche die Bedingung für den Sprung festgelegt und ihn ausführt, sofern die richtigen Flags gesetzt sind. Wird *cmpq* für einen Vergleich verwendet folgt eine Instruktion mit *sete*,[25] *setne*,[26] *setl*,[27] *setle*,[28] *setg*[29] oder *setge*,[30] welche den Operanden entweder bei richtig gesetzten Flags auf eins, oder sonst auf null setzen.
In RISC-V hingegen werden Vergleiche und Sprünge direkt ausgewertet. Hier gibt es Instruktionen wie *sgt a b c*, welche a auf eins setzen wenn b>c gilt, sonst wird a auf null gesetzt. Ein direkten Vergleich zwischen x86- und RISC-V kann man in Tabelle 4 sehen.

Es gibt nicht für jeden Vergleich in RISC-V eine Instruktion. Es existieren lediglich die Instruktionen *slt*,[31] *sgt*[32] und *seqz*,[33] mit welchen alle 6 Vergleiche $=$, $<$, $>$, $\leq$, $\geq$ und $\neq$ umgesetzt werden müssen. Bedingte Sprünge funktionieren folgendermaßen: *beq a b addr* setzt den Programm-Counter (*pc*) auf die Adresse *addr* wenn *a==b* gilt, sonst wird der *pc* nur um eine Instruktion erhöht. Hier gibt es alle nötigen Instruktionen, allerdings

[19] je = jump if equal
[20] jne = jump if not equal
[21] jl = jump if less
[22] jle = jump if less equal
[23] jg = jump if greater
[24] jge = jump if greater equal
[25] sete = set if equal
[26] setne = set if not equal
[27] setl = set if less
[28] setle = set if less equal
[29] setg = set if greater
[30] setge = s et if greater equal
[31] slt = set if less
[32] sgt = set if greater
[33] seqz = set if equal zero

Python	x86		RISC-V	

if x < 2:
block.2
else:
block.3
	`cmpq`
	`jl`
	`jmp`
		2, x
		block.2
		block.3
			blt
			j
				x, 2, block.2
				block.3

x=(2<3)	`cmpq` `setl`	2, 3 x	*slt*	x, 2, 3

Tabelle 4: Ein Beispiel in Pseudocode wie Vergleiche und Sprünge in x86 und RISC-V gehandhabt werden.

sind die Instruktionen *ble*[34] und *bgt*[35] nur als Pseudoinstruktionen im Emulator vorhanden und werden über die anderen Branch-Instruktionen realisiert.
Diese Unterschiede zwischen beiden Assemblersprachen führen dazu, dass in dieser Stufe viele Änderungen nötig sind. Die größte Änderung erfolgt im Pass *Select Instructions*. Da RISC-V nicht direkt auf dem Stack agieren kann und große Immediates gesondert behandelt werden, müssen hier viele Abfragen und Aufteilungen realisiert werden. Ein weiteres Problem ist, dass im original Compiler genutzt wird, dass bedingte Sprünge und Vergleichsoperatoren in x86-Assembler den gleichen Suffix besitzen und sich je nach Sprung oder Vergleich lediglich der Präfix ändert. Dies wird nicht übernommen. Es wird stattdessen eine Funktion *select_branch_op* eingeführt, welche für die Auswahl der Sprunginstruktionen verantwortlich ist. Die Vergleiche hingegen werden in *select_op* behandelt.
Das Umschreiben der Vergleiche auf die drei existierenden Instruktionen wird in *expand_compare* gehandhabt. Hier werden auch große Immediates in Vergleichen behandelt, da es für den Vergleich von Immediates mit Registern gesonderte Instruktionen *sgti* und *slti* existieren. Hierbei wird genutzt, dass *a>b* auch als *b<a* ausgedrückt werden kann, da *sgti* und *slti* Immediates nur als rechten Operanden behandeln können. Für die Vergleiche $\geq$ und $\leq$ wird genutzt, dass $a \geq b$ gilt wenn $\neg(a < b)$ gilt und $a \leq b$ wenn $\neg(a > b)$. $c = a == b$ wird implementiert durch $c = a - b$, $c == 0$ mittels der Instruktion *seqz*. Für Ungleich wird dieselbe Umwandlung genutzt, mit dem Zusatz, dass das Ergebnis noch negiert wird [Tabelle 5].

In den Passes *Uncover Live* und *Assign Homes* müssen ebenfalls Änderungen vorgenommen werden. Da RISC-V im Gegensatz zu x86 keine Byte-Register hat, werden die Funktionen *vars_arg*, *collect_local_arg*, und *assign_homes_arg* nicht mehr benötigt. Für die Sprünge müssen außerdem die Argumente in *read_vars* und *collect_local_instr* beachtet werden. In *Build interference* fällt ebenfalls die Funktion *interference_inst* weg, da hier die Instruktion *movzbq* behandelt wird, die in RISC-V weg fällt. Die Änderungen

[34] ble = branch if less equal
[35] ble = branch if greater

	a und b Immediates	a Immediate	b Immediate	keine Immediates
$c = a \geq b$	$c = a, c = \neg(c < b)$	$c = \neg(b > a)$	$c = \neg(a < b)$	$c = \neg(a < b)$
$c = a \leq b$	$c = a, c = \neg(c > b)$	$c = \neg(b < a)$	$c = \neg(a > b)$	$c = \neg(a > b)$
$c = a > b$	$c = a, c = c > b$	$c = b < a$	$c = a > b$	$c = a > b$
$c = a < b$	$c = a, c = c < b$	$c = b > a$	$c = a < b$	$c = a < b$
$c = a == b$	$c = a, c = c - b, c = (c == 0)$	$c = b - a, c = (c == 0)$	$c = a - b, c = (c == 0)$	$c = a - b, c = (c == 0)$
$c = a \neq b$	$c = a, c = a - b, c = \neg(c == 0)$	$c = b - a, c = \neg(c == 0)$	$c = a - b, c = \neg(c == 0)$	$c = a - b, c = \neg(c == 0)$

Tabelle 5: Umsetzung der einzelnen Ungleichungen im Pass *Select Instructions* in der Stufe Booleans und Bedingungen. Die Aufteilung in die Spalten entsteht dadurch, dass bei der Umsetzung höchstens der letzte Operand ein Immediate sein darf.

die in *Patch Instructions* in dieser Stufe behandelt werden werden nun bereits in *Select Instructions* behandelt, weshalb hier auch viel weg fällt.

3.4 Schleifen

In der Stufe *Schleifen* aus Kapitel 6 des Buches unterstützt die Sprache $\mathcal{L}_{While}$ zusätzlich While-Schleifen [Abb. 10]. Die Python Programme, die in dieser Stufe compiliert werden können, sehen also beispielsweise so aus wie in Abb. 11.

$\mathcal{L}_{Var}$	exp	::=	int | input_int() | $-\ exp$ | $exp + exp$ | $exp - exp$ | (exp)
	$stmt$	::=	print(exp) | exp
	exp	::=	var
	$stmt$	::=	$var = exp$
$\mathcal{L}_{If}$	cmp	::=	$==$ | $!=$ | $<$ | $<=$ | $>$ | $>=$
	exp	::=	True | False | exp and exp | exp or exp | not exp
		|	$exp\ cmp\ exp$ | exp if exp else exp
	$stmt$	::=	if $exp:\ stmt^+$ else : $stmt^+$
$\mathcal{L}_{While}$	$stmt$	::=	while $exp:\ stmt^+$
	$\mathcal{L}_{While}$	::=	$stmt^*$

Abbildung 10: konkrete Sprache $\mathcal{L}_{While}$ frei nach [Sie23] S.92.

```
x = 0
while (True if (input_int() == 5) else False):
    x = x + 42
print(x)
```

Abbildung 11: Tests *while-if* für die Stufe *Schleifen* als Beispiel Code für die Sprache $\mathcal{L}_{While}$.

Hier kommt bloß ein neuer AST-Knoten *While* hinzu. Da der Compiler für diese neue Funktion lediglich bereits bestehende Konstrukte, wie die in der vorherigen Stufe eingeführten Blöcke, benötigt gibt es hier keine RISC-V spezifischen Änderungen. Hier hat

die Modularität des Compileraufbaues Vorteile und erleichtert stark die Umwandlung auf eine andere Architektur.

3.5 Funktionen

Da wir nun Python Programme mit einem komplexeren Kontrollfluss compilieren können, ist es nun an der Zeit diese Komplexität wieder aufzuspalten, indem Teile eines Programms in Funktionen ausgelagert werden können. In Kapitel 8 des Buches sollen nun also auch Programme compiliert werden können, welche Funktionen enthalten [Abb. 12]. Diese sehen zum Beispiel aus wie in Abb. 13

$\mathcal{L}_{Var}$	exp	::=	int | input_int() | $-exp$ | $exp + exp$ | $exp - exp$ | (exp)
	$stmt$	::=	print(exp) | exp
	exp	::=	var
	$stmt$	::=	$var = exp$
$\mathcal{L}_{If}$	cmp	::=	$== \mid != \mid < \mid <= \mid > \mid >=$
	exp	::=	True | False | exp and exp | exp or exp | not exp
		|	$exp\ cmp\ exp$ | exp if exp else exp
	$stmt$	::=	if $exp:\ stmt^+$ else : $stmt^+$
$\mathcal{L}_{While}$	$stmt$	::=	while $exp:\ stmt^+$
$\mathcal{L}_{Fun}$	$type$	::=	int | bool | void | tuple[$type^+$] | Callable[[$type, \ldots$], $type$]
	exp	::=	$exp(exp, \ldots)$
	$stmt$	::=	return exp
	def	::=	def $var(var : type, \ldots) -> type:\ stmt^+$
	$\mathcal{L}_{Fun}$	::=	$def \ldots stmt \ldots$

Abbildung 12: konkrete Sprache $\mathcal{L}_{Fun}$ frei nach [Sie23] S.126.

```
def add(x : int, y : int) -> int:
    return x + y

print(add(40, 2))
```

Abbildung 13: Test *add* für die Stufe *Funktionen* als Beispiel Code für die Sprache $\mathcal{L}_{Fun}$.

Wir haben bereits in Calling Conventions und Stack Frames der RISC-V Architektur gesehen, dass der Einstieg und Ausstieg aus einem Programm komplizierter ist als in x86, da Instruktionen wie `return` nicht existieren, beziehungsweise im Falle von `call` nur als Pseudoinstruktion verfügbar sind. Wir müssen für diese Stufe des Compilers darauf achten, dass wir den Stack richtig auf- und abbauen und Register die wir innerhalb der Funktionen nutzen vorher auch richtig auf dem Stack ablegen. Genau dieses Stack-Management implementieren wir bereits im *Prelude and Conclusion* Pass für Programmstart und -ende. Dies machen wir uns zunutze indem wir den generierten *Prelude* einfach bei jedem Funktionseintritt und die *Conclusion* bei jedem Verlassen einer Funktion aufrufen. Dabei können wir auf die Änderungen dieses Passes aus den letzten Stufen zurück-

greifen und müssen daher wenig anpassen. In diesem Pass wird seit der Stufe *Tuples*, welche in dieser Arbeit übersprungen wird, neben dem Stack auch der Heap gehandhabt. Da wir diesen für diese Stufe nicht explizit benötigen, ändern wir hier funktional nichts, übersetzen aber die verwendeten Befehle und Register auf RISC-V, damit der Compilerprozess nicht gestört wird.
Die Funktionsaufrufe selbst werden im Pass *Select Instructions* gehandhabt. Hier wird im x86-Compiler mit der Instruktion `leaq`[36] gearbeitet, um die Adressen der Funktionen, welche aufgerufen werden in Registern zu speichern. In RISC-V entspricht dies der Pseudo-Instruktion *la*[37] . Nachdem wir nun alle x86-Instruktionen mit den entsprechenden RISC-V Instruktionen ersetzt haben ist die letzte Stufe des Compilers fertig.
Wir können nun also Programme compilieren, welche als Operanden Integer, Variablen und Boolsche Werte handhaben können. Mit diesen Operanden können einfache mathematische Operationen durchgeführt werden $(+, -)$ und sie können verglichen werden $(<, >, ==, \leq, \geq, \neq)$. Außerdem ist ein Kontrollfluss mittels *while*-Schleifen, *if-else*-Blöcken und Funktionen möglich.

3.6 Interpreter Anpassungen

Nachdem wir nun die Änderungen im Compiler betrachtet haben, welche den Hauptteil dieser Arbeit ausmachen, wollen wir betrachten welche Änderungen im Interpreter nötig sind. Der Interpreter selbst ist stark abhängig vom Compiler, da er nur die Ergebnisse der Zwischensprachen des Compilers in Python emuliert, aber ansonsten nicht direkt von der Ziel-Architektur abhängt.
Da die vom Compiler erzeugte Zwischensprache ab dem Pass *Select Instructions* Instruktionen der Ziel-Architektur darstellt, muss der Interpreter für diesen und die folgenden Zwischensprachen angepasst werden. In der Funktion *eval_instrs* werden die meisten Änderungen vorgenommen. Hier wird das Verhalten der Register und des Speichers für die erzeugten Assembler-Instruktionen emuliert. Diese wird im Compiler von x86-Instruktionen auf RISC-V-Instruktionen abgeändert, weshalb die komplette Funktion umgeschrieben werden muss.
Die Funktion bekommt eine Liste an Assembler-Instruktionen und emuliert Instruktion für Instruktion das Verhalten der Register und des Speichers, welche als Dictionary in Klassenvariablen hinterlegt sind. Nun wird in einem großen *if-else*- Block nach den verschiedenen Instruktionen unterschieden. Für jede Instruktion wird die symbolische Darstellung der Operanden, welche vom Compiler erzeugt wird, zwischengespeichert. Falls diese Operanden in der Instruktion verarbeitet werden, so wird die symbolische Darstellung durch die Funktion *eval_arg* umgewandelt in die numerische Darstellung, welche im entsprechenden Dictionary hinterlegt ist. Im Anschluss wird das Verhalten der behandelten Instruktion emuliert, indem das erwartete Verhalten der Instruktion auf den Operanden mittels Hilfsfunktionen in Python ausgeführt wird. Das Ergebnis wird durch die Funktion

[36] leaq = load effective address
[37] la=load address

store_arg im entsprechenden Register- oder Speicher-Dictionary abgespeichert.
Für die meisten im Compiler verwendeten x86-Instruktionen gibt es sehr äquivalente RISC-V-Instruktionen, welche ähnlich emuliert werden können. Wie wir im Kapitel Booleans und Bedingungen gesehen haben, werden Sprünge und Vergleiche hingegen in beiden Architekturen unterschiedlich implementiert. Daher sind bei dieser Art Instruktionen größere Änderungen notwendig. Vergleichsinstruktionen sind unter x86 von dem `flags`-Register abhängig. Diese Anhängigkeit gibt es in RISC-V nicht. Daher können diese Instruktionen, wie alle bisherigen auch, mithilfe von Hilfsfunktionen und dem oben beschriebenen Schema emuliert werden. Sprung-Instruktionen hingegen müssen gesondert behandelt werden, da von ihrem Ergebnis abhängt an welcher Stelle der Assemblercode weiter interpretiert wird. Daher werden diese Instruktionen in einem *elif* zusammen behandelt. Das Schema ist hierbei sehr ähnlich, beide Operanden werden zunächst an die Funktion *eval_arg* übergeben. Im Anschluss werden die zurückgegebenen Werte je nach Sprungbedingung verglichen. Das Ergebnis dieser Vergleiche wird in der boolschen Variablen *perform_jump* gespeichert. Im Anschluss wird anhand dieser Variablen entschieden ob ein Sprung, zu dem in der Instruktion angegebenen Ziel, erfolgt oder nicht. Sollte der Sprung stattfinden und das Ziel der Block *conclusion* sein, so wird der Interpreter an dieser Stelle beendet, da die Instruktionen in diesem Block nur für das Aufräumen des Stacks zuständig sind. An dem Ergebnis des Programms auf Interpreter-Ebene ändert dieser Block daher nichts.
Insgesamt sieht man, dass der Interpreter selbst zwar keinen Assemblercode generiert und daher nicht direkt abhängig von der Zielsprache ist. Durch seine starke Abhängigkeit vom Compiler ist er allerdings indirekt von der Ziel-Architektur abhängig, da er ihre Instruktionen emulieren können muss.

4 Evaluation

In diesem Kapitel wollen wir nun den RISC-V Compiler, dessen Entstehung wir im letzten Kapitel betrachtet haben, evaluieren. Dabei interessiert uns einerseits welche Python Programme korrekt nach RISC-V übersetzt werden, als auch wie schnell der Compiler und die entstehenden Assembler-Programme sind. Am Ende werden wir die Ergebnisse beider Untersuchungen bewerten um einen Eindruck zu gewinnen wie gut der hier gebaute Compiler performt.

4.1 Testen der implementierten Funktionalitäten

Als erstes wollen wir die Korrektheit des gebauten Compilers evaluieren. Hierfür sehen wir uns die im Support-Code vorhandenen Tests als Messwerte an. Für die Stufen Integer und Variablen bis Schleifen laufen alle Tests erfolgreich durch. Hier scheinen die Test keine Lücken oder Fehler beim neue RISC-V Compiler auf zu zeigen. Ob dies damit gleich zu setzen ist, dass der RISC-V Compiler Fehlerlos funktioniert wird im Abschnitt Bewertung der Ergebnisse diskutiert.
Für die letzte behandelte Stufe Funktionen laufen allerdings nicht alle Tests fehlerfrei durch, wie auch in Tabelle 6 zu sehen ist.

Stufen	#vorhandener Tests	#laufender Tests	welche Tests werden aufgeführt
Integer und Variablen	32	32	var, int64
Register Allocation	32	32	var, int64
Booleans und Bedingungen	75	75	var, int64, if
Schleifen	82	82	var, int64, if, loop
Funktionen	96	93	var, int64, if, loop, function

Tabelle 6: Angabe der Tests-Kategorien die für die einzelnen Stufen des Compilers vorhanden sind und wie viele durchlaufen.

Hier schlagen 3 Tests fehl und zwar *fun-in-tuple*, *map-tuple* und *collect-in-fun*. Wenn man sich diese Tests und ihre Unterschiede zu den durchlaufenden Tests anguckt, fällt auf, dass dies die einzigen Tests sind, welche Tuple benutzen. Daher werden hier im Compiler Funktionen aufgerufen und Zeiger gesetzt, welche nicht auf RISC-V umgebaut wurden, da diese Stufe des Compilers ausgelassen wurde. Diese Tests funktionieren also nicht, da sie Konstrukte benötigen, welche außerhalb der Funktion dieser Stufe liegen. Das bedeutet aber auch, dass die Tests nicht wie die Kapitel in [Sie23] unabhängig voneinander sind.
Außerdem ist die vorgegebene Test Infrastruktur etwas fehleranfällig, da das Bestehen der Tests von der Richtigkeit der *golden*-Dateien abhängt. Sollten hier falsche Werte stehen, so schlagen Tests fehl, obwohl sie im Zweifelsfall den richtigen Assemblercode generieren. Besser wäre es mit eine vorgefertigten Testumgebung wie pytest zu arbeiten

und die Vergleichsdaten der *golden*-Dateien durch einen Testlauf der Pythonprogramme neu zu generieren.
Zusammenfassend lässt sich für den RISC-V Compiler sagen, dass für alle Stufen alle Tests, welche aufgrund der implementierten Funktionalität für RISC-V erfolgreich durchlaufen können, dies auch tun. Die Testumgebung selbst sollte aber bei weiterer Arbeit an diesem Projekt neu aufgesetzt werden.

4.2 Testen der Leistung des Python-Compilers

Nachdem wir uns die Korrektheit des Compilers angeguckt haben, vergleichen wir nun auch die Geschwindigkeit des erzeugten Assemblercodes und betrachten ob er mit der Laufzeit des nativen Python Interpreter mithalten kann. Hierfür wurden 2 Programme ausgewählt. Das Erste implementiert die Suche nach der n-ten Fibonacci Zahl rekursiv [Abb. 14(a)]. Die Laufzeit hierfür liegt in $\mathcal{O}(2^n)$. Das zweite Programm implementiert eine einfache Schleife, welche n-mal durchlaufen wird [Abb. 14(b)]. Die Laufzeit hierfür liegt in $\mathcal{O}(n)$. Es wurden diese beiden Programme gewählt, da man bei diesen die Laufzeit gut skalieren kann um die beiden verwendeten Compiler bzw. Interpreter gut vergleichen zu können.

```
def fib(n:int)->int:

  if n == 1 or n == 2:
    return 1
  else:
    return fib(n-1) +
        fib(n-2)

print(fib(3))
```

(a) Python Testprogramm fibonacci.py

```
def loop(n:int)->int:
    j = 0
    big = False
    while not big:
        j = j + 1
        if j>=n:
            big = True
    return j

print(loop(10))
```

(b) Python Testprogramm loop.py

Abbildung 14

Die Bewertung des gebauten Compilers anhand der Laufzeiten ist allerdings nicht aussagekräftig, da wir hier die Laufzeit eines Interpreters mit der addierten Laufzeit des Compilers und Emulators vergleichen. Aus den Daten lässt sich daher nicht ablesen welcher Teil der Laufzeit dem generierten Assemblercode und welcher Teil dem Emulator zuzuordnen ist. Diese Daten werden bei der Bewertung des Ergebnisses daher keine große Rolle spielen.
Dennoch wollen wir betrachten ob sich diese kombinierte Laufzeit und die des Interpreters in derselben Größenordnung bewegen.

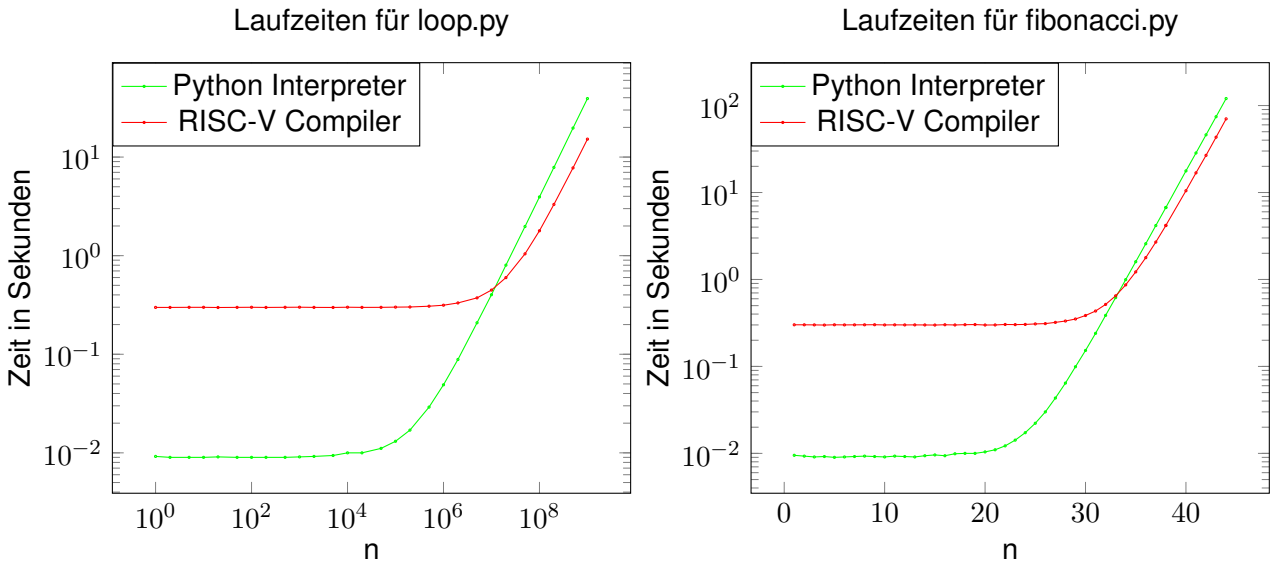

Abbildung 15: Laufzeiten für die n-te rekursiv berechnete Fibonacci Zahl und eine n-fache Schleife vom Python-Interpreter und dem hier gebauten Python-Compiler, abhängig von n.

In Abb. 15 sieht man die Graphen der Laufzeiten abhängig von n. Hier lässt sich für beide Programme erkennen, dass der RISC-V Compiler zumindest bei kleinen Zahlen langsamer läuft als der Interpreter. Dies ist auch nicht weiter verwunderlich, da der native Python Interpreter die Programme auf x86 interpretiert und der so entstehende Code direkt auf der x86-CPU ausgeführt werden kann. Unser Compiler hingegen compiliert für RISC-V, welches ohne entsprechendes Board nicht direkt ausführbar ist. Daher verwenden wir QEMU. Das heißt allerdings, dass wir hier nicht unbedingt die Laufzeit des Compilers mit der des Interpreters vergleichen, sondern viel mehr die von QEMU und dem Interpreter, da die Compilierzeit selbst nicht von der Größe von n abhängt.
Für größere n nähern sich beide Kurven aneinander an und ab einem gewissen Punkt ist der RISC-V Compiler in beiden Fällen sogar etwas schneller als der Python-Interpreter. Dies zeigt uns, dass sich die Laufzeiten des entwickleten Compilers in der richtigen Größenordnung bewegen. Ohne eine Testumgebung mit direkt vergleichbaren Werten können wir nicht mehr über die Laufzeiten aussagen.

4.3 Bewertung der Ergebnisse

Nachdem wir betrachtet haben welche Tests der RISC-V Compiler besteht und wie sich seine Laufzeit mit der des nativen Python-Interpreters vergleicht, wollen wir die gefundenen Ergebnisse nun bewerten.
Da wir hier den entstandenen RISC-V Compiler bewerten wollen, und nicht QEMU, können wir die Ergebnisse aus Testen der Leistung des Python-Compilers nicht wirklich mit

einbeziehen. Um vergleichbare Ergebnisse zu erzielen müsste der RISC-V Compiler auf einer RISC-V Architektur ausgeführt werden. Dieses Vorhaben würde den Rahmen dieser Arbeit allerdings sprengen. Konzentrieren wir uns also auf die Tests. Hier haben wir festgestellt, dass der RISC-V Compiler alle Tests besteht welche er bestehen kann. Dies bedeutet aber nicht zwangsweise, dass er jedes mögliche Python Programm fehlerfrei übersetzt. Die Tests welche im Support-Code enthalten sind decken nicht alle Möglichkeiten ab welche die Sprache L_{fun} erlaubt. Dies liegt daran, dass der Compiler für RISC-V einen drei-Adressen-Code implementiert, in x86 allerdings nur 2 Operanden pro Instruktion erlaubt sind. Dadurch gibt es Kombinationen aus Instruktionen und Operanden, speziell Immediates, welche in RISC-V möglich sind und in x86 nicht. Diese werden durch die Tests nicht abgefragt.
Über die Korrektheit des RISC-V Compiler lässt sich also ohne weitere Tests nur die Aussage treffen, dass allgemeine Fälle richtig compiliert werden, Randfälle aber unter Umständen fehlschlagen können.

5 Verwandte Arbeiten

In dieser Arbeit wurde anhand eines Lehrbuches für einen Compiler, welcher Python nach x86 übersetzt, ein Compiler von Python nach RISC-V geschrieben. Direkt verwandte Literatur, welche ebenfalls ein angewandtes Lehrbuch im Compilerbau als Vorlage nimmt und die Zielsprache ändert gibt es nicht. Es gibt allerdings verwandte Arbeiten zu dem hier bearbeiteten Lehrbuch [Sie23]. Dieses Buch bedient sich einem sehr praktischen Ansatz und gibt dem Leser neben konkreten Programmieraufgaben auch Support-Code für den entstehenden Compiler an die Hand.
Einen ähnlichen Ansatz verwendet das Buch *Compiling to Assembly: From Scratch* [[Kel20]]. Hier wird eine Teilsprache von TypeScript mittels eines in TypeScript geschriebenen Compilers nach 32-Bit ARM übersetzt. Hier wird dem Leser weniger die Aufgabe gestellt einen Compiler mittels der beschriebenen Prinzipien selbst zu schreiben, viel mehr wird ein vom Autor geschriebener Compiler vorgestellt und erklärt.
Neben diesen beiden angewandten Lehrbüchern gibt es noch weitere, welche das Thema Compilerbau an sich vermitteln. Die Bücher *Compiler Prinzipien, Techniken und Werkzeuge* [[ULSA08]] und *Modern Compiler Design: Second Edition* [[GRCJJ12]] verwenden diesen Ansatz, wobei Ersteres dem Leser Aufgaben an die Hand gibt um das Thema weiter zu verinnerlichen und Zweiteres dafür dem Leser viele Prinzipien mittels Pseudocode vermittelt.
Wie das Thema Compilerbau für Studenten, welche auch Zielgruppe von [Sie23] sind, am besten gestaltet werden kann, wird in dem Artikel *EDUCATIONAL PEARL: A Nanopass framework for compiler education* [[SWD05]] betrachtet. Hier wird die Idee diskutiert den komplexen Zusammenhang eins Compilers in kleine, möglichst unabhängige, Unterabschnitte (Passes) zu unterteilen, um dem Studenten die Funktionalitäten möglichst einfach nahe zu bringen.
Eine andere Art der Verwandten Arbeit stellt *riscv-gnu-toolchain* [[Dab14]] dar. Hier wird ein Compiler von C auf RISC-V zur Verfügung gestellt. Host- und Quellsprache sind somit verschieden, aber die Zielsprache ist gleich, wodurch sich die Backends ähnlich gestalten.
Neben diesen Lehrbüchern bietet der Kurs *From Nand to Tetris: Building a Modern Computer From First Principles* [[NN17]] eine Anleitung zum selber bauen eines einfachen Compilers. Hier wird ebenfalls mit einem sehr angewandten Ansatz ein Compiler in mehreren Schritten gebaut. Ziel-und Quellsprache allerdings werden in dem Projekt selbst entwickelt und entsprechen keiner vorher vorhandenen Programmiersprache. Außerdem wird in dem Kurs neben dem Compiler in den ersten Projekten die Hardware erschlossen und zuteilen programmiert. Und in späteren Projekten wird ein Betriebssystem programmiert, sodass Teilnehmer am Ende des Kurses einen Computer mit grundlegenden Funktionen von Grund auf selbst programmiert und erschlossen haben.
Man sieht es gibt viele Arbeiten die sich grundlegend mit dem Thema Compilerbau beschäftigen und auch einige die einen sehr angewandten Ansatz wählen. Dennoch behandelt diese Arbeit ein Thema was in dieser Art bisher nicht behandelt wurde.

6 Zusammenfassung und Ausblick

Am Ende dieser Arbeit lässt sich sagen, dass der entstandene RISC-V Compiler für die im Kurs vermittelten Algorithmen und zur Verfügung stehenden Tests funktioniert. Das Übertragen der theoretischen Prinzipien aus [Sie23] auf RISC-V gelingt sehr einfach. Das Buch selbst ist damit auch für andere Architekturen als die dort verwendete x86-Architektur möglich. Wird beim Bau des Compilers zusätzlich der Support Code verwendet, dann stellt sich dies anders dar. Der zur Verfügung gestellte Code weist bei der Übertragung auf RISC-V mehrere Schwächen auf. Er ist sehr nah an der x86 Architektur geschrieben und muss an mehreren Stellen angepasst werden, damit der einstehende Assembler Code aus der RISC-V Architektur läuft. Das Wissen, welches für diese Änderungen benötigt wird, wird im Buch nicht vermittelt. Dies bedeutet einen erheblichen Mehraufwand für jede neue Umsetzung auf eine neue Architektur.
Der entstandene Compiler kann eine Teilsprache von Python nach RISC-V übersetzen. Die möglichen Programme sind dabei dadurch stark eingeschränkt, dass sie nur Integer verarbeiten können. Eine zusätzliche Einbindung von Strings und Float Werten wäre für eine zukünftige Arbeit an diesem Projekt wünschenswert. Ebenfalls geeignet für zukünftige Arbeiten wäre das implementieren von *Kapitel 7-Tuples and Garbage Collection* da mit der Organisation des Heaps auch Objekte implementierbar wären, was die Nutzbarkeit des Compilers stark erhöhen würde. Diese Änderungen würden sich nah an dem Lehrbuch orientieren und sind mit einem geringen bis mittleren Zeitaufwand realisierbar. Im Hinblick auf die Korrektheit aktueller und zukünftiger Implementationen wären auch zusätzliche Tests, welche auf die RISC-V spezifischen Änderungen am Compiler abzielen erstrebenswert. Hier müsste man sich genauer mit der Frontend des Compilers auseinander setzen um zu sehen wie Randfälle im AST aussehen könnten und zustande kommen. Der Aufwand hierfür wäre daher recht hoch, da in dieser Arbeit bisher nur das Backend betrachtet wurde.

Abbildungsverzeichnis

1 RISC-V Assembler Code für den Aufruf einer einfachen Addier-Funktion aus der *main*. 5

2 Beispiel für den Inhalt der Testdateien. Die doppelten Pfeile zeigen dabei den Inhalt einer Datei an. Die einfachen Pfeile zeigen den zeitlichen Ablauf des Tests an, beginnend bei $\langle test \rangle$.in. 9

3 konkrete Sprache $\mathcal{L}_{Var}$ frei nach [Sie23] S.14. 11

4 Python Beispiel Code für die Sprache $\mathcal{L}_{Var}$. 12

5 Die Funktion *shift* aus *var.py*, welche den Umgang mit Immediates größer als 31-Bit regelt. 12

6 Darstellung eines 64-Bits Wertes in Binädarstellung mit Angabe der Paritätsbits für *li* und *addi*. 13

7 Berechnung für L_{before} für ein RISC-V Assembler Programm. W(k) und R(k) sind aus der Instruktion ablesbar, L_{before} wird von unten nach oben schrittweise berechnet. 15

8 Python Beispiel Code für die Sprache $\mathcal{L}_{If}$. 16

9 konkrete Sprache $\mathcal{L}_{If}$ frei nach [Sie23] S.66. 17

10 konkrete Sprache $\mathcal{L}_{While}$ frei nach [Sie23] S.92. 19

11 Tests *while-if* für die Stufe *Schleifen* als Beispiel Code für die Sprache $\mathcal{L}_{While}$. 19

12 konkrete Sprache $\mathcal{L}_{Fun}$ frei nach [Sie23] S.126. 20

13 Test *add* für die Stufe *Funktionen* als Beispiel Code für die Sprache $\mathcal{L}_{Fun}$. . 20

14 24

15 Laufzeiten für die n-te rekursiv berechnete Fibonacci Zahl und eine n-fache Schleife vom Python-Interpreter und dem hier gebauten Python-Compiler, abhängig von n. 25

Tabellenverzeichnis

1 RISC-V Register und ihre x86 Äquivalente in diesem Compiler inklusive der zugehörigen Savern [[WLPA14] S.137, [III18] S.147]. 4

2 Der Verlauf des Stack für das in Abb. 1 abgebildete Programm. 4

3 Die Passes werden in dieser Reihenfolge in den einzelnen Stufen des Compilers abgearbeitet. Markiert sind hier jeweils die Passes die in den einzelnen Stufen geändert oder hinzugefügt werden. 7

4 Ein Beispiel in Pseudocode wie Vergleiche und Sprünge in x86 und RISC-V gehandhabt werden. 18

5 Umsetzung der einzelnen Ungleichungen im Pass *Select Instructions* in der Stufe Booleans und Bedingungen. Die Aufteilung in die Spalten entsteht dadurch, dass bei der Umsetzung höchstens der letzte Operand ein Immediate sein darf. 19

6 Angabe der Tests-Kategorien die für die einzelnen Stufen des Compilers vorhanden sind und wie viele durchlaufen. 23

Literatur

[aut23] AUTHORS various: *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1:Basic Architecture*. 2023

[CAC+81] CHAITIN, Gregory J. ; AUSLANDER, Marc A. ; CHANDRA, Ashok K. ; COCKE, John ; HOPKINS, Martin E. ; MARKSTEIN, Peter W.: Register allocation via coloring. In: *Computer Languages* 6 (1981), Nr. 1, S. 47–57

[CCS00] CRYSTAL CHEN, Greg N. ; SHIMANO, Kirk: *risc vs. cisc*. 2000. – https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/ [Accessed: (2023-04-01)]

[Cha82] CHAITIN, G. J.: Register Allocation & Spilling via Graph Coloring. In: *SIGPLAN Not.* 17 (1982), jun, Nr. 6, S. 98–101

[Dab14] DABBELT, Palmer: *riscv-collabriscv-gnu-toolchain*. 2014. – https://github.com/riscv-collab/riscv-gnu-toolchain [Accessed: (2023-04-01)]

[Dev22] DEVELOPERS, The QEMU P.: *About QEMU*. 2022. – https://www.qemu.org/docs/master/about/index.html [Accessed: (2023-04-12)]

[GRCJJ12] GRUNE, Dick ; REEUWIJK, Henri E. B. v. ; CERIEL J.H. JACOBS, Koen L.: *Modern Compiler Design: Second Edition*. 2012

[III18] III, Harry H. P.: *RISC-V: An Overview of the Instruction Set Architecture*. 2018. – http://web.cecs.pdx.edu/~harry/riscv/RISCV-Summary.pdf [Accessed: (2023-04-04)]

[Jam95] JAMIL, Tariq: RISC versus CISC: Why less is more. In: *IEEE Potentials* 14 (1995), Nr. 3, S. 13–16

[Kel20] KELESHEV, Vladimir: *Compiling to Assembly: From Scratch*. 2020. – https://keleshev.com/compiling-to-assembly-from-scratch/ [Accessed: (2023-04-24)]

[NN17] NOAM NISAN, Shimon S.: *From Nand to Tetris:Building a Modern Computer From First Principles*. 2017. – https://www.nand2tetris.org/ [Accessed: (2023-04-24)]

[Sie20] SIEK, Jeremy G.: *IUCompilerCoursepython-student-support-code*. 2020. – https://github.com/IUCompilerCourse/python-student-support-code [Accessed: (2023-04-04)]

[Sie23] SIEK, J.G.: *Essentials of Compilation: An Incremental Approach*. MIT Press, 2023

[SWD05] SARKAR, Dipanwita ; WADDELL, Oscar ; DYBVIG, R. K.: EDUCATIONAL PEARL: A Nanopass framework for compiler education. In: *Journal of Functional Programming* 15 (2005), S. 653–667

[ULSA08] ULLMAN, Jeffrey D. ; LAM, Monica S. ; SETHI, Ravi ; AHO, Alfred V.: *Compiler Prinzipien, Techniken und Werkzeuge*. Pearson Deutschland, 2008

[Wil19] WILCOCK, Tyler: *RISC-V from scratch 4: Creating a function prologue for our UART driver (2 / 3)*. 2019. – `https://twilco.github.io/riscv-from-scratch/2019/07/28/riscv-from-scratch-4.html#a-prologue-to-function-prologues-and-epilogues` [Accessed: (2023-04-30)]

[WLPA14] WATERMAN, Andrew ; LEE, Yunsup ; PATTERSON, David A. ; ASANOVIĆ, Krste: The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1 / EECS Department, University of California, Berkeley. 2014 (UCB/EECS-2014-54). – Forschungsbericht