Zusammenfassung der Projektarbeit

Untersuchung der Transformation verschiedener Maschinenspracharchitekturen

Author: Sebastian Kunz
Betreuender Professor: Prof. Dr. Harald Görl

Universität der Bundeswehr München

Inhaltsverzeichnis

1. Allgemeine Informationen (Einleitung)

Befehlssatzarchitektur (ISA)
Eine Befehlssatzarchitektur (ISA) beschreibt alle Befehle, welche von einer zugehörigen Architektur verarbeitet werden können. Hierbei können verschiedene CPU Modelle die gleiche Architektur verwenden. So verwenden z.B. viele der von Intel produzierten CPUs die x86 (32Bit) oder deren 64Bit Erweiterung x86_64 Architektur. Die einzelnen Befehle einer Befehlssatzarchitektur können sowohl in Binärer- bzw. Hexadezimaler Darstellung (Maschinencode) oder mithilfe von Mnomics und Assembler als für Menschen lesbaren Code dargestellt werden. Je nach Architektur unterscheiden sich die Befehle, welche durch den Maschinencode ausgedrückt und somit verarbeitet werden können. Die Befehlssatzarchitekturen unterscheiden sich primär in den folgenden Eigenschaften:

Eigenschaften von Befehlssatzarchitekturen

1. Typ des Befehlssatzes

- CISC – „Complex Instruction Set Computing“
  - Großer Befehlssatz
  - Komplexe Befehle
  - Unterschiedliche Dauer bei Ausführung der Befehle (kann mehrere Taktzyklen benötigen)
  - Große Bauart

- RISC – „Reduced Instruction Set Computing“
  - Kleiner Befehlssatz
  - Simple Befehle
  - Ein Takt pro Befehl
  - Kleine Bauart
  - Energieeffizient

Darüber hinaus bestehen noch weitere Befehlsatz-Typen, welche allerdings für die kommerzielle Nutzung in Form von Desktop PC´s oder Smartphones (und damit auch für die Projektarbeit) eine eher untergeordnete Rolle spielen. Im vergleich zu RISC und CISC setzen diese auf parallele Verarbeitung von Befehlen. Hierzu zählen:

- VLIW – „Very Long Instruction Word“
- EPIC – „Explicitly Parallel Instruction Computing“

2. Bitbreite

Die Bitbreite einer Befehlsatzarchitektur unterscheidet sich in der Größe, der Daten- und Adressregister sowie der Verarbeitungseinheiten.
Primär ist allerdings die Breite des Datenregisters wichtig, wodurch diese zumeist als die Bitbreite der Befehlssatzarchitektur angesehen wird.

Gängige Bitbreiten sind hierbei:

- 8 Bit (Lediglich Mikrocontroller)
- 16 Bit
- 32 Bit
- 64 Bit (Aktueller Standard bei Desktop PCs und Smartphones)

32Bit war lange Zeit der Standard bei Desktop PC´s und Smartphones. Aufgrund des höheren Speicherbedarfs und der Verarbeitung von größeren Dateien ist heute allerdings 64Bit Standard, da mit einem 32Bit System schlichtweg nicht alle Adresse angesteuert werden könnten.
Beispielsweise ist es 32Bit Systemen lediglich möglich etwa 4GB (2^32Bit) an Arbeitsspeicher anzusteuern, was für moderne PC´s nicht mehr ausreichend ist.

3. Art und Anzahl der Register

Die Art und Anzahl der verfügbaren Register fließt direkt in die Kodierung eines Befehlssatzes mit ein. So gibt es Beispielsweise Architekturen welche eigens für Gleitkommazahlen vorgesehene Register bereitstellen, wohin gegen andere nur die Speicherung von Integer Werten ermöglichen und somit eine Umrechnung notwendig ist. Die Anzahl an Registern kann ebenfalls variieren, wobei kleinere Anzahlen an Registern dazu führen können, dass Werte öfters im langsameren Arbeitsspeicher abgelegt werden müssen.
Die Anzahl kann allerdings auch innerhalb einer Architektur variieren, da möglicherweise lediglich eine Mindest- und keine konkrete Anzahl an Registern vorgeben wird.

4. Operandenanzahl

Die Menge an Operanden, welche eine CPU entgegennehmen kann, ist ein sehr wichtiges Kriterium und hat ebenfalls einen direkten Einfluss auf die Kodierung des Befehlssatzes. Hierbei wird in folgende Typen unterschieden:

- *Ein-Adress-Architektur:*
  Ein Befehl holt maximal einen Operanden aus dem Arbeitsspeicher. Werden mehr Operanden benötigt müssen diese vorab in interne Prozessorregister (meistens den *Akkumulator*) geladen werden.

- *Zwei-Adress-Architektur:*
  Ein Befehl holt maximal zwei Operanden aus dem Arbeitsspeicher, beispielsweise die Summanden einer Addition. Das Ergebnis kann dann z.B an einem der Speicherplätze der übergebenen Operanden oder einem allgemeinen (für alle Operationen dieser Art eingesetzten) Register gespeichert werden.

- *Drei-Adress-Architektur:*
  Ein Befehl holt maximal drei Operanden aus dem Arbeitsspeicher. Typischerweise die beiden Operanden einer arithmetischen oder logischen Verknüpfung und als dritten Operanden die Adresse, wohin das Ergebnis zurückgespeichert werden soll.

Darüberhinaus existiert eine **Null-Adress Variante**, wie die Stack-Architektur. Bei dieser Architektur werden beispielsweise durch einen Additionsbefehl zwei Werte, welche bereits auf dem Stack liegen, addiert. Somit muss kein zusätzlicher Operand mit dem Befehl übergeben werden. Um die Addition korrekt auszuführen, müssen allerdings zuvor beide Summanden auf den Stack gelegt werden. Die Null-Adress-Architektur spielt im Vergleich zu den anderen Architekturen eine eher untergeordnete Rolle.

2. Von Programmier- zur Maschinensprache

Durch die Kompilation von Programmcodes, wie z.B. C-Code, wird einen für die Zielarchitektur lesbarer Maschinencode erzeugt. Sogenannte Crosscompiler (wie GCC oder LLVM) ermöglichen hierbei, das Kompilieren von Programmen für andere Zielsysteme als das Hostsystem auf dem der Kompilierungsvorgang durchgeführt wird. Es kann beispielsweise x86 Maschinencode auf einem Arm64 System erzeugt werden.
Im folgenden erläutere ich wie aus dem gegebene C-Codes, unter Verwendung des GCC-Compilers ohne Optimierung, der zugehörige x86 Maschinencode erzeugt wird. Zur besseren Lesbarkeit wird die Assembler-Darstellung mit Intel-Syntax verwendet.

Ausgangscode in C:

```
1|   int add4(int a) {
2|        int result = a + 4;
3|        return result;
4|   }
```

Resultierender x86 Assemblercode (Intel-Syntax)

```
1|   push    rbp
2|   mov     rbp, rsp
3|   mov     dword ptr [rbp - 4], edi
4|   mov     eax, dword ptr [rbp - 4]
5|   add     eax, 4
6|   pop     rbp
7|   ret
```

1. Aufbau des Stack-Frames

Beim Aufruf einer Funktion, repräsentiert durch Zeile eins im Ausgangscode, wird der sog. Stack-Frame aufgebaut. Hierfür wird zuerst der Stack-Frame der aufrufenden Funktion gesichert, um im Anschluss der aktuellen in Ausführung befindlichen Funktion diesen wieder laden zu können. Hierbei wird die Basisadresse der aufrufenden Funktion, welche zu diesem Zeitpunkt noch im Register rbp hinterlegt ist, auf dem Stack abgelegt (PUSH, Zeile 1, Resultierender Code), wo sie später wieder entnommen werden kann.
Anschließend wird die Basis-Adresse der aktuellen Funktion im Register des Base-Pointers gesichert. Die Adresse der aufgerufenen Funktion wurde zuvor auf den Stack gelegt, weshalb einfach der Inhalt des Stack-Pointers mittels mov Befehl in das Register des Base-Pointers bewegt wird (Zeile 2, Resultierender Code).

```
1|   push    rbp
2|   mov     rbp, rsp
```

2. Vorbereitung für die arithmetische Operation

Als nächstes werden die an die Funktion übergebenen Parameter auf der Stack geladen, um mit ihnen arbeiten zu können. Für die übergebenen Parameter stehen eigens reserviert Register bereit. In unserem konkreten Beispiel beinhaltet das Register edi den übergebenen Parameter (int a, Zeile 1, Ausgangscode). Die Adressierung der Stack-Adressen wird mithilfe eines Offsets relativ zur Basis-Adresse der aktuellen Funktion realisiert.

Mithilfe des mov Befehls (Zeile 3, Resultierender Code) wird der Wert des Übergabeparameters (Register edi) an die Adresse rbp-4 bewegt.

Im Anschluss wird der nun in rbp-4 befindliche Wert des übergebenen Parameters in das Register verschoben, welches für den Rückgabewert von Funktionen vorgesehen ist (eax).

```
3|   mov     dword ptr [rbp - 4], edi
4|   mov     eax, dword ptr [rbp - 4]
```

3. Ausführung der Arithmetischen Operation

Um die Addition des konstanten Wertes durchzuführen (Zeile 2, Ausgangscode), wird auf den Wert des übergebenen Parameters, welcher sich inzwischen im Register eax befindet, der konstante Wert, also die Zahl vier, mittels des add Befehls (Zeile 5, Resultierender Code) addiert.

```
5|   add     eax, 4
```

4. Abbau des Stack-Frames

Zuletzt wird nun die in Schritt eins gespeicherte Basis-Adresse der aufrufenden Funktion, in das entsprechende vorgesehene Register geladen (POP, Zeile 6, Resultierender Code) und im Anschluss an die Stelle im Code zurückgesprungen, an welcher die nun ausgeführte Funktion aufgerufen wurden (Zeile 7, Resultierender Code).

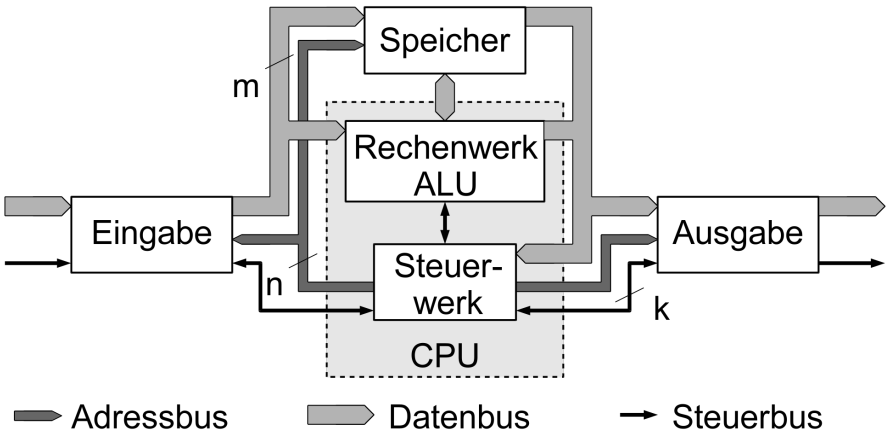
```
6|   pop     rbp
7|   ret
```

Hinweis:

Da es sich hierbei um ein sehr einfaches Beispiel handelt, welches ohne Optimierung übersetzt wurde, scheinen einige Zeilen auf den ersten Blick sinnlos und könnten durchaus optimiert werden. So könnte beispielsweise der Übergabeparameter (edi) direkt in das Rückgaberegister (eax) verschoben werden, was allerdings bei komplexeren Funktionsinhalten zu Problemen führen könnte. Da derartiger Code schnell unübersichtlich werden kann, handelt sich bei diesem Beispiel lediglich um eine Verdeutlichung wie ein Programmcode aussehen könnte.

3. Ausführung von Maschinencode in der CPU

Der aus Befehl und Operanden bestehende, kompilierte Maschinencode wird im Anschluss durch den Linker in einer ausführbare Datei untergebracht, welche neben dem Maschinencode noch weitere Informationen (Metadaten usw.) enthält. Bei einem Programmstart wird der entsprechende Programmcode in den Arbeitsspeicher geladen und zur Ausführung gebracht.
Während der Ausführung zeigt der Programmcounter (PC) immer auf den aktuell in Ausführung befindlichen Befehl im Speicher. Das Steuerwerk (Control-Unit, CU) wertet diesen Befehl aus und stellt die Weichen für die weitere Hardware. Handelt es sich beispielsweise um einen arithmetischen Befehl wird die Arithmetic Logic Unit (ALU) aktiviert die Operanden zu verwerten. Müssen hierfür erst Werte aus dem Speicher geholt werden, wird ein entsprechender Ladebefehl ausgeführt usw. Das beschrieben Konzept basiert auf der Von-Neumann-Architektur, welche im folgenden Bild dargestellt ist:



Quelle: https://wissensplattform-schueler.de/von-neumann-

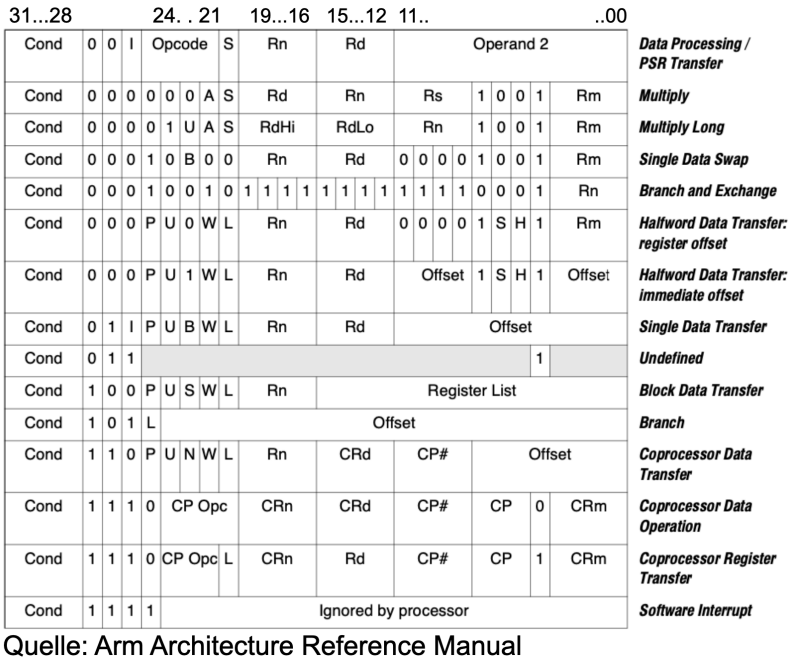
4. x86 vs Arm

x86	arm
55	ff 43 00 d1
48 89 e5	e0 0f 00 b9
89 7d fc	e8 0f 40 b9
8b 45 fc	08 11 00 11
83 c0 04	e8 0b 00 b9
5d	e0 0b 40 b9
c3	ff 43 00 91
90	c0 03 5f d6

Die relevantesten Architekturen im kommerziellen Bereich stellen die beiden Hersteller Intel und Arm, weshalb diese im Folgenden näher betrachtet werden sollen. Mit ihrer x86_64 Architektur verfolgt Intel das CISC Prinzip, wohingegen ARM bei ihrer Arm_64 Architektur auf das Prinzip eines RISC Befehlssatzes setzt. Neben diesem grundlegenden Unterschied stechen aber auch beim betrachten des Maschinencodes (im folgenden in Hexadezimaler Darstellung) die Unterschiede direkt ins Auge. Während bei ARM Alle Instruktionen die gleiche länge haben, variieren die Längen der Instruktionen bei x86. Hierdurch und durch die CISC-Typische Zusammenfassung mehrer kleinere Befehle durch einen komplexeren, sind x86 Programme tendenziell kleiner als die gleichen Programme ARM Maschinen. Die durchweg gleichbleibende Länge der Instruktionen bei ARM hat aber durchaus andere Vorteile. So kann beispielsweise der Programmcounter immer um die feste Länge eines Befehls (hier 32Bit) erweitert werden, während bei x86 eine Auswertung des Befehls und die Ermittlung dessen Länge notwendig ist. Was im vergleich zu einer konstanten Addition mehr Hardware benötigt und zudem in längeren Laufzeiten resultieren kann.

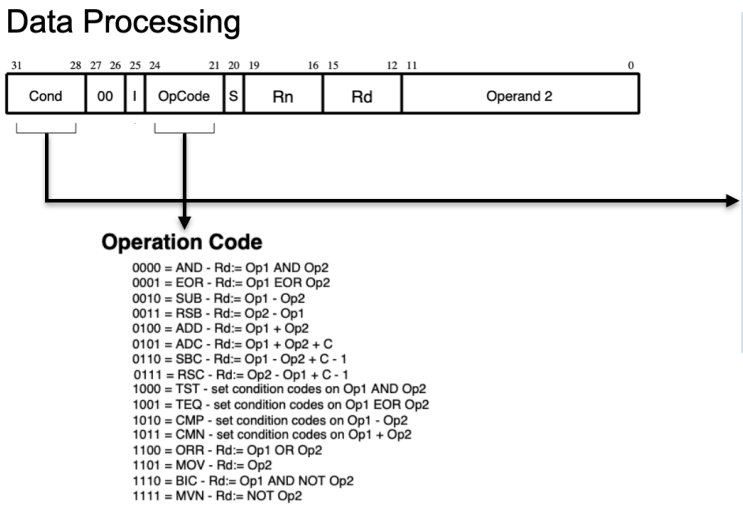
Bei den hier gezeigten x86 Befehlen beschreibt der erste Hexadezimale Block (die ersten 8 Bit) immer den Befehl. Im Anschluss folgen in Block zwei und drei die Operanden. Operationen die häufig vorkommen, wie beispielsweise die Operationen welche zum Aufbau des Stack-Frames benötigt werden, haben eigene Instruktionen, wodurch diese teilweise nur aus einem Block bestehen (z.B. stellt die 55 den Befehl „push rbp" aus dem Beispiel in Kapitel 2 dar). Bei ARM Befehlen weichen diese Zuweisungen von der Hexadezimalen Darstellung zu der konkreten Instruktion deutlich ab. Je nach Operation kann die konkrete Aufteilung variieren.

Dies ist im folgenden Ausschnitt der ARM-Dokumentation zu sehen.

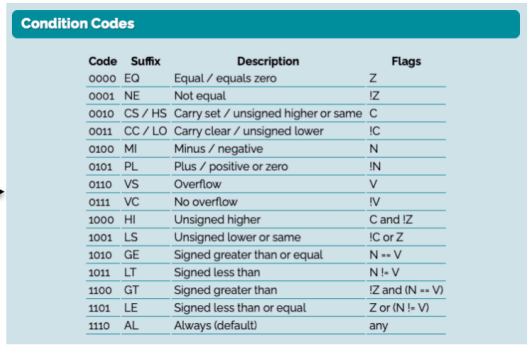


Quelle: Arm Architecture Reference Manual

Sind die Bits 25 bis 27 mit den Werten 001 beschrieben, handelt es sich um einen Data-Processing bzw. PSR-Transfer Befehl, welcher wiederum eine eigene Decodierung aufweist.
Dies ist im folgenden Bild dargestellt:



Quelle: Arm Architecture Reference Manual



Condition Codes

Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any

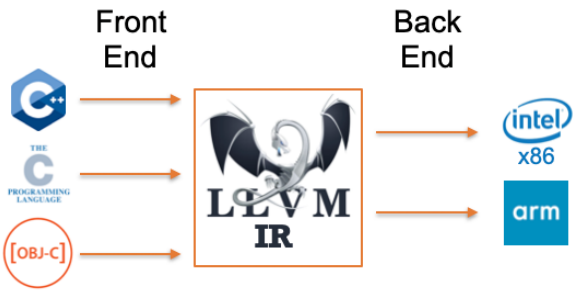
Quelle: http://www.davespace.co.uk/

Der eigentlich Befehl (auch Opcode gennant) befindet sich in diesem Fall an der Stelle von Bit Nr. 21 bis 24 und nimmt somit lediglich vier Bits und nicht 8 Bits wie bei x86 ein. Trotz der Länge von vier Bit liegt der Opcode nicht direkt in einem Block der zur Aufteilung für die Hexadezimale Darstellung gewählt wird, weshalb nur eine bedingte Übersetzung zwischen Hex-Wert und konkretem Befehl möglich ist.
Erwähnenswert ist zudem der Condition-Block, also Bedingungsblock am Anfang jeder Instruktion. Mit diesem ist es möglich einen Befehl nur dann auszuführen, wenn die vorgegebene Bedingung (zu sehen in den Condition Codes im Bild) erfüllt ist. Dies stellt einen wesentlichen Vorteil gegenüber der x86 Variante dar, welche Bedingte-Sprünge verwenden muss um im Code über die Instruktion an eine andere Stelle zu springen.
Da durch das Pipelining-Prinzip bereits vor der eigentlich Ausführung eines Befehls Vorbereitungen getroffen werden, um so die Ausführung zu beschleunigen, müssen diese Vorbereitungen am Ziel des Sprungs nachgeholt werden, was einen derartigen Sprung sehr zeitintensiv macht. Bei ARM können einzelne Befehle einfach ohne Sprung übergangen werden, wodurch die bereits vorangestellten Vorbereitungen für die kommenden Befehle nicht obsolet werden.

Hinweis:
Auf das Pipelining Prinzip wird bewusst nicht weiter eingegangen, da es für die Transformierung und damit das Projekt nur eine untergeordnete Rollte spielt.

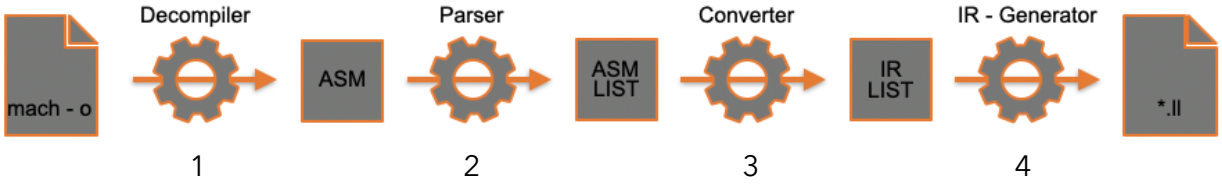
5. Transformation

Viele Ansätze, eines bereits für eine konkrete Architektur kompilierten Codes, auch für andere Architekturen lauffähig zu bekommen basieren auf der Emulation der Ursprungsarchitektur. So ist es beispielsweise möglich mit Hilfe von virtuellen Maschinen eine komplette Architektur inkl. Betriebssystem zu simulieren und somit architekturfremde Programme auf dem Gastsystem auszuführen. Neben der Tatsache, dass die entsprechende Emulationssoftware sowohl die gewünschte Gast- als auch Zielarchitektur unterstützen muss, ist es notwendig bei jedem erneuten Ausführen mehr Rechenleistung für die Emulation aufzuwenden.
Durch die einmalige und direkte Umwandlung einer ausführbaren Datei und des darin enthaltenen Maschinencodes, in den Maschinencode einer anderen Architektur würde eine native Bereitstellung des Programms ohne weitere Softwareabhängigkeit bei jeder Ausführung und ohne Leistungseinbußen ermöglichen.
Das dies möglich sein muss, legt zum einen die Tatsache nahe, dass Programme welche in den gängigen Programmiersprachen wie C, C++ o.Ä. mit den richtigen Compilern für nahezu jede Architektur kompiliert werden kann. Zum anderen unterstützt Apple mit seiner Rosetta Software ein Just in Time Transformation zwischen der vorher verwendeten x86 Architektur hin zur neuen ARM Architektur, auf welchem der hauseigene M1-Prozessor beruht.
Ein Ansatz der Transformation wäre die direkte Übersetzung einer Maschinensprache in eine andere. Dieser Ansatz hat jedoch gleich mehrerer Nachteile. Zum einen weißt die Methode große Schwächen im Bereich der Erweiterbarkeit zum Ausgangs und Ziel Architekturlauf. So ist es beispielsweise beim Hinzufügen einer neuen Ziel Architektur notwendig, für jeden Ausgangsarchitektur eine neue Transformation zu programmieren. Darüber hinaus müssten Optimierungen im Transformierten Maschinencode ebenfalls selbst im Programm vorgenommen werden, um Leistungseinbußen so gering wie möglich zu halten.
Eine durchaus elegantere Lösung stellt daher der Zwischenweg über eine universelle Assemblersprache dar, in welche der Maschinencode der Ausgangsarchitektur übersetzt wird. Im Anschluss wird dann von der universellen Sprache zur Zielarchitektur transformiert.
Hierdurch ist es beim Erweitern der Software um eine neue Ausgangssprache lediglich eine Übersetzung in die Zwischensprache für selbige zu entwickeln. Gleiches gilt für eine neu hinzugefügte Zielarchitektur.
Eine derartige Assembler-Sprache zu entwickeln würde den Rahmen der Projektarbeit sprengen. Erfreulicherweise existieren bereits einige Ansätze solcher universellen Sprachen wie die LLVM-IR (LLVM Immediate Representation), welche vom LLVM Compiler-Unterbau verwendet wird und als Zwischensprache und Ansatz meines Projektes dient.
Die LLVM-IR versucht alle Besonderheiten der Sprachen so abzudecken, das möglichst einfach von ihr in jede beliebige Zielarchitektur (für die es entsprechende Anbindungen gibt) übersetzt werden kann.
Hierfür verwendet der LLVM Compiler sogenannte Front-Ends, welche den Code von vielen Programmiersprachen wie C, Swift, Python usw. in die LLVM-IR überführen.
Die Back-Ends der LLVM-Toolchain stellen dann die Übersetzung in Maschinencode für konkrete Zielarchitekturen her.

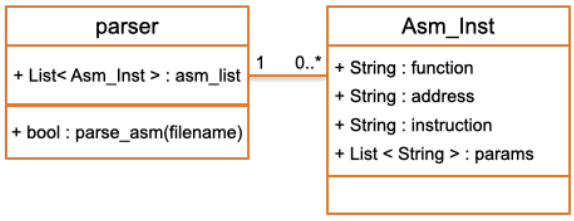


6. Projekt

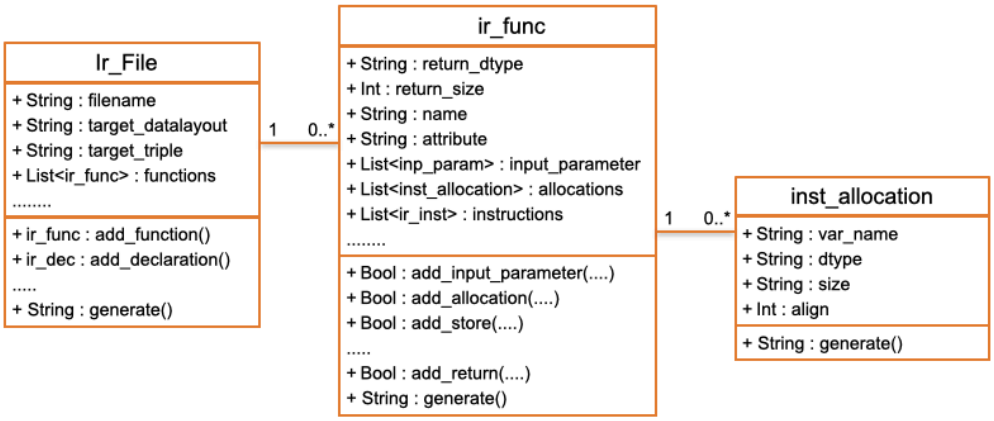
Die Idee des Projekts besteht im wesentlichen darin, ein LLVM-Front End für bereits kompilierten Maschinencode bereitzustellen, welches diesen in die LLVM-IR überführt. Als Ausgangsarchitektur wurde hierbei die arm64 Architektur gewählt.
Die folgende Grafik erläutert den generellen Ablauf des Programms:



1. Im ersten Schritt wird die ausführbare Datei dekompiliert. Dieser Vorgang wird zum größten Teil durch die Software OBJDUMP realisiert.
2. Das so entstandene Assembler-Listig wird im Anschluss mittels des Persers in eine für die Weiterverarbeitung geeignetere Form gebracht. Hierbei wird jede Instruktion als Instanz der Klasse Asm_Inst gebracht. Hierdurch kann im späteren Verlauf über die Liste mit Befehlen iteriert und mit den Instanzvariablen direkt auf Eigenschaften der einzelnen Instruktionen wie den Befehl oder die Operanden zugegriffen werden.

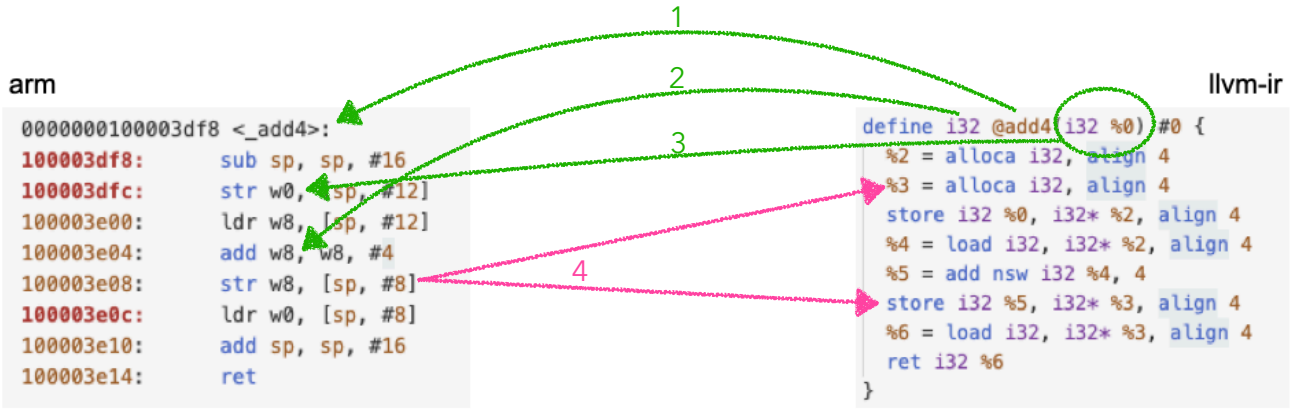


3. Im Anschluss Analysiert der Konverter, die so entstandene Liste und sucht direkte Übersetzungen in die LLVM-IR. Ein konkretes Beispiel folgt am Ende des Kapitels.
4. Als letztes wird mittels des IR-Generators ein Instanz der Klasse Ir-File erzeugt, zu welcher wiederum Instanzen der Klasse ir-func hinzugefügt werden können. Diese Funktionen wiederum bestehen aus einer Liste von LLVM-IR Befehlen (hier im Bild beispielhaft der Befehl „allocation“). Durch das Aufrufen der Klassenfunktion generate() wird ein String mit dem fertigen LLVM-IR Code zurückgeliefert, welcher dann exportiert werden kann..



Beispiel Transformation

Obwohl die LLVM-IR viele Ähnlichkeiten und teilweise sogar direkte Gemeinsamkeiten mit den gängigen reellen Assemblersprachen aufweist, gibt es doch einige Besonderheiten auf die geachtet werden muss. Einen markanten Unterschied stellt die unendlich Anzahl an verfügbaren Registern dar. Erst beim Überführen in den Maschinencode für eine konkrete Architektur wird entschieden, wie viele Werte tatsächlich in Register geschrieben werden können und für wie viele auf den Stack (wenn vorhanden) ausgewichen werden muss.
Bei der Konvertierung von arm64 in die LLVM-IR kann wie in dem folgenden Beispiel vorgegangen werden:



1. Funktionsname ermitteln:

Der Funktionsname kann direkt dem Label des dekompilierten ASM-Listings entnommen werden.

2. Rückgabetyp der Funktion

Der Rückgabetyp der Funktion kann über das Register, welches für den Rückgabewert reserviert ist, ermittelt werden. Entsprechend der ARM-Calling Convention ist hierfür indirekt das Register R8 reserviert. Dieses wird während der Ausführung der Funktion mit dem Ergebnis beschrieben. Am Ende der Funktion wird der Rückgabewert allerdings in das Register R0 überführt. Es handelt sich also lediglich um ein virtuelles Rückgaberegister, kann aber für die Ermittlung des Rückgabetyps der Funktion dennoch herangezogen werden.
Handelt es sich bei dem Register R8 um ein W-Register (also W8), so ist dieses für die Speicherung von 32Bit Integer Werten vorgesehen, wodurch der Rückgabetyp i32 entspräche. Würde es sich um ein X-Register handeln wäre es ein 64Bit Integer Wert.
V-Register beschreiben hingegen Gleitkomma Register und so weiter. Die entsprechenden Register-Benennungen können der ARM-Dokumentation entnommen werden weshalb ich hier nicht weiter auf diese eingehen möchte.

3. Funktionsparameter bestimmen

Ähnlich wie bei beim Rückgabetyp hilft auch hier ein Blick in die Calling-Convention. Für die Übergabeparameter von Funktionen sind die Register R0 bis R7 reserviert. Auch hier kann wieder der Datentyp des Registers, analog zur Beschreibung oben, ermittelt werden.
Sollte die Funktion mehr als acht Funktionsparameter aufweisen, werden diese vom Stack in entsprechende Register geladen.

4. Umwandlung einzelner Instruktionen

Für die Umwandlung konkreter Befehle verwendet meine Software eine Art Übersetzungstabelle. Als Beispiel verwende ich den Store Befehl, welcher bei ARM den Wert aus einem Register an eine bestimmte Adresse im Stack schreibt. Da die LLVM-IR keinen Stack besitzt sondern mit unendlich vielen Register arbeitet, wird die Stack-Adresse einfach durch ein neues Register ersetzt. Wird jedoch bei der LLVM-IR mit anderen Registern als mit den Registern der Funktionsparameter gearbeitet, muss zunächst Speicher allokiert werden. Daraus ergibt sich aus einem ARM Store Befehl zwei LLVM-IR Befehle.

7. Sonstiges

Wichtige Terminal Commands die für die Projektarbeit verwendet wurden

LLVM

- LLVM-IR Bitcode-Respresentation erzeugen:
  $ clang sum.c -emit-llvm -c -o sum.bc

- LLVM-IR Assembly-Representation erzeugen
  $ clang sum.c -emit-llvm -S -c -o sum.ll

- LLVM-IR Assembly to LLVM-IR Bitcode
  $ llvm-as sum.ll -o sum.bc

- LLVM-IR to Executeable
  $ clang sum.ll -o sum

- GCC Cross Compilation:
  gcc -target x86_64-apple-darwin19.6.0 -O0 test_m_arm.c -o test_m_x86

OBJDUMP

- Disasemle with platform native ASM
  $ objdump -D prog

- Disasemble with Intel ASM
  $ objdump -x86-asm-syntax=intel -d prog

Speicherplätze von APPLE SDK´s

Location of .h Files:

1: /Library/Developer/CommandLineTools/SDKs/MacOSX10.15.sdk/usr/include/stdio.h
3: /Library/Developer/CommandLineTools/SDKs/MacOSX11.3.sdk/usr/include/stdio.h

8. Quellen

LLVM-Front End Bauen
https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl01.html

Inside a excecuteable in Mac OS (macho)
https://adrummond.net/posts/macho

Calling Conventions x86 and arm
https://en.wikipedia.org/wiki/Calling_convention

Unterschiede ARM und INTEL
https://www.androidauthority.com/arm-vs-x86-key-differences-explained-568718/

Was ist Maschinensprache?
https://bmu-verlag.de/maschinensprache/

Apple muss Rosetta abschalten
https://t3n.de/news/hinweise-macos-113-rosetta-2-deaktivierung-1363324/

Arm Architecture Reference Manual
https://developer.arm.com/documentation/ddi0487/latest/

ARM Cond-Codes
http://www.davespace.co.uk/

Mac Life Benchtest
https://www.maclife.de/news/apple-silicon-trotz-x86-emulation-schneller-jeder-andere-mac-mac-life-100118026.html

LLVM / CLANG
https://clang.llvm.org

HEX to ASCII
https://www.rapidtables.com/convert/number/hex-to-ascii.html

Befehlssatzarchitektur:
https://de.wikipedia.org/wiki/Befehlssatzarchitektur

LLVM-Tutorial I
https://www.youtube.com/watch?v=m8G_S5LwlTo

Des weiteren wurden verwendet:
Vorlesungsvideos und Skript zur Vorlesung
IT-Sicherheit und Cyberarchitekturen an der UnibwM im Jahr 2020

Alle Links wurde zuletzt am 12.07.21 um 21.30Uhr geöffnet