Reverse Engineering for Beginners
Reverse Engineering für Einsteiger

(Understanding Assembly Language)

Why two titles? Read here: on page ix.

Dennis Yurichev

<first_name @ last_name . com>

©©©

© 2013-2020, Dennis Yurichev.

Diese Arbeit ist lizenziert unter der Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) Lizenz. Um eine Kopie dieser Lizenz zu lesen, besuchen Sie https://creativecommons.org/licenses/by-sa/4.0/.


Übersetzer gesucht!

Vielleicht möchten Sie mir bei der Übersetzung dieser Arbeit in andere Sprachen (außer Englisch und Russisch) helfen. Senden Sie mir einen übersetzten Textteil, egal wie kurz und ich arbeite ihn in den LATEX-Quellcode ein.

Hier lesen.

Geschwindigkeit ist nicht wichtig, denn im Endeffekt ist es ein Open-Source-Projekt. Ihr Name wird als Mitwirkender des Projekts erwähnt. Koreanisch, Chinesisch und Persisch sind für Verleger reserviert. Eine englische und russische Version mache ich selber, allerdings ist mein Englisch immer noch schrecklich. Ich bin also dankbar über alle Anmerkungen bezüglich der Grammatik und Ähnlichem. Auch mein Russisch ist teilweise fehlerhaft, also bin ich auch hier dankbar über Anmerkungen!

Also zögern Sie nicht mir zu schreiben: <first_name @ last_name . com> / <first_name . last_name @ gmail . com>.
Inhaltsverzeichnis (gekürzt)

1 Code-Muster ........................................... 1

2 Wichtige Grundlagen ................................. 408

3 Fortgeschrittenere Beispiele ....................... 412

4 Java .................................................... 413

5 Finden von wichtigen / interessanten Stellen im Code 418

6 Betriebssystem-spezifische Themen ............... 448

7 Tools ................................................... 504

8 Beispiele für das Reverse Engineering proprietärer Dateiformate 508

9 Dynamic Binary Instrumentation (DBI) ............ 510

10 Weitere Themen ..................................... 511

11 Bücher / Lesenswerte Blogs ....................... 519

12 Communities ........................................ 522

Nachwort ................................................ 524

Anhang .................................................... 526

Verwendete Abkürzungen ............................ 548

Glossar ................................................... 552

Index ..................................................... 554
Inhaltsverzeichnis

1 Code-Muster .................................................. 1
   1.1 Die Methode ............................................. 1
   1.2 Einige Grundlagen .................................... 1
      1.2.1 Eine kurze Einführung in die CPU ................. 1
      1.2.2 Zahlensysteme .................................. 3
   1.3 Leere Funktion .......................................... 5
      1.3.1 x86 ............................................... 5
      1.3.2 ARM ............................................. 5
      1.3.3 MIPS ............................................ 6
      1.3.4 Leere Funktionen in der Praxis ............... 6
   1.4 Die einfachste Funktion .............................. 7
      1.4.1 x86 ............................................... 7
      1.4.2 ARM ............................................. 7
      1.4.3 MIPS ............................................ 8
   1.5 Hallo, Welt! ............................................ 8
      1.5.1 x86 ............................................... 8
      1.5.2 x86-64 .......................................... 14
      1.5.3 ARM ............................................. 18
      1.5.4 MIPS ............................................ 24
      1.5.5 Fazit ............................................ 29
      1.5.6 Übungen ......................................... 29
   1.6 Stack .................................................... 29
      1.6.1 Warum wächst der Stack nach unten? ........... 29
      1.6.2 Für was wird der Stack benutzt? ................ 30
      1.6.3 Rückgabe Adresse der Funktion speichern ....... 30
      1.6.4 Ein typisches Stack Layout .................... 36
      1.6.5 Rauschen auf dem Stack ......................... 36
      1.6.6 Übungen ......................................... 41
   1.7 German text placeholder ............................. 41
      1.7.1 x86 ............................................... 41
      1.7.2 ARM ............................................. 49
      1.7.3 Fazit ............................................ 49
      1.7.4 Übrigens ......................................... 50
   1.8 scanf() .................................................. 50
      1.8.1 Ein einfaches Beispiel .......................... 50
      1.8.2 Häufiger Fehler ................................ 60
      1.8.3 Globale Variablen ............................... 61
      1.8.4 scanf() ......................................... 70
      1.8.5 German text placeholder ....................... 80
   1.9 Zugriff auf übergebene Argumente .................. 80
      1.9.1 x86 ............................................... 81
      1.9.2 x64 ............................................... 83
      1.9.3 ARM ............................................. 86
      1.9.4 MIPS ............................................ 89
   1.10 Pointer .................................................. 90
      1.10.1 Werte zurückgeben ............................. 90
      1.10.2 Eingabewerte vertauschen ................. 100
   1.11 GOTO Operator ....................................... 101
      1.11.1 Dead code .................................... 104
      1.11.2 German text placeholder ................... 105
   1.12 Bedingte Sprünge ................................... 105
      1.12.1 einfaches Beispiel ............................ 105
      1.12.2 Betrag berechnen .............................. 122
1.21.5 Verschachtelte structs .......................................................... 341
1.21.6 Bitfields in einem struct .................................................... 344
1.21.7 Übungen ............................................................................ 351
1.22 Unions ................................................................................... 351
  1.22.1 Pseudozufallszahlengenerator Beispiel .................................. 351
  1.22.2 Berechnung der Maschinengenauigkeit ................................ 351
  1.22.3 FSSCALE Ersatz ................................................................. 356
  1.22.4 Schnelle Berechnung der Quadratwurzel ......................... 357
1.23 64-Bit-Werte in 32-Bit-Umgebungen ........................................ 358
  1.23.1 Rückgabe von 64-Bit-Werten ............................................ 358
  1.23.2 Übergabe von Argumenten bei Addition und Subtraktion .... 359
  1.23.3 Multiplikation und Division ................................................ 362
  1.23.4 Verschiebung nach rechts .................................................. 365
  1.23.5 32-Bit-Werte in 64-Bit-Werte umwandeln ....................... 367
1.24 SIMD ..................................................................................... 367
  1.24.1 Vektorisierung .................................................................. 368
  1.24.2 SIMD strlen() Implementierung ........................................ 378
1.25 64 Bit ..................................................................................... 381
  1.25.1 x86-64 ............................................................................. 381
  1.25.2 ARM .............................................................................. 388
  1.25.3 Fließkomzzahlen ............................................................... 388
  1.25.4 Kritik an der 64-Bit-Architektur ....................................... 388
1.26 Arbeiten mit Fließkomzzahlen und SIMD .................................... 388
  1.26.1 Ein einfaches Beispiel ......................................................... 388
  1.26.2 Fließkomzzahlen als Argumente übergeben ....................... 396
  1.26.3 Beispiel mit Vergleich ....................................................... 397
  1.26.4 Berechnen der Maschinengenauigkeit: x64 und SIMD ........ 399
  1.26.5 Erneute Betrachtung des Beispiels zum Pseudozufallszahlengenerator .......................................................... 400
  1.26.6 Zusammenfassung ............................................................. 400
1.27 ARM-spezifische Details ............................................................. 401
  1.27.1 Zeichen (#) vor einer Zahl ................................................ 401
  1.27.2 Adressierungsmodi ............................................................. 401
  1.27.3 Laden einer Konstanten in ein Register .............................. 402
  1.27.4 Relocs in ARM64 ............................................................... 404
1.28 MIPS-spezifische Details ........................................................... 405
  1.28.1 Laden einer 32-Bit-Konstante in ein Register ................. 405
  1.28.2 Weitere Literatur über MIPS ............................................. 407

2 Wichtige Grundlagen .................................................................... 408
  2.1 German text placeholder .......................................................... 408
  2.1.1 Nutzen von IMUL anstatt MUL ........................................ 409
  2.1.2 Einige weitere Anmerkungen zum Zweierkomplement ........ 410

3 Fortgeschrittenere Beispiele ........................................................... 412
  3.1 strstr()-Beispiel .................................................................... 412

4 Java ............................................................................................ 413
  4.1 Java ......................................................................................... 413
    4.1.1 Einführung ........................................................................ 413
    4.1.2 Rückgabe eines Wertes ..................................................... 413
    4.1.3 Einfache Berechnungsfunktionen ...................................... 414
    4.1.4 JVM²-Speichermodell ...................................................... 414
    4.1.5 Einfache Funktionsaufrufe ................................................ 414
    4.1.6 Aufrufen von beep() ......................................................... 414
    4.1.7 Linearer Kongruenzgenerator PRNG³ ............................... 415
    4.1.8 Bedingte Sprünge .............................................................. 415
    4.1.9 Argumente übergeben ....................................................... 415
    4.1.10 Bit-Felder ........................................................................ 415
    4.1.11 Schleifen ......................................................................... 415
    4.1.12 switch() ......................................................................... 415
    4.1.13 Arrays ............................................................................. 415
    4.1.14 Zeichenketten ................................................................. 415

²Java Virtual Machine
³Pseudozufallszahlen-Generator
5 Finden von wichtigen / interessanten Stellen im Code 418
5.1 Ausführbare Dateien Identifizieren .......................................................... 419
  5.1.1 Microsoft Visual C++ ........................................................................... 419
  5.1.2 GCC .................................................................................................. 419
  5.1.3 Intel Fortran ...................................................................................... 419
  5.1.4 Watcom, OpenWatcom ..................................................................... 419
  5.1.5 Borland ............................................................................................. 420
  5.1.6 Other known DLLs ........................................................................... 421
5.2 Kommunikation mit der außen Welt (Funktion Level) .............................. 421
5.3 Kommunikation mit der Auß en Welt (Win32) .......................................... 421
  5.3.1 Oft benutzte Funktionen in der Windows API .................................. 422
  5.3.2 Verlängerung der Testphase .............................................................. 422
  5.3.3 Entfernen nerviger Dialog Boxen ....................................................... 422
  5.3.4 tracer: Alle Funktionen innerhalb eines bestimmten Modules abfangen . 422
5.4 Strings ...................................................................................................... 423
  5.4.1 Text strings ...................................................................................... 423
  5.4.2 Strings in Binär finden ...................................................................... 428
  5.4.3 Error/debug Nachrichten ................................................................. 429
  5.4.4 Verdächtige magic strings ................................................................. 429
5.5 assert() Aufrufe ....................................................................................... 430
5.6 Konstanten ............................................................................................... 430
  5.6.1 Magic numbers .................................................................................. 431
  5.6.2 Spezifische Konstanten .................................................................... 433
  5.6.3 Nach Konstanten suchen ................................................................. 433
5.7 Die richtigen Instruktionen finden ............................................................ 433
5.8 Verdächtige Code muster ......................................................................... 435
  5.8.1 XOR Instruktionen .......................................................................... 435
  5.8.2 Hand geschriebener Assembler code .............................................. 435
5.9 Using magic numbers while tracing .......................................................... 436
5.10 Schleifen .................................................................................................. 437
  5.10.1 Muster in Binärdatein finden ............................................................. 438
  5.10.2 Memory „snapshots“ comparing ...................................................... 445
5.11 Andere Dinge ........................................................................................... 446
  5.11.1 Die Idee ........................................................................................... 446
  5.11.2 Anordnung von Funktionen in Binär Code ..................................... 446
  5.11.3 kleine Funktionen ......................................................................... 447
  5.11.4 C++ ............................................................................................... 447

6 Betriebssystem-spezifische Themen 448
6.1 Methoden zur Argumentenübergabe (Aufrufkonventionen) .................. 448
  6.1.1 cdecl ............................................................................................... 448
  6.1.2 stdcall ............................................................................................ 448
  6.1.3fastcall ............................................................................................ 449
  6.1.4 thiscall ............................................................................................ 451
  6.1.5 x86-64 ............................................................................................ 451
  6.1.6 Rückgabewerte von float- und double-Typen .................................. 454
  6.1.7 Verändern von Argumenten ............................................................. 454
  6.1.8 Einen Zeiger auf ein Argument verarbeiten ..................................... 455
6.2 lokaler Thread-Speicher ........................................................................... 456
  6.2.1 Nochmals Linearer Kongruenzgenerator ...................................... 457
6.3 Systemaufrufe ........................................................................................... 461
  6.3.1 Linux ............................................................................................... 462
  6.3.2 Windows .......................................................................................... 462
6.4 Linux .......................................................................................................... 462
  6.4.1 Positionsabhängiger Code ............................................................... 462
  6.4.2 LD_PRELOAD-Hack in Linux .......................................................... 465
6.5 Windows NT .............................................................................................. 467
  6.5.1 CRT (win32) .................................................................................... 467
  6.5.2 Win32 PE ....................................................................................... 471
  6.5.3 Windows SEH .................................................................................. 479
7 Tools

7.1 Binäre Analyse .................................................................................. 504
  7.1.1 Disassembler ........................................................................ 504
  7.1.2 Decompiler ........................................................................ 505
  7.1.3 Vergleichen von Patches ........................................................ 505
7.2 Live-Analyse .................................................................................... 505
  7.2.1 Debugger ........................................................................... 505
  7.2.2 Tracen von Bibliotheksaufrufen ........................................... 505
  7.2.3 Tracen von Systemaufrufe ................................................... 505
  7.2.4 Netzwerk-Analyse (Sniffing) .................................................. 506
  7.2.5 Sysinternals ....................................................................... 506
  7.2.6 Valgrind ........................................................................... 506
  7.2.7 Emulatoren ...................................................................... 506
7.3 Andere Tools ................................................................................ 506
  7.3.1 Rechner ............................................................................ 507
7.4 Fehlt etwas? .................................................................................. 507

8 Beispiele für das Reverse Engineering proprietärer Dateiformate ..... 508

8.1 German text placeholder .................................................................. 508
  8.1.1 Einfachste XOR-Verschlüsselung überhaupt ......................... 508
  8.2 German text placeholder ............................................................ 509

9 Dynamic Binary Instrumentation (DBI) ............................................ 510

10 Weitere Themen .............................................................................. 511
  10.1 Patchen von ausführbaren Dateien .......................................... 511
    10.1.1 x86-Code .................................................................... 511
  10.2 Statistiken von Funktionsargumenten ..................................... 511
  10.3 Intrinsische Compiler-Funktionen ............................................. 512
  10.4 Compiler Anomalien ............................................................... 512
    10.4.1 Oracle RDBMS 11.2 und Intel C++ 10.1 ......................... 512
    10.4.2 MSVC 6.0 .................................................................... 513
    10.4.3 Zusammenfassung .......................................................... 513
  10.5 Itanium .................................................................................. 514
  10.6 8086-Speichermodell .............................................................. 516
  10.7 Basic Block Reordering ............................................................ 517
    10.7.1 Profile-guided Optimization ............................................ 517

11 Bücher / Lesenswerte Blogs ............................................................ 519
  11.1 Bücher und andere Materialien .................................................. 519
    11.1.1 Reverse Engineering ....................................................... 519
    11.1.2 Windows .................................................................... 519
    11.1.3 C/C++ ........................................................................ 519
    11.1.4 x86 / x86-64 ................................................................. 520
    11.1.5 ARM .......................................................................... 520
    11.1.6 Assembler .................................................................. 520
    11.1.7 Java ........................................................................... 520
    11.1.8 UNIX ......................................................................... 520
    11.1.9 Programmierung Allgemein ........................................... 520
    11.1.10 Kryptografie ............................................................... 520

12 Communities .................................................................................. 522

Nachwort ............................................................................................ 524
  12.1 Fragen? .................................................................................. 524

Anhang ................................................................................................ 526
  .1 x86 ......................................................................................... 526
    .1.1 Terminologie .................................................................... 526
    .1.2 Mehrzweckregister ........................................................... 526
    .1.3 FPU German text placeholder ............................................ 530
Verwendete Abkürzungen

Glossar

Index
Vorwort

Warum zwei Titel?


Die beiden Bücher unterscheiden sich lediglich im Titel, dem Dateinamen (UAL-XX.pdf beziehungsweise RE4B-XX.pdf), URL und ein paar der einleitenden Seiten.

Über Reverse Engineering

Es gibt verschiedene verbreitete Interpretationen des Begriffs Reverse Engineering:
1) Reverse Engineering von Software: Rückgewinnung des Quellcodes bereits kompilierter Programme;
2) Das Erfassen von 3D Strukturen und die digitalen Manipulationen die zur Duplizierung notwendig sind;
3) Nachbilden von DBMS-Strukturen.
Dieses Buch behandelt die erste Interpretation.

Voraussetzungen

Grundlegende Kenntnisse der Programmiersprache C. Empfohlene Literatur: 11.1.3 on page 519.

Übungen und Aufgaben

...befinden sich nun alle auf der Website: http://challenges.re.

Lob für

https://beginners.re/#praise.

Danksagung

Für das geduldige Beantworten aller meiner Fragen: Slava „Avid“ Kazakov, SkullC0DEr.

---

4 https://news.ycombinator.com/item?id=17549050
5 Database Management Systems
6 goto-vlad@github
7 https://github.com/pixjuan
8 https://github.com/73696e65
9 https://github.com/TheRenaissance
Für die Übersetzung des Buchs ins Vereinfachte Chinesisch: Antiy Labs (antiy.cn), Archer.
Für die Übersetzung des Buchs ins Koreanische: Byungho Min.
Für die Übersetzung des Buchs ins Niederländische: Cedric Sambre (AKA Midas).
Für die Übersetzung des Buchs ins Spanische: Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames.
Für die Übersetzung des Buchs ins Portugiesische: Thales Stevan de A. Gois, Diogo Mussi, Luiz Filipe.
Für die Übersetzung des Buchs ins Französische: Florent Besnard, Marc Remy, Baudouin Landais, Téo Dacquet, BlueSkeye@GitHub.
Für die Übersetzung des Buchs ins Deutsche: Dennis Siekmeier, Julius Angres, Dirk Loser, Clemens Tamme.
Für die Übersetzung des Buchs ins Polnische: Kateryna Rozanova, Aleksander Mistrzewicz, Wiktoria Lewicka.
Für die Übersetzung des Buchs ins Japanische: shmz@github.
Vasil Kolev, der unglaublich viel Arbeit in die Korrektur vieler Fehler investiert hat.
Danke auch an alle, die auf github.com Anmerkungen und Korrekturen eingebracht haben.
Es wurden viele LaTeX-Pakete genutzt: Vielen Dank an deren Autoren.

Donors
Dank an diejenigen, die mich während der Zeit in der ich wichtige Teile des Buchs geschrieben habe unterstützt haben:
2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar ($50), James Truscott ($4.5), Luis Rocha ($63), Joris van de Vis ($127), Richard S Shultz ($20), Jang Minchang ($20), Shade Atlas (5 AUD), Yao Xiao ($10), Pawel Szczur (40 CHF), Justin Simms ($20), Shawn the Rock ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararaoa ($50), Philippe Teuwen ($4), Martin Haeberli ($10), Victor Cazacov ($5), Tobias Sturzenegger (10 CHF), Sonny Thai ($15), Bayna AlZaabi ($75), Redfive B.V. (€25), Joona Oskari Heikkila (€5), Marshall Bishop ($50), Nicolas Werner (€12), Jeremy Brown ($100), Alexandre Borges ($25), Vladimir Dikovski ($50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent (€30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe (€14), Maxim Dyakonov (€3), Sebastian Aguiler (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov (€100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu ($10), Tudor Azoitei ($40), Z0vsky (€10), Yu Dai ($10), Anonymous (€15), Vladislav Chelnokov ($25), Renad Noveljic ($50), Ryan Smith (€50), Andreas Schommer (€5), Nikolay Gavrilov ($300).
Vielen Dank an alle Spender!

Mini-FAQ
F: Was sind die Voraussetzungen die der Leser dieses Buchs erfüllen sollte?
A: Grundlagenwissen der Programmiersprachen C und / oder C++ sind wünschenswert.

---

10 https://github.com/pinkrab
11 https://github.com/paolostivanin
12 https://github.com/Internaut401
13 https://github.com/besnardf
14 https://github.com/mremy
15 https://github.com/T30rix
16 https://github.com/BlueSkeye
17 https://github.com/DSiekmeier
18 https://github.com/JAngres
19 https://github.com/PolymathMonkey
20 https://github.com/shmz
21 https://vasil.ludost.net/
F: Sollte ich wirklich x86/x64/ARM und MIPS auf einmal lernen? Ist das nicht zuviel?
A: Anfänger können erstmal nur über x86/x64 lesen und den ARM- und MIPS-Teil überspringen oder überfliegen.

F: Kann ich eine russische oder englische Version als Druckausgabe kaufen?

F: Gibt es eine EPUB- oder MOBI-Version?
A: Dieses Buch ist in hohem Maße abhängig von TeX- / LATEX-spezifischen Techniken, was das Konvertieren zu HTML schwierig macht (EPUB und MOBI basieren auf HTML).

F: Was sollte ich heutzutage noch Assembler lernen?
A: Falls Sie kein BS\textsuperscript{22}-Entwickler sind, werden Sie vermutlich nie in Assembler programmieren müssen — aktuelle Compiler (2010 und später) können sehr viel besser optimieren als Menschen\textsuperscript{23}.
Auch sind aktuelle CPU\textsuperscript{24}s sehr komplexe Komponenten und Wissen über Assembler hilft nicht wirklich um die Interna zu verstehen.


F: Ich habe in der PDF-Datei auf einen Link geklickt. Wie komme ich zurück?

F: Darf ich dieses Buch drucken / für Lehrzwecke benutzen?
A: Selbstverständlich! Das ist der Grund warum es unter der Creative Commons Lizenz (CC BY-SA 4.0) veröffentlicht wird.


F: Wie kann man einen Job im Bereich des Reverse Engineering bekommen?
A: Von Zeit zu Zeit gibt es Threads zu Jobangeboten auf \textsuperscript{25}. Versuchen Sie es dort einmal.
Ein ähnlicher Job-Thread ist unter„netsec“ subreddit zu finden.

F: Ich habe eine Frage...
A: Senden Sie sie mir per E-Mail (<first_name @ last_name . com> / <first_name . last_name @ gmail . com>).

Über die koreanische Übersetzung
Es ist jetzt unter dieser Webseite verfügbar.
Wenn Sie also ein echtes Buch in Ihrem Buchregal auf koreanisch haben und mich bei meiner Arbeit unterstützen wollen, können Sie das Buch nun kaufen.

\textsuperscript{22}Betriebssystem
\textsuperscript{23}Ein lesenswerter Artikel zu diesem Thema: [Agner Fog, The microarchitecture of Intel, AMD and VIA CPUs, (2016)]
\textsuperscript{24}Central Processing Unit
\textsuperscript{25}reddit.com/r/ReverseEngineering/
Über die persische Übersetzung (Farsi)

In 2016 wurde das Buch von Mohsen Mostafa Jokar übersetzt, der in der iranischen Community auch für die Übersetzung des Radare Handbuchs bekannt ist). Es ist auf der Homepage des Verlegers (Pendare Pars) verfügbar.

Hier ist ein Link zu einem 40-seitigen Auszug: https://beginners.re/farsi.pdf.


Über die chinesische Übersetzung

Im April 2017 wurde die chinesische Übersetzung von Chinese PTPress fertiggestellt, bei denen auch das Copyright liegt.


Der Hauptübersetzer ist Archer, dem der Autor sehr viel verdankt. Archer war extrem akribisch (im positiven Sinne) und meldete den Großteil der bekannten Fehler, was für Literatur wie dieses Buch extrem wichtig ist. Der Autor empfiehlt diesen Service jedem anderen Autor!


---

27 http://goo.gl/2Tzx0H
Kapitel 1

Code-Muster

1.1 Die Methode

Als der Autor dieses Buches zunächst C und später C++ lernte, schrieb er kleine Codestücke, kompilierte sie und untersuchte anschließend den Assembler-Code des Compilers. Dies machte es sehr einfach zu verstehen, was in dem Code den er geschrieben hatte passierte. Er hat dies so oft getan, dass der Zusammenhang zwischen dem C/C++-Code und dem was der Compiler daraus macht tief in seinem Verstand verankert ist. So ist es einfach sich schnell einen Überblick über das Aussehen und die Funktion einer C-Quelle zu verschaffen. Vielleicht ist diese Vorgehensweise auch für andere hilfreich.

Übrigens gibt es eine hervorragende Webseite auf der dasselbe mit verschiedenen Compilern getan werden kann, anstatt diese zu installieren. Auch diese kann genutzt werden: https://godbolt.org/.

Übungen


Optimierungsstufen und Debug-Informationen


1.2 Einige Grundlagen

1.2.1 Eine kurze Einführung in die CPU

Die CPU ist die Komponente, die den Maschinencode ausführt aus dem ein Programm besteht.

1 Tatsächlich tut er dies immer noch wenn er einen bestimmten Code-Teil nicht versteht.
Ein kurzes Glossar:


**Maschinencode**: Code den die CPU direkt verarbeitet. Jeder Befehl ist in der Regel durch ein paar Byte kodiert.

**Assembler-Sprache**: Mnemonics und einige Makro-ähnliche Erweiterungen um das Leben der Programmierer zu erleichtern.

**CPU-Register**: Jede CPU hat eine feste Anzahl von Mehrzweck-Registern (GPR³). ≈ 8 bei x86, ≈ 16 bei x86-64, ≈ 16 bei ARM. Die einfachste Möglichkeit um Register zu verstehen, ist sie als typlose, temporäre Variable zu betrachten. Stellen Sie sich vor Sie würden mit einer Hochsprache arbeiten und könnten nur acht 32-Bit (oder 64-Bit) Variablen nutzen. Dennoch ist damit eine Menge möglich!

Man kann sich jetzt fragen, warum die Unterscheidung zwischen Maschinencode und einer Hochsprache notwendig ist. Die Antwort liegt einfach in der Tatsache, dass Menschen und CPUs nicht gleich sind—Es ist sehr viel einfacher für Menschen eine Programmiersprache wie C/C++, Java, Python, usw. zu lesen. Für eine CPU jedoch ist es einfacher eine geringere Abstraktion zu verarbeiten.

Vielleicht wäre es möglich eine CPU zu entwickeln die direkt eine Hochsprache ausführen kann, diese wäre aber sehr viel komplexer als es heute der Fall ist. In ähnlicher Weise ist es für Menschen äußerst unkomfortabel in einer Assembler-Sprache zu programmieren. Diese ist sehr hardwarenah und kaum zu realisieren ohne schwer zu findende Fehler zu machen. Das Programm, welches die Hochsprache in Assembler konvertiert nennt sich Compiler oder Übersetzer.

Einfache Anmerkungen zu den verschiedenen ISAs


---
²Instruction Set Architecture  
³General Purpose Registers  
⁴Reduced Instruction Set Computing  
⁵Befehle mit fester Länge sind übrigens so praktisch weil die nächste (oder vorherige) Befehlsadresse ohne großen Aufwand berechnet werden kann. Diese Eigenschaft wird im Kapitel switch() diskutiert.  
⁶zum Beispiel MOV/PUSH/CALL/Jcc
1.2.2 Zahlensysteme

Nowadays octal numbers seem to be used for exactly one purpose—file permissions on POSIX systems—but hexadecimal numbers are widely used to emphasize the bit pattern of a number over its numeric value.

Alan A. A. Donovan, Brian W. Kernighan — The Go Programming Language

Menschen sind an das Dezimal-System gewöhnt, möglicherweise weil sie zehn Finger haben. Trotzdem hat die Zahl 10 keine besondere Bedeutung in der Wissenschaft und Mathematik. Das natürliche Zahlensystem in der Digitaltechnik ist binär: 0 für die Abwesenheit und 1 für die Anwesenheit von Strom in einer Leitung. Die binäre 10 ist 2 im Dezimalsystem; Die binäre 100 ist 4 im Dezimalsystem und so weiter.


Wichtige Dinge zum Merken:

1. *Nummer* ist eine Nummer, während *Ziffer* in der Regel eine einzelne Zahl ist;
2. Eine Zahl ändert sich nicht beim Konvertieren in ein anderes Zahlensystem: nur die Darstellung ist anders.

Wie konvertiert man eine Zahl von einer Basis in eine andere?


Wofür steht 1234?

\[
10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4 = 1234 \text{ oder } 1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 1 = 1234
\]

Das Gleiche gilt für binäre Zahlen, nur zur Basis 2 statt 10. Wofür steht 0b101011?

\[
2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43 \text{ oder } 32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43
\]

Stellenwertsysteme können den Additionssystemen wie den römischen Ziffern gegenübergestellt werden. Möglicherweise hat die Menschheit zu Stellenwertsystemen gewechselt, weil diese für einfache Operationen (Addition, Multiplikation, usw.) per Hand oder auf Papier einfacher zu verwenden sind.

Tatsächlich können binäre Zahlen genauso addiert, subtrahiert und so weiter werden wie man es in der Schule gelernt hat, es stehen allerdings nur zwei Ziffern zur Verfügung.

Binärzahlen sind sperrig, wenn sie im Quellcode oder Speicherauszügen auftreten. Hier bietet sich die Verwendung des Hexadezimalsystems an. Die Basis besteht hier aus den Ziffern 0..9 und den sechs lateinischen Buchstaben A..F. Jede hexadezimale Ziffer besteht aus vier Bits oder vier binären Ziffern, was die Konvertierung zwischen Hexadezimal und Binär, selbst im Kopf, sehr einfach macht.

<table>
<thead>
<tr>
<th>Hexadezimal</th>
<th>Binär</th>
<th>Dezimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
<td>13</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
<td>14</td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
<td>15</td>
</tr>
</tbody>
</table>

Woran sieht man welche Basis gerade verwendet wird?

Dezimalzahlen werden für gewöhnlich ohne Zusatz geschrieben, zum Beispiel 1234. Einige Assembler betonen die Basis 10 jedoch mit dem Zusatz "d": 1234d.

Binärzahlen sind manchmal mit dem Präfix "0b" gekennzeichnet: 0b10011011 (GCC hat eine Nicht-standardisierte Erweiterung hierfür). Ein weiterer Weg ist der "b" Suffix, zum Beispiel: 10011011b. In diesem Buch wird versucht durchgängig die "0b"-Präfix-Variante zu benutzen.

Hexadezimalzahlen werden in C/C++ und anderen Hochsprachen mit dem "0x"-Präfix versehen: 0x1234ABCD, oder sie haben einen "h"-Suffix: 1234ABCDh - dies ist eine weit verbreitete Variante in Assembler und Debuggern. Wenn die Zahl mit A..F startet, muss eine 0 davor geschrieben werden: 0ABCDEFh. In diesem Buch wird versucht durchgängig die "0x"-Präfix-Variante zu benutzen.


**Oktalsystem**

Dies ist ein weiteres Zahlensystem, welches in der Computerprogrammierung sehr verbreitet ist: die 8 Ziffern (0..7) sind jeweils drei Bits zugeordnet. Dies macht das Konvertieren in andere Zahlensysteme sehr einfach. Das Oktalsystem wurde fast überall ersetzt, dennoch gibt es, überraschenderweise, ein *NIX-Tool, welches häufig genutzt wird und eine Oktalzahl als Aufrufparameter hat: chmod.

Wie viele *NIX-Nutzer wissen, ist der Aufrufparameter von chmod eine Zahl mit drei Ziffern. Die erste beschreibt die Rechte des Besitzers einer Datei, die zweite der Gruppe zu der die Datei gehört und die dritte ist für alle anderen. Jede Ziffer kann in ihrer binären Form repräsentiert werden:

<table>
<thead>
<tr>
<th>Dezimal</th>
<th>Binär</th>
<th>Bedeutung</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>111</td>
<td>rwx</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>rw-</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>r-x</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>r--</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>-wx</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>-w-</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>--x</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>---</td>
</tr>
</tbody>
</table>

Jedes Bit wird also abgebildet auf die Flags: lesen/schreiben/ausführen.


Die Oktalzahl 644 ist in binärer Form 110100100, oder (in Dreierbitgruppen) 110 100 100.

Man erkennt nun sehr schön, dass jedes dieser Dreiergruppen die Rechte für Besitzer/Gruppe/Andere beschreibt: zuerst rw-, dann r-- und zuletzt nochmals r--.


Das Oktalsystem wird von allen standardkonformen C/C++-Compilern unterstützt. Dies kann gelegentlich zu Verwirrung führen, weil Oktalzahlen mit einer vorangestellten Null gekennzeichnet werden, also zum Beispiel 0377, was 255 in der Dezimalschreibweise entspricht. Ein versehentlicher Schreibfehler wie "9" statt "9" wird der Compiler erkennen, weil die 9 keine Ziffer des Oktalsystems ist. GCC meldet einen Fehler wie:

```
error: invalid digit "9" in octal constant.
```

---

8 GNU Compiler Collection
Teilbarkeit
Wenn Sie eine Dezimalzahl wie 120 sehen, ist schnell ersichtlich, dass diese durch 10 teilbar ist, weil die letzte Ziffer eine Null ist. Genauso ist 123400 durch 100 teilbar, weil die letzten zwei Ziffern Nullen sind.
Auf die gleiche Weise ist die Hexadezimalzahl 0x1230 durch 0x10 (oder 16) und 0x123000 durch 0x1000 (oder 4096) teilbar.
Die Binärzahl 0b1000101000 ist durch 0b1000 (8) teilbar und so weiter. Diese Eigenschaft kann häufig dazu genutzt werden um schnell herauszufinden ob ein Speicherinhalt zu einer bestimmten Grenze aufgefüllt ist. Beispielsweise starten Sektionen in PE-Dateien fast immer an Adressen die in hexadezimaler Schreibweise mit drei Nullen enden: 0x41000, 0x10001000, usw. Der Grund dafür ist, dass fast alle diese Sektionen an Grenzen ausgerichtet sind, die Vielfache von 0x1000 (4096) Byte sind.

Langzahlarithmetik und Basis
Langzahlarithmetik nutzt sehr große Zahlen die in mehreren Bytes gespeichert sein können. Der öffentliche und private Schlüssel im RSA-Algorithmus beispielsweise benötigt 4096 Bit und mehr.

Aussprache

Fließkommdazahlen
Um Fließkommdazahlen von Integer unterscheiden zu können, wird ihnen oft eine “.0” angehängt, zum Beispiel 0.0, 123.0, und so weiter.

1.3 Leere Funktion
Die denkbar einfachste Funktion ist sicher eine die nichts macht:

Listing 1.1: C/C++-Code

```c
void f()
{
    return;
};
```

Kompilieren wir diesen Quellcode!

1.3.1 x86
Nachfolgende die Ausgabe die sowohl der optimierende GCC als auch der MSVC-Compiler auf einer x86-Plattform produziert:

Listing 1.2: German text placeholder GCC/MSVC (German text placeholder)

```c
f:
  ret
```

Es gibt lediglich eine Anweisung RET, welche die Ausführung zurück an caller übergibt.

1.3.2 ARM

Listing 1.3: German text placeholder Keil 6/2013 (German text placeholder) German text placeholder

```c
f PROC
BX lr
ENDP
```

Die Rücksprungadresse ist im ARM-ISA nicht auf dem lokalen Stack gesichert sondern im Link-Register. Die BX LR-Anweisung führt dazu, dass die an diese Adresse gesprungen wird—-also die Ausführung wieder an den caller übergeben wird.

10Portable Executable
1.3.3 MIPS
Es gibt zwei Namenskonventionen für Register in der MIPS-Welt: durch Zahlen (von $0 bis $31) oder Pseudonamen ($V0, $A0, usw.).

Die Assembler-Ausgabe von GCC unten listet die Register mit Nummern auf:

<table>
<thead>
<tr>
<th>j</th>
<th>$31</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td></td>
</tr>
</tbody>
</table>

...während IDA\(^\text{11}\) Pseudonamen nutzt:

<table>
<thead>
<tr>
<th>j</th>
<th>$ra</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td></td>
</tr>
</tbody>
</table>

Die erste Anweisung ist die Sprunganweisung die die Ausführung durch Springen an die Adresse in Register $31 (oder $RA) wieder an den caller übergibt.

Dies ist das Register analog zu LR\(^\text{12}\) in ARM.

Die zweite Anweisung ist NOP\(^\text{13}\) und macht gar nichts; sie kann hier erst mal ignoriert werden.

Eine Anmerkung zu MIPS-Anweisungen und Registernamen

Register und Anweisungsnamen sind in Bezug auf MIPS traditionellerweise klein geschrieben. Um einen einheitlichen Stil zu haben, werden in diesem Buch jedoch Großbuchstaben genutzt. Dies gilt für alle ISAs die in diesem Buch besprochen werden.

1.3.4 Leere Funktionen in der Praxis

Abgesehen von der Tatsache, dass leere Funktionen nutzlos sind, begegnet man ihnen in Low-Level-Code häufiger.

Zum Einen sind Debugging-Funktionen wie die Folgende recht verbreitet:

void dbg_print (const char *fmt, ...)
{
    #ifdef _DEBUG
        // open log file
        // write to log file
        // close log file
    #endif

    void some_function()
    {
        ...

        dbg_print ("we did something\n");

        ...
    }
}

Bei einem Nicht-Debug-Build (d.h. “Release”), ist _DEBUG nicht definiert. Die Funktion dbg_print() ist, obwohl sie bei der Ausführung aufgerufen wird, ohne Inhalt.

Eine verbreitete Art des Software-Schutzes ist es verschiedene Build-Versionen zu erstellen: eine für legale Kunden und eine Demo-Version. Der Letzteren können zum Beispiel wichtige Funktionen fehlen, wie folgt:

\(^\text{11}\) Interaktiver Disassembler und Debugger entwickelt von Hex-Rays
\(^\text{12}\) Link Register
\(^\text{13}\) No Operation

### 1.4 Die einfachste Funktion

Die einfachste, mögliche Funktion ist vermutlich eine, die lediglich einen konstanten Wert zurückgibt. Hier ist sie:

```c
int f()
{
    return 123;
}
```

Und nun in kompiliierter Version!

#### 1.4.1 x86

Nachfolgend das, was sowohl der optimierende GCC- als auch MSVC-Compiler auf einer x86-Plattform erzeugt:

```assembly
f:
    mov eax, 123
    ret
```

Es gibt zwei Anweisungen: die erste platziert den Wert 123 in das EAX-Register, welches per Konvention als Speicherplatz für den Rückabewert genutzt wird. Die zweite ist RET, die die Ausführung wieder an die aufrufende Funktion übergibt. Diese wird das Ergebnis vom EAX-Register übernehmen.

#### 1.4.2 ARM

Auf der ARM-Plattform gibt es einige kleine Unterschiede:

```assembly
f PROC
  mov r0, #0x7b ; 123
  bx lr
ENDP
```

ARM nutzt das Register R0 für das Speichern des Rückgabewerts der Funktion. Also wird in diesem Beispiel der Wert 123 dorthin kopiert. Die Rücksprungadresse wird nicht auf dem lokalen Stack sondern im Link-Register gespeichert. Die Anweisung BX LR führt dazu, dass die Ausführung an dieser Stelle fortgeführt wird. In diesem Fall wird also die Kontrolle wieder an die aufrufende Funktion übergeben.

Erwähnenswert ist der irreführende Name der MOV-Anweisung sowohl beim x86- als auch ARM-Befehlssatz: die Daten werden nicht verschoben sondern kopiert.
### 1.4.3 MIPS

Es gibt zwei verschiedene Konventionen bei der Benamung von Registern in der MIPS-Welt: mit einer Nummer (von $0 bis $31) oder mit einem Pseudonamen ($V0, $A0, usw.).

GCC benamt in der Ausgabe die Register mit Nummern:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>$31</td>
</tr>
<tr>
<td>li</td>
<td>$2,123</td>
</tr>
</tbody>
</table>

...während IDA Pseudonamen verwendet:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>jr</td>
<td>$ra</td>
</tr>
<tr>
<td>li</td>
<td>$v0, 0x7B</td>
</tr>
</tbody>
</table>

Das $2 (oder $V0)-Register wird zum Speichern des Rückgabewerts genutzt. LI steht für “Load Immediate” und ist das MIPS-Äquivalent zu MOV.

Die anderen Anweisungen sind Sprungbefehle (J oder JR), die die Ausführung wieder an die aufrufende Funktion übergeben, indem an die Adresse gesprungen wird, die im $31 (oder $RA)-Register.

Dieses Register ist analog zum LR in der ARM-Architektur.

Möglicherweise wundert man sich warum die Positionen der Lade- (LI) und Sprunganweisung (J oder JR) vertauscht sind. Dies geschieht durch ein RISC-Feature das “branch delay slot” genannt wird.

Die Begründung liegt in der Eigenart einiger RISC-Befehlssets die hier jedoch nicht so wichtig ist. Man sollte aber im Hinterkopf behalten, dass bei MIPS die Anweisung nach dem Sprungbefehl noch vor dieser ausgeführt wird.

Als Konsequenz sind Verzweigungsbefehle immer mit der Anweisung vertauscht, die zuvor ausgeführt werden muss.

#### Anmerkungen zu MIPS-Anweisungen / Registernamen


### 1.5 Hallo, Welt!


```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

#### 1.5.1 x86

**MSVC**

Das Beispiel wird jezt in MSVC 2010 kompiliert:

```
cl 1.cpp /Fa1.asm
```

(Die /Fa-Option weist den Compiler an, Assembler-Code auszugeben.)
MSVC erstellt Assembler-Code im Intel-Syntax. Der Unterschied zum AT&T-Syntax wird später in **1.5.1 on page 11** behandelt.

Der Compiler generiert die Datei `1.obj`, die anschließend zu `1.exe` gelinkt wird. In diesem Fall besteht die Datei aus zwei Segmenten: CONST (für konstante Daten) und _TEXT (für Quellcode).


Aus diesem Grund kann das Beispiel auch wie folgt geschrieben werden:

```c
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
   printf($SG3830);
   return 0;
}
```

Nochmal zurück zum Assembler-Listing: wie man sehen kann ist die Zeichenkette gemäß dem C/C++-Standard mit einem 0-Byte abgeschlossen. Mehr über C/C++-Zeichenketten ist im Abschnitt **5.4.1 on page 423** zu finden.

In dem Code-Segment _TEXT ist lediglich eine Funktion: main(). Diese startet mit einem Prolog-Teil und endet mit einem Epilog-Teil (wie fast alle Funktionen) **14**.

Nach dem Funktions-Prolog ist der Aufruf der `printf()`-Funktion zu sehen: `CALL _printf`. Vor dem Aufruf wird die Adresse der Zeichenkette (oder ein Zeiger darauf) mit dem Inhalt unserer Begrüßung auf dem Stack gespeichert. Dies geschieht durch die PUSH-Anweisung.

Wenn `printf()` die Ausführung wieder an main() über gibt, befindet sich die Adresse der Zeichenkette (oder ein Zeiger darauf) immer noch auf dem Stack. Da diese jedoch nicht mehr benötigt wird, muss der Stapel-Zeiger (das ESP-Register) korrigiert werden.

ADD ESP, 4 bedeutet, dass der Wert 4 zu dem ESP-Register-Wert addiert wird.

Warum 4? Da dies ein 32-Bit-Programm ist, werden exakt 4 Byte benötigt um Adressen auf dem Stack abzulegen. Wenn dies x64-Code wäre, würden 8 Byte benötigt. ADD ESP, 4 ist quasi gleichbedeutend mit POP Register jedoch ohne die Verwendung von Registern **15**.

Aus dem gleichen Grund generieren einige Compiler (wie der Intel C++-Compiler) die Anweisung POP ECX anstatt ADD (dieses Muster kann zum Beispiel im Oracle RDBMS-Code gefunden werden, da dieser mit dem Intel-Compiler erstellt wurde). Diese Anweisung hat nahezu den gleichen Effekt, nur dass die Inhalte des ECX-Registers überschrieben werden. Der Intel C++-Compiler nutzt POP ECX vermutlich, da der OpCode für diese Anweisung kürzer ist als ADD ESP, x (1 Byte für POP und 3 Byte für ADD).

**14** Mehr darüber in dem Abschnitt über Funktions-Prologe und -Epiloge (?? on page ??).

**15** Statusregister der CPU können sich jedoch ändern
Nachfolgend ein Beispiel unter der Verwendung von POP anstatt ADD aus Oracle RDBMS:

<table>
<thead>
<tr>
<th>Listing 1.14: Oracle RDBMS 10.2 Linux (app.o file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:0800029A push ebx</td>
</tr>
<tr>
<td>.text:0800029B call qksfroChild</td>
</tr>
<tr>
<td>.text:080002A0 pop ecx</td>
</tr>
</tbody>
</table>

Nachdem printf() aufgerufen wurde, enthält der Original-C/C++-Code die Anweisung return 0 als Rückgabewert der main()-Funktion.

In dem hier gezeigten Code ist dies durch die Anweisung XOR EAX, EAX realisiert. XOR ist lediglich ein „exklusives Oder“ aber der Compiler nutzt dies oft anstatt MOV EAX, 0—auch hier wieder aufgrund des leicht kürzeren OpCodes (2 Byte für XOR und 5 Byte für MOV).

Einige Compiler erzeugen SUB EAX, EAX, was Subtrahiere den Wert in EAX vom Wert in EAX bedeutet. In jedem Fall erzeugt dies einen Wert von Null.

Die letzte Anweisung RET gibt die Ausführungskontrolle wieder an die aufrufende Funktion caller. Üblicherweise ist dies C/C++ CRT-Code welcher wiederum die Kontrolle an das Betriebssystem (BS) übergibt.

GCC

Als nächstes wird der gleiche C/C++-Code mit GCC 4.4.1 unter Linux kompiliert: gcc 1.c -o 1. Mithilfe des IDA-Disassemblers wird untersucht, wie die main()-Funktion erzeugt wurde. IDA nutzt, genau wie MSVX den Intel-Syntax.

<table>
<thead>
<tr>
<th>Listing 1.15: Code in IDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>main proc near</td>
</tr>
<tr>
<td>var_10 = dword ptr -10h</td>
</tr>
<tr>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>and esp, 0FFFFFFF0h</td>
</tr>
<tr>
<td>sub esp, 10h</td>
</tr>
<tr>
<td>mov eax, offset aHelloWorld; &quot;hello, world\n&quot;</td>
</tr>
<tr>
<td>mov [esp+10h+var_10], eax</td>
</tr>
<tr>
<td>call printf</td>
</tr>
<tr>
<td>mov eax, 0</td>
</tr>
<tr>
<td>leave</td>
</tr>
<tr>
<td>retn</td>
</tr>
</tbody>
</table>

Das Ergebnis ist fast das gleiche. Die Adresse der hello, world-Zeichenkette (im Daten-Segment) wird zunächst in das EAX-Register geladen und anschließend auf dem Stack gesichert.

Zusätzlich beinhaltet der Funktions-Prolog AND ESP, 0FFFFFFF0h —diese Anweisung richtet den ESP-Register-Wert an eine 16-Byte-Grenze aus. Dies führt dazu, dass alle Werte im Stack auf die gleiche Weise ausgerichtet sind. Die CPU kann Anweisungen schneller ausführen, wenn die zu verarbeitenden Daten auf einer an 4- oder 16-Byte-Grenzen ausgerichteten Adresse liegen.

SUB ESP, 10h reserviert 16 Byte auf dem Stack, auch wenn - wie später gezeigt wird - nur 4 Byte benötigt werden.

Der Grund liegt darin, dass auch die Größe des Stacks an eine 16-Byte-Grenze ausgerichtet ist.

Die Adresse der Zeichenkette (oder ein Zeiger darauf) wird anschließend direkt ohne die PUSH-Anweisung auf dem Stack gespeichert. ITvar_10 —ist eine lokale Variable und ein Argument für printf(). Mehr dazu später.

Anschließend wird die printf()-Funktion aufgerufen.

Anders als MSVC erzeugt GCC ohne Optimierung Die Anweisung MOV EAX, 0 anstatt des kürzeren OpCodes.

---

16 Wikipedia
17 C Runtime library
18 GCC kannAssembler-Ausgaben im Intel-Syntax erzeugen mit der Options -S -masm=intel.
Die letzte Anweisung LEAVE ist ein Äquivalent zu der Kombination aus MOV ESP, EBP und POP EBP. Mit anderen Worten: diese Anweisung setzt den Stapel-Zeiger (ESP) zurück und stellt die initialen Werte des EBP-Registers wieder her. Dies ist notwendig weil die Registerwerte (ESP und EBP) zu Beginn der Funktion (durch MOV EBP, ESP / AND ESP,...).

**GCC: German text placeholder**

Im nächsten Beispiel ist sichtbar, wie dies im AT&T-Syntax dargestellt werden kann. Dieser Syntax ist sehr viel populärer in der UNIX-Welt.

Listing 1.16: Das Beispiel kompiliert mit GCC 4.7.3

```
gcc -S 1_1.c
```

Das Ergebnis ist wie folgt:

```plaintext
Listing 1.17: GCC 4.7.3

```

Der Quellcode beinhaltet Makros (beginnend mit einem Punkt), die hier aber nicht von Belang sind. An dieser Stelle werden aus Gründen der Übersichtlichkeit alle Makros außer .string ignoriert. Letzteres kodiert eine Null-terminierte Zeichenkette, die einem C-String entspricht.

Die resultierende Ausgabe ist diese 19:

Listing 1.18: GCC 4.7.3

```
```

19Um die „unnötigen“ Makros zu unterdrücken kann die GCC-Option -fno-asynchronous-unwind-tables genutzt werden
Einige der Hauptunterschiede zwischen Intel und AT&T-Syntax sin:

- Quell- und Zieloperanden sind in umgekehrter Reihenfolge angegeben.
  - Im Intel-Syntax: `<Anweisung> <Ziel-Operand> <Quell-Operand>`.
  - Im AT&T-Syntax: `<Anweisung> <Quell-Operand> <Ziel-Operand>`.

Hier ist eine einfache Möglichkeit um sich den Unterschied zu merken: Beim Umgang mit dem Intel-Syntax, kann man sich ein Gleichheitszeichen (=) zwischen den Operanden vorstellen und beim AT&T-Syntax einen Pfeil nach rechts (→) ²⁰.

- AT&T: Vor einem Register-Namen muss ein Prozentzeichen (%) und vor Zahlen ein Dollarzeichen ($) stehen. Statt eckigen werden runde Klammern genutzt.

- AT&T: An eine Anweisung ist ein Suffix angehängt, der die Operandengröße angibt:
  - q — quad (64 bits)
  - l — long (32 bits)
  - w — word (16 bits)
  - b — byte (8 bits)

Nochmals zu dem kompilierten Ergebnis: Dieses ist identisch mit der Anzeige in IDA, jedoch mit einem kleinen Unterschied: 0xFFFFFFFF0 wird als -16 angezeigt. Der eigentliche Wert ist der selbe: 16 im Dezimalsystem ist 0x10 im Hexadezimalsystem. Für 32-Bit-Datentypen ist -0x10 identisch mit 0xFFFFFFFF0.


**String-Patching (Win32)**

Man kann die Zeichenkette "hello, world" in der ausführbaren Datei mit Hiew finden:

![Hiew](https://example.com/hiew.png)

**Abbildung 1.1: Hiew**

Man kann jetzt versuchen die Meldung ins Spanische zu übersetzen:

²⁰Einige C-Standard-Funktionen (z.B. memcpy(), strcpy()) sind die Parameter ebenfalls wie im Intel-Syntax aufgelistet: erst der Zeiger zum Ziel, dann der Zeiger auf die Speicher-Quelle)
Die spanische Version ist ein Byte kürzer als die englische, als muss am Ende ein 0x0A-Byte (\n) und ein Null-Byte eingefügt werden.

Es funktioniert.

Was wenn eine längere Nachricht eingefügt werden soll? Hinter dem originalen englischen Text befinden sich einige Nullbytes. Es ist schwierig zu sagen, ob diese überschrieben werden dürfen: es ist möglich, dass die zum Beispiel in dem CRT-Code genutzt werden. Vielleicht aber auch nicht. Wie dem auch sei: diese Daten sollten nur überschrieben werden, wenn wirklich klar ist was man tut.

**String-Patching (Linux x64)**

Nachfolgend wird der Patch einer ausführbaren Datei unter einem 64 Bit-Linux mit rada.re gezeigt:

```
dennis@bigbox ~/tmp % gcc hw.c

dennis@bigbox ~/tmp % radare2 a.out

-- SHALL WE PLAY A GAME?

[0x00400430]> / hello

Searching 5 bytes from 0x00400000 to 0x00601040: 68 65 6c 6c 6f

Searching 5 bytes in [0x400000-0x601040]

hits: 1

0x004005c4 hit0_0 .HHhello, world;0.

[0x004005c4]> s 0x004005c4

px

- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF

0x004005c4 6865 6c6c 6f2c 2077 6f72 6c64 0000 0000 hello, world....

0x004005d4 011b 033b 3000 0000 0500 0000 1cfe ffff ...;0........

0x004005e4 7c00 0000 5cfe ffff 4c00 0000 52ff ffff |...\...L...R...

0x004005f4 a400 0000 6c00 0000 feff ffff ....l........

0x00400604 0c00 0000 1400 0000 0000 0000 017a 5200 .............zR.

0x00400614 0178 1001 1b0c 0708 9001 0710 1400 0000 .x..............

0x00400624 1c00 0000 08fe ffff 2a00 0000 0000 0000 ........*.......

0x00400634 0000 0000 1400 0000 0000 0000 017a 5200 .............zR.

0x00400644 0178 1001 1b0c 0708 9001 0710 1400 0000 .x..............

0x00400654 1c00 0000 08fe ffff 2a00 0000 0000 0000 ........$.......

0x00400664 0000 0000 1400 0000 0000 0000 017a 5200 .............zR.

0x00400674 0178 1001 1b0c 0708 9001 0710 1400 0000 .x..............

0x00400684 1c00 0000 08fe ffff 2a00 0000 0000 0000 ........*.......

0x00400694 0000 0000 1400 0000 0000 0000 017a 5200 .............zR.

0x004006a4 0178 1001 1b0c 0708 9001 0710 1400 0000 .x..............

0x004006b4 1c00 0000 08fe ffff 2a00 0000 0000 0000 ........$.......

0x004006c4 0000 0000 1400 0000 0000 0000 017a 5200 .............zR.

0x004006d4 0178 1001 1b0c 0708 9001 0710 1400 0000 .x..............

0x004006e4 1c00 0000 08fe ffff 2a00 0000 0000 0000 ........*.......

0x004006f4 0000 0000 1400 0000 0000 0000 017a 5200 .............zR.

0x00400704 0178 1001 1b0c 0708 9001 0710 1400 0000 .x..............

0x00400714 1c00 0000 08fe ffff 2a00 0000 0000 0000 ........$.......

0x00400724 0000 0000 1400 0000 0000 0000 017a 5200 .............zR.

0x00400734 0178 1001 1b0c 0708 9001 0710 1400 0000 .x..............

0x00400744 1c00 0000 08fe ffff 2a00 0000 0000 0000 ........$.......

0x00400754 0000 0000 1400 0000 0000 0000 017a 5200 .............zR.

0x00400764 0178 1001 1b0c 0708 9001 0710 1400 0000 .x..............

0x00400774 1c00 0000 08fe ffff 2a00 0000 0000 0000 ........$.......

0x00400784 0000 0000 1400 0000 0000 0000 017a 5200 .............zR.

0x00400794 0178 1001 1b0c 0708 9001 0710 1400 0000 .x..............

0x004007a4 1c00 0000 08fe ffff 2a00 0000 0000 0000 ........$.......

0x004007b4 0000 0000 1400 0000 0000 0000 017a 5200 .............zR.

0x004007c4 0178 1001 1b0c 0708 9001 0710 1400 0000 .x..............

0x004007d4 1c00 0000 08fe ffff 2a00 0000 0000 0000 ........$.......

0x004007e4 0000 0000 1400 0000 0000 0000 017a 5200 .............zR.

0x004007f4 0178 1001 1b0c 0708 9001 0710 1400 0000 .x..............

0x00400804 1c00 0000 08fe ffff 2a00 0000 0000 0000 ........$.......

[0x0040080c]> oo+

File a.out reopened in read-write mode

[0x0040080c]> w hola, mundo\x00

[0x0040080c]> q
```
Was hier passiert ist folgendes: suchen von „hello“ mit dem `/`-Kommando, dann Setzen des cursor (oder `seek` im rada.re-Wording) an diese Adresse. Um sicher zu gehen, dass die richtige Stelle gesetzt ist, kann mit px der Datenblock ausgegeben werden. `00+` versetzt rada.re in den Lese-Schreib-Modus. w schreibt einen ASCII string an die aktuelle Adresse. Hinweis: `\00 am Ende ist das Null-Byte. q beendet rada.re.

Software-Lokalisation zu MS-DOS-Zeiten

Der hier beschriebene Weg war in den 1980ern und 1990ern sehr verbreitet, um MS-DOS-Programme in die russische Sprache zu übersetzen. Russische Wörter und Sätze sind in der Regel etwas länger als ihre englischen Gegenstücke, was der Grund ist, dass viele lokalisierte Programme eine Menge seltsamer Akronyme und Abkürzungen haben.

Möglicherweise passierte dies in der Zeit auch in anderen Sprachen.

In Delphi müssen die Längen der Zeichenketten falls nötig korrigiert werden.

1.5.2 x86-64

MSVC: x86-64

Hier das gleiche Beispiel mit der 64-Bit-Variante von MSVC kompiliert:

```
Listing 1.20: MSVC 2012 x64

$SG2989 DB 'hello, world', 0AH, 00H

main PROC
  sub  rsp, 40
  lea  rcx, OFFSET FLAT:$SG2989
  call printf
  xor  eax, eax
  add  rsp, 40
  ret  0
main ENDP
```

In x86-64 wurden alle Register auf 64-Bit erweitert und die Registernamen mit einem R-Prefix versehen. Um den Stack weniger oft zu nutzen (also um auf externen Speicher / Cache selterner zuzugreifen), existiert ein verbreiteter Weg um Funktionsargumente per Register (fastcall) 6.1.3 on page 449 zu übergeben. Das heißt ein Teil der Funktionsargumente wird in Registern übergeben, der Rest—über den Stack. In Win64 werden vier Funktionsargumente in den Registern RCX, RDX, R8 und R9 übergeben. Das ist was hier sichtbar ist: der Zeiger zu der Zeichenkette für `printf()` ist jetzt nicht im Stack übergeben sondern im RCX-Register. Die Zeiger sind nun 64-Bit breit, also werden sie in den 64-Bit-Registern übergeben (die jetzt den R-Prefix haben). Aus Gründen der Rückwärtskompatibilität ist es aber immer noch möglich mit dem E-Prefix auf 32-Bit-Teile zuzugreifen. Nachfolgend, der Aufbau der RAX/EAX/AX/AL-Register in x86-64:

```
<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX\x64</td>
</tr>
<tr>
<td>EAX</td>
</tr>
<tr>
<td>AX</td>
</tr>
<tr>
<td>AH</td>
</tr>
</tbody>
</table>
```

Die `main()`-Funktion gibt einen Wert vom Typ `int` zurück, der in C/C++ aus Gründen der Kompatibilität und Portabilität immernoch 32 Bit breit ist. Daher wird am Ende der Funktion das EAX-Register auf Null
gesetzt (das heißt der 32-Bit-Part des Registers) anstatt RAX. Auf dem lokalen Stack sind zusätzliche 40 Byte reserviert. Dieser Bereich wird „shadow space“ genannt und wird in Abschnitt 1.9.2 on page 84 noch genauer betrachtet.

**GCC: x86-64**

Nachfolgend das Beispiel unter einem 64 Bit-Linux-System mit GCC kompiliert:

```
.string "hello, world\n"
main:
    sub    rsp, 8
    mov    edi, OFFSET FLAT:format ; "hello, world\n"
    xor    eax, eax ; Anzahl der uebergebenen Register
    call   printf
    xor    eax, eax
    add    rsp, 8
    ret
```


Der Zeiger zu der Zeichenkette ist in EDI (also, dem 32-Bit-Teil) gesichert. Warum wird nicht der 64-Bit-Teil RDI genutzt?

Es ist wichtig sich zu vergegenwärtigen, dass alle MOV-Anweisungen im 64-Bit-Modus, die etwas in den niederwertigen 32-Bit-Teil eines Registers schreiben, auch den höherwertigen 32-Bit-Teil des Registers löschen (siehe Intel-Handbücher: 11.1.4 on page 520).

Die Anweisung MOV EAX, 01123344h schreibt also den richtigen Wert in RAX, weil die höherwertigen Bits auf Null gesetzt werden.

In der Objekt-Datei (.o) eines Kompilats sind ebenfalls alles OpCodes der verwendeten Anweisungen zu sehen. 22:

```
.text:00000000004004D0  main   proc near
.text:00000000004004D0  48 83 EC 08  sub    rsp, 8
.text:00000000004004D4  BF E8 05 40 00  mov    edi, offset format ; "hello, world\n"
.text:00000000004004D9  31 C0  xor    eax, eax
.text:00000000004004DB  E8 D8 FE FF FF  call   _printf
.text:00000000004004E0  31 C0  xor    eax, eax
.text:00000000004004E2  48 83 C4 08  add    rsp, 8
.text:00000000004004E6  C3  ret
.text:00000000004004E6  main   endp
```

Wie man sehen kann verändert die Anweisung zum Schreiben in EDI an der Adresse 0x4004D4 fünf Byte. Dieselbe Anweisung die einen 64-Bit-Wert in RDI schreibt, verändert 7 Bytes. Offensichtlich versucht GCC etwas Speicherplatz zu sparen. Nebenbei ist es sicher, dass das Datensegment, welches die Zeichenkette enthält niemals an Adressen höher 4GiB reserviert wird.


**Adress-Patching (Win64)**

Wenn das Beispiel in MSCV2013 mit der Option /MD kompiliert wird (was zu einer kleineren ausführbaren Datei durch das linken mit MSVCR*.DLL führt), kommt zuerst die main() -Funktion und kann einfach gefunden werden:

รวมทุกข้อความในภาพ:

2. Dies muss aktiviert werden: Optionen → Disassembly → Number of opcode bytes
3. ABI!
Als Experiment kann die Adresse des Pointers um 1 **German text placeholder** werden:
Abbildung 1.5: Hiew

Hiew zeigt „ello, world“ als Zeichenkette und beim Ausführen der gepatchten Datei wird eben dieser Text ausgegeben.

**Aussuchen einer anderen Zeichenkette einer Binärdatei (Linux x64)**

Die Binärdatei die beim Kompilieren des Beispiels mit GCC 5.4.0 unter Linux x64 entsteht, beinhaltet noch viele andere Zeichenketten: die meisten sind importierte Funktions- und Bibliotheksnamen.

Mit objdump können die Inhalte aller Sektionen der kompilierten Datei ausgegeben werden:

```
$ objdump -s a.out
```

```
a.out: file format elf64-x86-64

Contents of section .interp:
   400238 2f6c6962 36342f6c 642d6l6e75782d /lib64/ld-linux-x86-64
   400248 78362d 36342e73 6f2e3200 x86-64.so.2.

Contents of section .note.ABI-tag:
   400254 04000000 10000000 01000000 474e5500 ..........GNU.
   400264 00000000 02000000 06000000 20000000 .......... ...

Contents of section .note.gnu.build-id:
   400274 04000000 14000000 03000000 474e5500 ..........GNU.
   400284 fe461178 5bb710b4 bbf2aca8 5ec1ecl0 .F.x[.......^...
   400294 cf3f7ae4 .^?

... Es ist kein Problem die Adresse der Zeichenkette „/lib64/ld-linux-x86-64.so.2“ an printf() zu übergeben:

```
#include <stdio.h>

int main()
{
    printf(0x400238);
    return 0;
}

Schwer zu glauben, aber dieser Code gibt die erwähnte Zeichenkette aus.
Beim Ändern der Adresse zu 0x400260 wird die Zeichenkette „GNU“ ausgegeben. Diese Adresse gilt für die hier verwendete GCC-Version, Toolkonfiguration und so weiter. Auf anderen Systemen kann die ausführbare Datei leicht unterschiedlich sein, was auch die Adressen verändern kann. Auch das Hinzufügen und Entfernen von Quellcode kann Adressen vor- und zurückschieben.

1.5.3 ARM
Für die Experimente mit ARM-Prozessoren wurden verschiedene Compiler genutzt:
- Apple Xcode 4.6.3 IDE mit dem LLVM-GCC 4.2-Compiler 24.
- GCC 4.9 (Linaro) (für ARM64), verfügbar als Win32-Executable unter http://go.yurichev.com/17325.

Wenn nicht anders angegeben wird immer der 32-Bit ARM-Code (inklusive Thumb und Thumb-2-Mode) genutzt. Wenn von 64-Bit ARM die Rede ist, dann wird ARM64 geschrieben.

**German text placeholder Keil 6/2013 (German text placeholder)**

Beginnen wir mit dem Kompilieren des Beispiels mit Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

Der armcc-Compiler erstellt Assembler-Quelltext im Intel-Syntax, hat aber High-Level-Makros bezüglich der ARM-Prozessoren 25. Es ist hier wichtig die „richtigen“ Anweisungen zu sehen, deswegen ist hier das Ergebnis mit IDA kompiliert.

```
Listing 1.23: German text placeholder Keil 6/2013 (German text placeholder) IDA

.text:00000000 main
.text:00000000 10 40 2D E9 STMFD SP!, {R4,LR}
.text:00000004 1E 0E 8F E2 ADR R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB BL 2printf
.text:0000000C 00 00 A0 E3 MOV R0, #0
.text:00000010 10 80 BD E8 LDMFD SP!, {R4,PC}
.text:0000001EC 68 65 6C 6C+aHelloWorld DCB "hello, world",0 ; DATA XREF: main+4
```

Im ersten Beispiel ist zu erkennen, dass jede Anweisung 4 Byte groß ist. Tatsächlich wurde der Code für den ARM- und nicht den Thumb-Mode erstellt.

Die erste Anweisung, STMFD SP!, {R4,LR} 26, arbeitet wie eine x86-PUSH-Anweisung um die Werte der beiden Register (R4 and LR) auf den Stack zu legen.


---

24 Tatsächlich nutzt Apple Xcode 4.6.3 GCC als Front-End-Compiler und LLVM Code Generator
25 d.h. der ARM-Mode hat keine PUSH/POP-Anweisungen
26 STMFD
Die Anweisung dekrementiert zunächst den \textit{SP}^{28}, so dass er auf den Bereich im Stack zeigt, der für neue Einträge frei ist. Anschließend werden die Werte der Register R4 und \textit{LR} an der Adresse gespeichert auf den der (modifizierten) \textit{SP} zeigt.

Diese Anweisungen (wie \texttt{PUSH} im Thumb-Mode) ist in der Lage mehrere Register-Werte auf einmal zu speichern, was sehr nützlich sein kann. Übrigens: in x86 gibt es dazu kein Äquivalent. Außerdem ist erwähnenswert, dass die STMFD-Anweisung eine Generalisierung der \texttt{PUSH}-Anweisung (ohne deren Eigenschaften) ist, weil sie auf jedes Register angewandt werden kann und nicht nur auf \textit{SP}. Mit anderen Worten kann STMFD genutzt werden um eine Reihe von Registern an einer angegebenen Speicher-Adresse zu sichern.

Die ADR R0, aHelloWorld-Anweisung addiert oder subtrahiert den Wert im \texttt{PC}^{29}-Register zum Offset an dem die hello, world-Zeichenkette ist. Man kann sich nun fragen, wie das PC-Register hier genutzt wird. Dies wird „positionsabhängiger Code“^{30} genannt.

Code dieser Art kann an nicht-festen Adressen im Speicher ausgeführt werden. Mit anderen Worten: dies ist \texttt{PC}-relative Adressierung. Die ADR-Anweisung berücksichtigt den Unterschied zwischen der Adresse dieser Anweisung und der Adresse an dem die Zeichenkette gespeichert ist. Der Unterschied (Offset) ist immer gleich, egal an welcher Adresse der Code vom BS geladen wurden. Dementsprechend ist alles was gemacht werden muss, die Adresse der aktuellen Anweisung (vom PC) zu addieren um die absolute Speicheradresse der Zeichenkette zu bekommen.

\hspace{1cm} \texttt{BL \_2printf}^{31}-Anweisung ruft die \texttt{printf()}-Funktion auf. Die Anweisung funktioniert wie folgt:

\begin{itemize}
  \item Speichere die Adresse hinter der BL-Anweisung (0xC) in \textit{LR};
  \item anschließend wird übergebe die Kontrolle an \texttt{printf()} indem dessen Adresse ins \texttt{PC}-Register geschrieben wird.
\end{itemize}

Wenn \texttt{printf()} die Ausführung beendet, müssen Informationen vorliegen, wo die Ausführung weitergehen soll. Das ist der Grund warum jede Funktion die Kontrolle an die Adresse, gespeichert im \textit{LR}-Register übergibt.

Dies ist ein Unterschied zwischen einem „reinem“ RISC-Prozessor wie ARM und CISC^{32}-Prozessoren wie x86, bei denen die Rücksprungsadresse in der Regel auf dem Stack gespeichert wird. Mehr dazu ist im nächsten Abschnitt zu lesen (\textit{1.6 on page 29}).


Als nächstes schreibt die Anweisung \texttt{MOV R0, \#0}^{33} lediglich 0 in das R0-Register weil der Rückgabewert hier gespeichert wird und die gezeigte C-Funktion 0 als Argument für die return-Anweisung hat.

Die letzte Anweisung LDMFD SP!, R4,PC^{34} lädt die Werte nacheinander vom Stack (oder eine andere Speicheradresse) um sie in die Register R4 und PC zu sichern. Außerdem wird der Stack Pointer SP inkrementiert. Hier arbeitet der Befehl wie POP.

Die erste Anweisung STMFD sichert das Register-Paar R4 und LR auf dem Stack, jedoch werden R4 und PC während der Ausführung von LDMFD wiederhergestellt.

Wie bereits bekannt, wird die Adresse die nach der Ausführung einer Funktion angesprungen wird in dem LR-Register gesichert. Die allererste Anweisung sichert diese Wert auf dem Stack weil das gleiche Register von der main() -Funktion genutzt wird, wenn printf() aufgerufen wird. Am Ende der Funktion kann dieser Wert direkt in das PC-Register geschrieben werden und so die Ausführung an der Stelle fortgesetzt werden an der die Funktion aufgerufen wurde.

Da \texttt{main()} in der Regel die erste Funktion in C/C++ ist, wird die Kontrolle an das BS oder einen Punkt in der CRT übergeben.

All dies erlaubt das Auslassen der BX LR-Anweisung am Ende der Funktion.

\begin{itemize}
  \item \texttt{PUSH} SP/ESP/RSP \texttt{German text placeholder} x86/x64. \texttt{SP German text placeholder} ARM.
  \item Program Counter. IP/EIP/RIP \texttt{German text placeholder} x86/64. \texttt{PC German text placeholder} ARM.
  \item \texttt{Branch with Link}
  \item \texttt{Complex Instruction Set Computing}
  \item das heißt MOVE
  \item \texttt{LDMFD}^{35} ist eine inverse Anweisung von STMFD
\end{itemize}
DCB ist eine Assemblerdirektive die ein Array von Bytes oder ASCII anlegt, ähnlich der DB-Direktive in der x86-Assembler-Sprache.

**German text placeholder Keil 6/2013 (German text placeholder)**

Nachfolgend das gleiche Beispiel mit dem Keil-Compiler im Thumb-Mode erstellt:

```
armcc.exe --thumb --c90 -O0 1.c
```

In *IDA* wird folgende Ausgabe erzeugt:

```
Listing 1.24: German text placeholder Keil 6/2013 (German text placeholder) + IDA

```
<table>
<thead>
<tr>
<th>.text:00000000  main</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:00000000  10  B5  PUSH  {R4,LR}</td>
</tr>
<tr>
<td>.text:00000002  C0  A0  ADR  R0, aHelloWorld ; &quot;hello, world&quot;</td>
</tr>
<tr>
<td>.text:00000004  06  F0  2E  F9  BL  __2printf</td>
</tr>
<tr>
<td>.text:00000008  00  20  MOV  R0, #0</td>
</tr>
<tr>
<td>.text:0000000A  10  BD  POP  {R4,PC}</td>
</tr>
<tr>
<td>.text:00000304  68  65  6C  +aHelloWorld  DCB &quot;hello, world&quot;,0 ; DATA XREF: main+2</td>
</tr>
</tbody>
</table>
```

Leicht zu erkennen sind die 2-Byte (16 Bit) OpCodes, die wie bereits erwähnt Thumb-Anweisungen sind. Die BL-Anweisung besteht aus zwei 16-Bit-Anweisungen, weil es für die printf()-Funktion unmöglich ist einen Offset zu laden, wenn der kleine Speicherbereich in einem 16-Bit-Opcode genutzt wird. Aus diesem Grund lädt die erste 16-Bit-Anweisung die höherwertigen 10 Bit des Offsets und die zweite Anweisung die niederwertigen 11 Bit.


Zusammenfassend kann die BL-Thumb-Anweisung eine Adresse bis current_PC ± 2M kodieren.

Wie für die anderen Anweisungen in dieser Funktion arbeiten PUSH und POP wie die beschreibenden STMFD/LDMFD, nur dass das SP-Register hier nicht explizit genannt wird. ADR arbeitet genau wie in dem vorherigen Beispiel. MOVZ schreibt 0 in das Register R0 um 0 zurückzugeben.

**German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**

Xcode 4.6.3 ohne Optimierung produziert eine Menge redundanten Code, so dass im Folgenden die optimierte Ausgabe gelistet ist bei der die Anzahl der Anweisungen so klein wie möglich ist. Der Compiler-Schalter ist -O3.

```
Listing 1.25: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)
```

```
<table>
<thead>
<tr>
<th>_text:000028C4  hello_world</th>
</tr>
</thead>
<tbody>
<tr>
<td>_text:000028C4  80  40  2D  E9  STMFD  SP!, {R7,LR}</td>
</tr>
<tr>
<td>_text:000028C8  86  06  01  E3  MOV  R0, #0x1686</td>
</tr>
<tr>
<td>_text:000028CC  0D  70  A0  E1  MOV  R7, SP</td>
</tr>
<tr>
<td>_text:000028D0  00  00  40  E3  MOVT  R0, #0</td>
</tr>
<tr>
<td>_text:000028D4  00  00  8F  E0  ADD  R0, PC, R0</td>
</tr>
<tr>
<td>_text:000028D8  C3  05  00  EB  BL  _puts</td>
</tr>
<tr>
<td>_text:000028DC  00  00  A0  E3  MOV  R0, #0</td>
</tr>
<tr>
<td>_text:000028E0  80  80  BD  E8  LDMFD  SP!, {R7,PC}</td>
</tr>
<tr>
<td>_cstring:00003F62  48  65  6C  6C+aHelloWorld_0  DCB &quot;Hello world!&quot;,0</td>
</tr>
</tbody>
</table>
```

Die Anweisungen STMFD und LDMFD sind bereits bekannt. Die MOV-Anweisung schreibt lediglich die Nummer 0x1686 in das Register R0. Dies ist der Offset der auf die Zeichenkette „Hello world!” zeigt.

Die MOV T R0, #0 (MOV e T op)-Anweisung schreibt 0 in die höherwertigen 16 Bit des Registers. Das Problem ist hier, dass die generische MOV-Anweisung im ARM-Mode nur die niederwertigen 16 Bit des Registers beschreibt.


Die Anweisung ADD R0, PC, R0 addiert den Wert im PC zum Wert im Register R0 um die absolute Adresse der „Hello world!“-Zeichenkette zu berechnen. Wie bereits bekannt ist dies „positionsabhängiger Code“, so dass diese Korrektur hier unbedingt notwendig ist.

Die BL-Anweisung ruft puts() anstatt printf() auf.

LLVM ersetzt den ersten printf()-Aufruf mit puts(). In der Tat ist printf() mit nur einem Argument identisch mit puts().

Die beiden Funktionen produzieren lediglich das gleiche Ergebnis, weil printf keine Formatkennzeichner, beginnend mit %, enthält. Sollte dies jedoch der Fall sein, wäre die Auswirkung der beiden Funktionen unterschiedlich.37

Warum hat der Compiler diese Ersetzung durchgeführt? Vermutlich hat dies Vorteile bei der Geschwindigkeit, weil puts() schneller ist und lediglich die Zeichen zu German text placeholder übergibt, anstatt jedes Zeichen mit % zu vergleichen.

Als nächstes ist die bekannte Anweisung MOV R0, #0 zu sehen um das Register R0 auf 0 zu setzen.

**German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**

Standardmäßig generiert Xcode 4.6.3 den Thumb-2-Code auf folgende Weise:

```
Listing 1.26: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)

<table>
<thead>
<tr>
<th>text:0000286C</th>
<th>__text:0000286C _hello_world</th>
</tr>
</thead>
<tbody>
<tr>
<td>text:0000286C 80 B5</td>
<td>PUSH {R7,LR}</td>
</tr>
<tr>
<td>text:0000286E 41 F2 D8 30</td>
<td>MOVW R0, #0x13D8</td>
</tr>
<tr>
<td>text:00002872 6F 46</td>
<td>MOV R7, SP</td>
</tr>
<tr>
<td>text:00002874 C0 F2 00 00</td>
<td>MOVT,W R0, #0</td>
</tr>
<tr>
<td>text:00002878 78 44</td>
<td>ADD R0, PC</td>
</tr>
<tr>
<td>text:0000287A 81 F6 38 EA</td>
<td>BLX puts</td>
</tr>
<tr>
<td>text:0000287E 00 20</td>
<td>MOVS R0, #0</td>
</tr>
<tr>
<td>text:00002880 80 0D</td>
<td>POP {R7,PC}</td>
</tr>
</tbody>
</table>

| text:000028B0 | 48 65 6C 6F 20+aHelloWorld DCB "Hello world!",0xA,0 |

```


Offensichtlich beginnen die Opcodes der Thumb-2-Anweisungen immer mit 0xFx oder 0xEx.

Im IDA-Listing jedoch sind die Bytes der Opcodes vertauscht weil für den ARM-Prozessor die Anweisungen wie folgt kodiert werden: Das letzte Byte kommt zuerst und danach das erste (für Thumb- und Thumb-2-Mode) oder für Anweisungen im ARM-Mode kommt das vierte Byte zuerst, dann das dritte, dann das zweite und zum Schluss das erste (aufgrund des unterschiedlichen German text placeholder).

Die Bytes sind also im IDA-Listing wie folgt angeordnet:

- für ARM und ARM64 Mode: 4-3-2-1;
- für Thumb Mode: 2-1;

36German text placeholder http://go.yurichev.com/17276
37Des weiteren benötigt puts() kein ‘n’ für den Zeilenumbruch am Ende der Zeichenkette, weswegen wir dies hier nicht sehen.
38ciselant.de/projects/gcc_printf/gcc_printf.html
für 16-Bit-Anweisungspaar in Thumb-2 Mode: 2-1-4-3.

Wie zu sehen ist, beginnend die Anweisungen MOVW, MOVT.W und BLX mit 0xFx.

Eine der Thumb-2-Anweisungen ist MOVW R0, #0x13D8 — sie speichert einen 16-Bit-Wert in den niederwertigeren Teil des R0-Registers und setzt die höherwertigen Bits auf 0.

Des weiteren funktioniert MOVT.W R0, #0 genau wie MOVT aus dem vorherigen Beispiel, jedoch nur für Thumb-2.

Neben den anderen Unterschieden wird in diesem Fall die BLX-Anweisung anstatt BL genutzt.

Der Unterschied ist, dass, neben dem Speichern von RA39 in das LR-Register und die Übergabe der Ausführungskontrolle an die puts()-Funktion, der Prozessor auch vom Thumb/Thumb-2-Mode in den ARM-Mode (oder zurück) wechselt.

Diese Anweisung ist hier eingefügt weil die Anweisung mit der die Kontrolle abgegeben wird wie folgt aussieht (im ARM-Mode kodiert):

```assembly
_SYMBOLstub1:00003FEC_putts ; CODE XREF: hello world+E
_SYMBOLstub1:00003FEC 44 F0 9F E5 LDR PC, =_imp_putts
```

Dies ist im Endeffekt ein Sprung an die Stelle an der die Adresse von puts() in der import-Sektion geschrieben wird.

Der aufmerksame Leser mag fragen: warum wird puts() nicht direkt an der Stelle im Code aufgerufen, an der es benötigt wird? Dies wäre nicht sehr speicherplatzeffizient.

Fast jedes Programm nutzt externe, dynamische Bibliotheken (wie DLL in Windows, so in *NIX oder.dylib in Mac OS X). Diese Bibliotheken beinhalten häufig genutzte Funktion wie die Standard-C-Funktion puts().

In einer ausführbaren Binärdatei (Windows PE .exe, ELF oder Mach-O) existiert eine Import-Sektion. Dies ist eine Liste von Symbolen (Funktionen oder globale Variablen) die, zusammen mit den Namen, von externen Modulen importiert werden.

Der BS-Loader lädt alle Module die gebraucht werden und bestimmt die korrekten Adressen von jedem Symbol, während diese in dem primären Modul aufgelistet werden.

In dem vorliegenden Fall ist __imp_putts eine 32-Bit-Variable die vom BS-Loader genutzt wird um die korrekte Adresse der Funktion in der externen Bibliothek zu speichern. Anschließend liest die LDR-Anweisung den 32-Bit-Wert dieser Variable und schreibt ihn in das PC-Register bevor die Ausführkontrolle dorthin übergeben wird.

Um also die Zeit zu reduzieren die der BS-Loader für dieses Vorgehen benötigt, ist es eine gute Idee die Adressen für jedes Symbol einmalig an eine geeignete Stelle zu schreiben.

Daneben wurde bereits erwähnt, dass es unmöglich ist einen 32-Bit-Wert in ein Register zu laden wenn nur eine Anweisung ohne Speicher-Zugriff genutzt wird.

Aus diesem Grund ist die optimale Lösung, eine separate Funktion im ARM-Mode zu allozieren die lediglich die Aufgabe hat die Ausführkontrolle an die dynamische Bibliothek zu übergeben und dann in diese kurze Funktion mit einer Anweisung (so genannte German text placeholder) aus dem Thumb-Code auszuführen.

Übrigens: in dem vorherigen Beispiel (für ARM-Mode kompiliert) wird die Ausführkontrolle durch BL an die gleiche German text placeholder übergeben. Der Prozessor-Modus wird hier jedoch aufgrund des Fehlens eines „X“ im Anweisungsnamen nicht gewechselt.

**Mehr über Thunk-Funktionen**


Hier sind einige weitere Beschreibung dieser Funktionstypen:

---

39 Rücksprungadresse

Microsoft und IBM haben beide in ihrem Intel-basierten System eine “16-Bit Umgebung” und eine “32-Bit-Umgebung” definiert. Beide können auf dem selben Computer und demselben Betriebssystem laufen (dank dem was Microsoft „Windows On Windows“ (WOW) nennt). Sowohl MS als auch IBM haben entschieden, den Vorgang der zwischen 16- und 32-Bit wechselt “Thunk” zu nennen; für Windows 95 existiert sogar ein Tool THUNK.EXE, das Thunk-Compiler genannt wird.

( The Jargon File )

ARM64

GCC

Das Beispiel wird im Folgenden mit GCC 4.1.8 in ARM64 kompiliert:

Listing 1.27: German text placeholder GCC 4.8.1 + objdump

```
0000000000400590 <main>:  
400590:  a9bf7bfd stp x29, x30, [sp,#-16]!
400594:  910003 fd mov x29, sp
400598:  90000000 adrp x0, 400000 <_init-0x3b8>
40059c:  91192000 add x0, x0, #0x648
4005a0:  97fffffa0 bl 400420 <puts@plt>
4005a4:  52800000 mov w0, #0x0 // #0
4005a8:  a8c17bfd ldp x29, x30, [sp],#16
4005ac:  d65f03c0 ret

Contents of section .rodata:

400640 01000200 00000000 48656c6c 6f210a00 ........Hello!
```

Es gibt keine Thumb- oder Thumb-2-Modes in ARM64, sondern nur ARM, also 32-Bit-Anweisungen. Die Register-Anzahl ist verdoppelt: ?? on page ??, 64-Bit-Register haben einen X-Prefix, 32-Bit-Teile ein W-.

Die STP-Anweisung (Store Pair) speichert zwei Register auf dem Stack gleichzeitig: X29 und X30.

Natürlich kann diese Anweisung dieses Registerpaar an einer beliebigen Stelle im Speicher sichern, aber da hier das SP-Register angegeben ist, wird das Paar auf dem Stack gesichert.

ARM64-Register sind 64 Bit breit, jedes von ihnen ist 8 Byte groß. Dementsprechend werden 16 Byte für das Speichern zweier Register benötigt.

Das Ausrufszeichen (“!”) nach dem Operanden bedeutet, dass zunächst der Wert 16 vom SP subtrahiert werden muss und erst dann die Werte vom Register-Paar auf den Stack geschrieben werden. Dies wird auch pre-index genannt. Mehr über den Unterschied von post-index und pre-index ist im Abschnitt 1.27.2 on page 401 zu finden.

Im Sprachgebrauch des gebräuchlicheren x86, ist die erste Anweisung analog zu den Anweisungen PUSH X29 und PUSH X30 zu verstehen. X29 wird als FP in ARM64 genutzt, und X30 als LR, weswegen sie am Anfang der Funktion gesichert und am Ende wiederhergestellt werden.

Die zweite Anweisung kopiert SP in X29 (oder FP) um den Stack Frame vorzubereiten.

ADRP und ADD-Anweisungen werden genutzt um die Adresse der Zeichenkette „Hello!” in das Register X0 zu schreiben, da das erste Funktionsargument in an dieser Stelle übergeben wird.

Es gibt in ARM keine Anweisung, die eine große Zahl in einem Register sichern kann, weil die Länge der Anweisungen auf 4 Byte begrenzt ist. Siehe dazu auch 1.27.3 on page 402). Aus diesem Grund müssen...
mehrere Anweisungen genutzt werden. Die erste (ADRP) schreibt die Adresse der 4KiB-Page in der die Zeichenkette sich befindet in das Register X0. Die zweite (ADD) addiert lediglich den Rest der Adresse. Siehe dazu auch 1.27.4 on page 404.

0x400000 + 0x648 = 0x400648, und die Zeichenkette „Hello!“ ist im .rodata Daten-Segmet an dieser Adresse zu sehen.

puts() wird anschließend mit der BL-Anweisung aufgerufen. Dies wurde bereits diskutiert: 1.5.3 on page 21. MOV schreibt 0 in W0. W0 sind die niederwertigeren 32 Bit des 64-Bit-Registers X0:

<table>
<thead>
<tr>
<th>Oberer 32-Bit-Teil</th>
<th>Unterer 32-Bit-Teil</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0</td>
<td>W0</td>
</tr>
</tbody>
</table>

Das Ergebnis der Funktion wird über X0 zurückgegeben und main() gibt 0 zurück. Dies ist also der Weg wie das Ergebnis vorbereitet wird. Der 32-Bit-Teil wird genutzt, weil der int-Datentyp in ARM64 aus Kompatibilitätsgründen, wie in x86-64, 32 Bit breit ist.

Da die Funktion einen 32-Bit int-Wert zurück gibt, müssen lediglich die unteren 32 Bits des X0-Registers gefüllt werden.

Um dies zu überprüfen wird das Beispiel leicht verändert und neu kompiliert. main() soll nun einen 64-Bit-Wert zurück geben:

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
    printf("Hello\n");
    return 0;
}
```

Das Ergebnis ist das gleiche, allerdings sieht MOV nun wie folgt aus:

```
4005a4:     d2800000  mov  x0, #0x0     // #0
```

LDP (Load Pair) stellt anschließend die Register X29 und X30 wieder her.

An dieser Stelle steht kein Ausrufszeichen nach der Anweisung; dies impliziert, dass der Wert zunächst vom Stack gelesen wird und erst dann wird SP um den Wert 16 verringert. Dies wird post-index genannt.

Eine neue Anweisung taucht hier in ARM64 auf RET. Diese arbeitet wie BX LR, jedoch wird ein spezielles Hinweis-Bit hinzugefügt, welches die CPU darüber informiert, dass dies ein Rücksprung aus einer Funktion ist und kein anderer Sprung, so dass die Ausführung optimiert werden kann.

Aufgrund der Einfachheit dieser Funktion, erstellt der optimierende GCC den gleichen Code.

### 1.5.4 MIPS

**ein Wort über „globale Zeiger“**

Ein wichtiges Konzept bei MIPS ist der „globale Zeiger“. Wie bereits bekannt, besteht besteht jede MIPS-Anweisung aus 32 Bit, so dass es nicht möglich ist eine 32-Bit-Anweisung darin unterzubringen: ein Anweisungspaar wird verwendet (wie GCC dies in dem Beispiel zum Laden der Zeichenkettenadresse getan hat).

Es ist jedoch möglich, Daten aus dem Adressbereich register−32768...register+32767 mit nur einer Anweisung zuladen, weil ein 16 Bit vorzeichenbehafeter Offset in einer einzelnen Anweisung kodiert werden kann. Es können also einige Register zu diesen Zweck alloziert werden und 64KiB-Bereiche für die am häufigsten genutzten Daten Dieses allozierte Register wird „globaler Zeiger“ genannt und zeigt in die Mitte des 64KiB-Bereichs.
Dieser Bereich enthält in der Regel globale Variablen und Adressen von importierten Funktionen wie `printf()`, weil die GCC-Entwickler entschieden, dass das Laden einiger Funktionsadressen so schnell sein sollte wie eine einzelne Anweisung anstatt zwei. In einer ELF-Datei ist dieser 64KiB-Bereich teils in der Sektion .sbss ("small BSS") für uninitialisierte Daten und teil in .sdata ("small data") für initialisierte Daten zu finden.

Dies impliziert, dass der Programmierer entscheiden kann, auf welche Daten ein schneller Zugriff (durch das Platzieren in .sdata/.sbss) möglich sein soll. Einige Programmierer „der alten Schule“ erinnern sich vielleicht an das MS-DOS Speichermodell 10.6 on page 516 oder MS-DOS Speicherverwaltungen wie XMS/EMS bei denen der komplette Speicher in 64KiB-Blöcke unterteilt war.


**German text placeholder GCC**

Nachfolgend ein Beispiel welches das Konzept der „globalen Zeiger“ veranschaulichen soll.

```
Listing 1.30: German text placeholder GCC 4.4.5 (German text placeholder)

1 $LC0:
2 ; \000 ist das Nullbyte im Oktalsystem:
3 .ascii "Hello, world!\012\000"
4 main:
5 ; Funktionsprolog:
6 ; Setze den globalen Zeiger:
7 lui $28, %hi(__gnu_local_gp)
8 addiu $sp, $sp, -32
9 addiu $28, $28, %lo(__gnu_local_gp)
10 ; speichere Ruecksprungadresse auf lokalem Stack:
11 sw $31, 28($sp)
12 ; Lade Adresse von puts() function vom glob. Zeiger in $25:
13 lw $25, %call16(puts)($28)
14 ; Lade Adresse der Zeichenkette in $4 ($a0):
15 lui $4, %hi($LC0)
16 ; Springe zu puts(), speichere Ruecksprungadresse im Link Register:
17 jalr $25
18 addiu $4, $4, %lo($LC0) ; branch delay slot
19 ; Ruecksprungadresse wiederherstellen:
20 lw $31, 28($sp)
21 ; Kopiere 0 von $zero zu $v0:
22 move $2, $0
23 ; Springe an Ruecksprungadresse:
24 j $31
25 ; Funktionsepilog:
26 addiu $sp, $sp, 32 ; branch delay slot
```


ADDIU folgt JALR (zur Erinnerung: `branch delay slots`). Das Register $4 wird auch $A0 genannt und für das Übergeben des ersten Funktionsarguments genutzt.**

JALR („Jump and Link Register“) springt zu der Adresse die im Register $25 gespeichert ist (Adresse von `puts()`) und speichert die Adresse der übernächsten Anweisung (LW) in RA. Dies ist sehr ähnlich zu ARM. Eine wichtige Sache ist, dass die Adresse in RA nicht die Adresse der nächsten Anweisung ist (da dies ein delay slot ist und vor der Sprunganweisung ausgeführt wird), sondern die Adresse der darauf folgenden Anweisung (nach dem delay slot). Da in diesem Fall während der Ausführung von JALR der Wert PC + 8 in RA geschrieben wird, ist dies die Adresse der LW-Anweisung nach ADDIU.


MOVE in Zeile 22 kopiert der Wert vom $0 ($ZERO)-Register in $2 ($V0).


die MIPS-Register-Tabelle ist im Anhang verfügbar ?? on page ??
MIPS besitzt ein *konstantes* Register, welches immer eine Null beinhaltet. Anscheinend hatten die MIPS-Entwickler die Idee, dass eine Null die beliebteste Konstante in der Programmierung ist, also wird in Zukunft immer das $0$-Register genutzt wenn eine Null benötigt wird.


J in Zeile 24 springt zu der Adresse in RA, was im Endeffekt einem Sprung aus einer Funktion entspricht. ADDIU nach J wird tatsächlich bevor J ausgeführt (siehe branch delayslots) und ist ein Teil des Funktions-Epilogs. Hier ist die Ausgabe, die IDA generiert. Jedes Register hat einen eigenen Pseudo-Namen:

```
Listing 1.31: German text placeholder GCC 4.4.5 (IDA)
1 .text:00000000 main:
2 .text:00000000
3 .text:00000000 var_10 = -0x10
4 .text:00000000 var_4 = -4
5 .text:00000000
6 ; Funktionsprolog:
7 ; GP setzen:
8 .text:00000000 lui $gp, (___gnu_local_gp >> 16)
9 .text:00000004 addiu $sp, -0x20
10 .text:00000008 la $gp, (___gnu_local_gp & 0xFFFF)
11 ; RA auf lokalem Stack sichern:
12 .text:00000010 sw $ra, 0x20+var_4($sp)
13 ; GP auf lokalem Stack sichern:
14 ; diese Anweisung fehlt aus irgendeinem Grund bei GCC:
15 .text:00000010 sw $gp, 0x20+var_10($sp)
16 ; Lade Adresse von puts() von GP in $t9:
17 .text:00000014 lw $t9, (puts & 0xFFFF($gp))
18 ; Adresse der Zeichenkette in $a0 hinterlegen:
19 .text:00000018 lui $a0, ($LC0 >> 16) "Hello, world!"
20 ; Springe zu puts(), speichere Ruecksprungadresse im link register:
21 .text:0000001c jalr $t9
22 .text:00000020 la $a0, ($LC0 & 0xFFFF) "Hello, world!"
23 ; RA wiederherstellen:
24 .text:00000024 lw $ra, 0x20+var_4($sp)
25 ; 0 von $zero zu $v0 kopieren:
26 .text:00000028 move $v0, $zero
27 ; Rueckspaltung zu RA:
28 .text:0000002c jr $ra
29 ; Funktionsepilog:
30 .text:00000030 addiu $sp, 0x20
```

Die Anweisung in Zeile 15 speichert den GP-Wert auf dem lokalen Stack. Diese Anweisung fällt seltsamerweise beim GCC, was vielleicht auf einen Fehler des Compilers hinweist. Der GP-Wert muss auch gespeichert werden weil jede Funktion ihren eigenen 64KiB-Datenbereich nutzen kann. Das Register mit der Adresse von puts() wird $T9, da Register mit Präfix T- temporäre Register sind deren Inhalte nicht erhalten werden müssen.

**German text placeholder GCC**

German text placeholder GCC ist ausführlicher.

```
Listing 1.32: German text placeholder GCC 4.4.5 (German text placeholder)
1 $LC0:
2 .ascii "Hello, world!\012\000"
3 main:
4 ; Funktionsprolog.
5 ; Sichere RA ($31) und FP auf Stack:
6 addiu $sp,$sp,-32
7 sw $31,28($sp)
8 sw $fp,24($sp)
```

43Arithmetisch-logische Einheit
44Anscheinend sind Funktionen die Listings erzeugen nicht so kritisch für GCC-Nutzer, so dass vielleicht noch unbehandelte Fehler existieren.
; Setze FP (Stack frame pointer):
  move $fp,$sp

; Setze GP:
  lui $28,%hi(_gnu_local_gp)
  addiu $28,$28,%lo(_gnu_local_gp)
; Lade Adresse der Zeichenkette:
  lui $2,%hi($LC0)
  addiu $4,$2,%lo($LC0)
; Lade Adresse von puts() mit GP:
  lw $2,%call16(puts)($28)
  nop

; puts() aufrufen:
  move $25,$2
  jalr $25
  nop ; branch delay slot

; GP vom lokalen Stack wiederherstellen:
  lw $28,16($fp)

; Setze Register $2 ($V0) zu Null:
  move $2,$0

; Funktionsepilog.
; SP wiederherstellen:
  move $sp,$fp
; RA wiederherstellen:
  lw $31,28($sp)
; FP wiederherstellen:
  lw $fp,24($sp)
  addiu $sp,$sp,32
; Springe zu RA:
  j $31
  nop ; branch delay slot

Es ist zu sehen, dass das FP-Register als Zeiger zum Stack Frame genutzt wird. Außerdem sind im Listing drei NOP-Anweisungen. Die zweite und dritte welche der Sprunganweisung folgt. Möglicherweise fügt der GCC-Compiler immer NOP-Anweisungen nach einer Sprung hinzu (wegen der branch delay slots) und entfernt diese wenn die Optimierung eingeschaltet ist. In diesem Fall bleiben sie also bestehen.

Nachfolgend das IDA-Listing:

```
.text:00000000 main:
.text:00000000 var_10 = -0x10
.text:00000000 var_8 = -8
.text:00000000 var_4 = -4
.text:00000000
; Funktionsprolog.
; Speichere RA und FP auf dem Stack:
.text:00000000 addiu $sp, -0x20
.text:00000004 sw $ra, 0x20+var_4($sp)
.text:00000008 sw $fp, 0x20+var_8($sp)
; Setze den FP (stack frame pointer):
.text:0000000C move $fp, $sp
; Setze GP:
.text:00000010 la $gp, __gnu_local_gp
.text:00000018 sw $gp, 0x20+var_10($sp)
; Lade die Adresse der Zeichenkette:
.text:0000001C lui $v0, (aHelloWorld >> 16) # "Hello, world!"
.text:00000020 addiu $a8, $v0, (aHelloWorld & 0xFFFF) # "Hello, world!"
; Lade die Adresse von puts() mit GP:
.text:00000024 lw $v0, (puts & 0xFFFF)($gp)
.text:00000028 or $at, $zero ; NOP
; Aufruf von puts():
.text:0000002C move $t9, $v0
.text:00000030 jalr $t9
.text:00000034 or $at, $zero ; NOP
; GP vom lokalen Stack wiederherstellen:
.text:00000038 lw $gp, 0x20+var_10($fp)
; Setze das Register $2 ($V0) zu 0:
.text:0000003C move $v0, $zero
```
Interessanterweise kennt IDA das Anweisungspaar LUI/ADDIU und fasst diese zu einer einzigen Pseudoanweisung LA („Load Address“) zusammen (Zeile 15). Es ist auch zu sehen, dass diese Pseudoanweisung eine Größe von 8 Byte hat! Dies ist eine Pseudoanweisung (oder Makro) weil es sich hier nicht um eine echte MIPS-Anweisung handelt, sondern eher um einen handlichen Namen für ein Anweisungspaar.

Eine weitere Sache ist, dass IDA keine NOP-Anweisung kennt. Also ist in den Zeilen 22, 26 und 41 OR $AT, $ZERO. Im Wesentlichen führt diese Anweisung eine ODER-Operation auf die Inhalte des $AT-Register aus, welche 0 ist. Dies entspricht natürlich einer Idle-Anweisung. MIPS hat wie viele andere ISA keine separate NOP-Anweisung.

**Aufgabe des Stack Frames in diesem Beispiel**

Die Adresse dieser Zeichenkette ist in einem Register übergeben. Warum wird dennoch der lokale Stack vorbereitet? Der Grund dafür liegt in der Tatsache, dass die Werte des Register RA und GP wegen des Aufrufs von printf() irgendwo gesichert werden müssen und hier eben der lokale Stack dafür genutzt wird. Wenn dies eine German text placeholder wäre, bestünde die Möglichkeit den Funktionsepilog und -prolog wegzulassen, wie hier: 1.4.3 on page 8.

**German text placeholder GCC: in GDB laden**

Listing 1.34: sample GDB session

```
root@debian-mips:~# gcc hw.c -O3 -o hw
root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw
Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>: lui gp,0x42
0x00400644 <main+4>: addiu sp,sp,-32
0x00400648 <main+8>: addiu gp,gp,-30624
0x0040064c <main+12>: sw ra,28(sp)
0x00400650 <main+16>: sw gp,16(sp)
0x00400654 <main+20>: lw t9,-32716(gp)
0x00400658 <main+24>: lui a0,0x40
0x0040065c <main+28>: jalr t9
0x00400660 <main+32>: addiu a0,a0,2080
0x00400664 <main+36>: lw ra,28(sp)
0x00400668 <main+40>: move v0,zero
0x0040066c <main+44>: jr ra
0x00400670 <main+48>: addiu sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
```
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820: "hello, world"
(gdb)

1.5.5 Fazit
Der Hauptunterschied zwischen x86/ARM and x64/ARM64-Code ist das der Zeiger auf den String 64 Bit lang ist. Moderne CPUs haben eine 64-Bit-Architektur um Speicherkosten zu reduzieren und den höheren Bedarf aktueller Anwendungen erfüllen zu können. Es ist möglich sehr viel mehr Speicher in dem Computer zu verwenden als 32-Bit-Zeiger adressieren können. Aus diesem Grund sind alle Zeiger 64 Bit lang.

1.5.6 Übungen
• http://challenges.re/48
• http://challenges.re/49

1.6 Stack
Der Stack ist eine der fundamentalen Datenstrukturen in der Informatik.\footnote{45 wikipedia.org/wiki/Call_Stack} \footnote{46 German text placeholder} \footnote{47 LIFO!} Technisch betrachtet ist es ein Stapelspeicher innerhalb des Prozessspeichers der zusammen mit den ESP (x86), RSP (x64) oder dem SP (ARM) Register als ein Zeiger in diesem Speicherblock fungiert.

Die häufigsten Stack-Zugriffsinstruktionen sind die PUSH- und POP-Instruktionen (in beidem x86 und ARM Thumb-Modus). PUSH subtrahiert vom ESP/RSP/SP 4 Byte im 32-Bit Modus (oder 8 im 64-Bit Modus) und schreibt dann den Inhalt des Zeigers an die Adresse auf die von ESP/RSP/SP gezeigt wird.

POP ist die umgekehrte Operation: Die Daten des Zeigers für die Speicherregion auf die von SP gezeigt wird werden ausgelesen und die Inhalte in den Instruktionsoperanden geschrieben (oft ist das ein Register). Dann werden 4 (beziehungsweise 8) Byte zum Stapel-Zeiger addiert.

Nach der Stackallokation, zeigt der Stapel-Zeiger auf den Boden des Stacks. PUSH verringert den Stapel-Zeiger und POP erhöht ihn. Der Boden des Stacks ist eigentlich der Anfang der Speicherregion die für den Stack reserviert wurde. Das wirkt zunächst seltsam, aber so funktioniert es.

ARM unterstützt beides, aufsteigende und absteigende Stacks.

Zum Beispiel die STMFD/LDMFD und STMED/LDMED\footnote{48 Store Multiple Empty Descending (German text placeholder)}\footnote{49 Load Multiple Empty Descending (German text placeholder)} Instruktionen sind alle dafür gedacht mit einem absteigendem Stack zu arbeiten (wächst nach unten, fängt mit hohen Adressen an und entwickelt sich zu niedrigeren Adressen). Die STMFA/LDMFA\footnote{50 Store Multiple Full Ascending (German text placeholder)}\footnote{51 Load Multiple Full Ascending (German text placeholder)} und STMEA/LDMEA\footnote{52 Store Multiple Empty Ascending (German text placeholder)}\footnote{53 Load Multiple Empty Ascending (German text placeholder)} Instruktionen sind dazu gedacht mit einem aufsteigendem Stack zu arbeiten (wächst nach oben und fängt mit niedrigeren Adressen an und wächst nach oben).

1.6.1 Warum wächst der Stack nach unten?
Intuitiv, würden man annehmen das der Stack nach oben wechs z.B Richtung höherer Adressen, so wie bei jeder anderen Datenstruktur.

Der Grund das der Stack rückwärts wächst ist wohl historisch bedingt. Als Computer so groß waren das sie einen ganzen Raum beanspruchten haben war es einfach Speicher in zwei Sektionen zu unterteilen, einen Teil für den German text placeholder und einen Teil für den Stack. Sicher war zu dieser Zeit nicht bekannt wie groß der German text placeholder und der Stack wachsen würden, während der Programm Laufzeit, also war die Lösung die einfachste mögliche.

\footnote{45 wikipedia.org/wiki/Call_Stack} \footnote{46 German text placeholder} \footnote{47 LIFO!} \footnote{48 Store Multiple Empty Descending (German text placeholder)}\footnote{49 Load Multiple Empty Descending (German text placeholder)} \footnote{50 Store Multiple Full Ascending (German text placeholder)}\footnote{51 Load Multiple Full Ascending (German text placeholder)} \footnote{52 Store Multiple Empty Ascending (German text placeholder)}\footnote{53 Load Multiple Empty Ascending (German text placeholder)}\footnote{54 LIFO!}
In [D. M. Ritchie and K. Thompson, The UNIX Time Sharing System, (1974)] kannen wir folgendes lesen:


1.6.2 Für was wird der Stack benutzt?
1.6.3 Rückgabe Adresse der Funktion speichern

Wenn man eine Funktion mit der CALL Instruktion aufruft, wird die Adresse direkt nach der CALL Instruktion auf dem Stack gespeichert und der unbedingte jump wird ausgeführt.

Die CALL Instruktion ist äquivalent zu dem PUSH address_after_call / JMP operand Instruktions paar. 

RET ruft die Rückkehr Adresse vom Stack ab und springt zu dieser — was äquivalent zu einem POP tmp / JMP tmp Instruktions paar ist.

Den Stack zum überlaufen zu bringen ist recht einfach, einfach eine endlos rekursive Funktion Aufrufen:

```c
void f()
{
    f();
}
```

MSVC 2008 hat eine Erkennung für das Problem:

```c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

ss.cpp

c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, function will cause...
```

...aber der Compiler erzeugt den Code trotzdem:

```c
?f@@YAXXZ PROC ; f
    push ebp
    mov ebp, esp
```

54German text placeholder http://go.yurichev.com/17270
...Auch wenn wir die Compiler Optimierungen einschalten (/Ox Option) wird der optimierte Code nicht den Stack zum überlaufen bringen. Stattdessen wird der Code korrekt ausgeführt:

GCC 4.4.1 generiert vergleichbaren Code in beiden Fällen, jedoch ohne über das Overflow Problem zu warnen.

**ARM**

ARM Programme benutzen den Stack um Rücksprung Adressen zu speichern, aber anders. Wie bereits erwähnt in „Hallo, Welt!“ (1.5.3 on page 18), wird der RA Wert im LR (German text placeholder) gespeichert. Wenn nun eine andere Funktion aufgerufen werden muss und auf das LR Register zu greift, muss der aktuelle Wert im Register irgendwo gespeichert werden.

Normal wird der Wert im Funktion Prolog gespeichert.

Oft sieht man Instruktionen wie z.B PUSH R4-R7, LR zusammen mit dieser Instruktion im Epilog POP R4-R7, PC—Somit werden Werte die in den Funktionen benötigt werden auf dem Stack gespeichert, inklusive LR.

Wenn eine Funktion nie eine andere Funktion aufruft, nennt man das in der RISC Terminologie eine leaf Funktion. Als Konsequenz ergibt sich, das leaf Funktionen nicht das LR Register speichern (da sie es nicht modifizieren). Wenn solche Funktionen klein sind und nur eine geringe Anzahl an Registern benutzt, ist es möglicher das der Stack gar nicht benutzt wird. Es ist also möglich leaf Funktionen zu benuten ohne den Stack zurück zu greifen, die Ausführung ist hier schneller als auf älteren x86 Maschinen weil kein externer RAM für den Stack benutzt wird. Diese Eigenschaft kann nützlich sein wenn der Speicher für den Stack noch nicht alloziert oder verfügbar ist.

Ein paar Beispiele für leaf Funktionen:

1.9.3 on page 87, 1.9.3 on page 87, 1.250 on page 290, 1.266 on page 308, 1.19.5 on page 308, 1.160 on page 187, 1.158 on page 185, 1.177 on page 203.

**Funktion Argumente übergeben**

Der übliche weg Argumente in x86 zu übergeben ist die „cdecl“ Methode:

```assembly
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

Die Callee Funktionen bekommen ihre Argumente über den Stackpointer.

So werden die Argumente auf dem Stack gefunden, noch vor der Ausführung der ersten Instruktion der f() Funktion:

55Ironic hier
57Bis vor einer weile war es sehr teuer auf PDP-11 und VAX Maschinen die CALL Instruktion zu benuten; bis zu 50% der Rechenzeit wurde allein für diese Instruktion verschwendet, man hat dabei festgestellt das eine große Anzahl an kleinen Funktionen zu haben ein [German text placeholder] [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].
Für mehr Informationen über andere Aufrufs Konventionen siehe Sektion: (6.1 on page 448).

Übrigens, die callee Funktion hat keine Informationen wie viele Argumente übergeben wurden. C Funktionen mit einer variablen Anzahl an Argumenten (wie z.B printf()) errechnen die Zahl der Argumente anhand der Formatstring spezifizier-er (alle spezifizier-er die mit dem % beginnen).

Wenn wir etwas schreiben wie z.B:

```c
printf("%d %d %d", 1234);
```

printf() wird die Zahlen 1234 und zwei zufällige Werte ausgeben, welche direkt neben 1234 auf dem Stack lagen.

Das ist auch der Grund warum es nicht wichtig ist wie die main() Funktion definiert ist: Als main(), main(int argc, char *argv[]) oder main(int argc, char *argv[], char *envp[]).

Tatsächlich ruf der CRT-Code die main() Funktion um Grunde so auf:

```c
push envp
push argv
push argc
call main
...
```

Wenn man main() als main() Funktion ohne Argumente definiert, dann liegen sie trotzdem auf dem Stack auch wenn sie nicht benutzt werden. Wenn man main() als main(int argc, char *argv[]), definiert kann man auf die ersten beiden Argumente der Funktion zugreifen, das dritte bleibt aber weiterhin “Unsichtbar” für andere Funktionen. Es ist aber auch u.a möglich die Main Funktion als main(int argc) schreiben und sie wird noch immer funktionieren.

**Alternative Wege Argumente zu übergeben**

Es sollte bemerkt werden das nichts einen Programmierer dazu zwingt Argumente über den Stack zu übergeben. Das ist keine generelle Anforderung. Jemand könnte auch einfach eine andere Methode implementieren ohne den Stack überhaupt zu benutzen.

Ein ziemlich beliebter Weg Argumente zu übergeben unter Assembler Neulingen ist über globale Variablen wie z.B:

```asm
Listing 1.35: Assembly code
```

```asm
...  
mov X, 123  
mov Y, 456  
call do_something  
...  
X dd  
Y dd  

do_something proc near  
; take X  
; take Y  
; do something  
retn  
do_something endp
```

58Nicht zufällig im eigentlichen Sinne sondern eher unvorhersehbar: 1.6.5 on page 36
Aber diese Methode hat Nachteile: Die `do_something()` Funktion kann sich selbst nicht rekursiv aufrufen (aber auch keine andere Funktion), weil sie ihre eigenen Argumente löschen muss. Die gleiche Geschichte mit lokalen Variablen: Wenn die Werte in globalen Variablen gespeichert sind, kann die Funktion sich nicht selbst aufrufen. Und das bedeutet wiederum das die Funktion nicht thread-Safe ist. Eine Methode solche Informationen auf dem Stack zu speichern macht die Dinge einfacher— Der Stack kann so viele Funktion Argumente und/oder Werte speichern, so viel Speicher wie der Computer hat.


Auf MS-DOS gab es einen Weg Funktion Argumente über Register zu übergeben, zum Beispiel dies ist ein Stück Code einer veralteten 16-Bit MS-DOS “Hallo, Welt!” Funktion:

```
mov dx, msg ; Adresse der Nachricht
mov ah, 9 ; 9 bedeutet `print string'
int 21h ; DOS "syscall"

mov ah, 4ch ; `Terminiere Programm' Funktion
int 21h ; DOS "syscall"

msg db 'Hello, World!$
```

Diese Methode ist der 6.1.3 on page 449 Methode sehr ähnlich. Sie ähnelt aber auch der Methode wie man auf Linux (6.3.1 on page 462) und Windows syscalls ausführt.

Wenn eine MS-DOS Funktion einen Bool’schen Wert zurück gibt (z.B., Single Bit bedeutet ein Fehler ist aufgetreten), wird dafür das CF Flag benutzt.

Zum Beispiel:

```
mov ah, 3ch ; create file
lea dx, filename
mov cl, 1
int 21h
jc error
mov file_handle, ax
...
error: ...
```

Im Falle eines Fehlers, wird das CF Flag gesetzt. Anderenfalls wird ein handle für die neu erstellte Datei über AX zurück gegeben.


```
public Get386Stepping
Get386Stepping proc

call MultiplyTest ; Multiplikations Test durchführen
jnc short G3s00 ; wenn nc, ist muttest ok
mov ax, 0
ret

G3s00:  
call Check386B0 ; Prüfe das B0 stepping
jnc short G3s05 ; wenn nc, ist es B1/later
mov ax, 100h ; It is B0/earlier stepping
ret

G3s05:  
call Check386D01 ; Prüfe das D1 stepping
jc short G3s10 ; wenn c, just es NICHT NOT D1
mov ax, 301h ; Es ist das D1/later stepping
```

59Korrekt implementiert, hat jeder Thread seinen eigenen Stack und seine eigenen Argumente/Variablen
Local variable storage

Eine Funktion kann Platz für lokale Variablen allokieren, indem sie einfach den Stapel-Zeiger verkleinert, indem sie die niedrigste Adresse des Stacks verschiebt.

Dieser Weg ist ziemlich schnell, egal wie viele Variablen deffiniert werden. Es ist aber keine Anforderung, lokale Variablen auf dem Stack zu speichern. Man kann lokale Variablen speicher wo immer man will, aber traditionell speichert man sie auf dem Stack.

x86: alloca() Funktion

Es macht Sinn einen Blick auf die alloca() Funktion zu werfen gefunden werden. Diese Funktion arbeitet wie malloc(), nur dass sie Speicher direkt auf dem Stack bereit stellt.

Der allozierte Speicher Chunk muss nicht wieder mit free() freigegeben werden, weil der Funktions Epilog das ESP Register wieder in seinen ursprünglichen Zustand versetzt und der allozierte Speicher wird einfach verworfen. Es macht Sinn sich anzuschauen wie alloca() implementiert ist. Mit einfachen Begriffen erklärt, diese Funktion verschiebt ESP in Richtung des Stack ende mit der Anzahl der Bytes die alloziert werden müssen und setzt ESP als einen Zeiger auf den allozierten block.

Beispiel:

```c
#include <alloca.h> // GCC
#include <malloc.h> // MSVC
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
    #ifdef __GNUC__
        snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
    #else
        _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
    #endif
    puts (buf);
}
```

Die _snprintf() Funktion arbeitetet genau wie printf(), nur statt die Ergebnisse nach placeholder aus zu geben (bsp. auf dem Terminal oder Konsole), schreibt sie in den buf buffer. Die Funktion puts() kopiert den Inhalt aus buf nach placeholder. Sicher könnte man die beiden Funktionen

60In MSVC, kann die Funktions Implementierung in alloca16.asm und chkstk.asm in C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel
Aufrufe könnten durch einen `printf()` Aufruf ersetzt werden, aber wir sollten einen genaueren Blick auf die Benutzung kleiner Buffer anschauen.

**MSVC**

Compilierung mit MSVC 2010:

Listing 1.36: MSVC 2010

```
... 
  mov  eax, 600 ; 00000258H
  call __alloca_probe_16
  mov  esi, esp
  push 3
  push 2
  push 1
  push OFFSET $SG2672
  push 600 ; 00000258H
  push esi
  call __snprintf
  push esi
  call _puts
  add  esp, 28
...
```

Das einzige `alloca()` Argument wird über EAX übergeben (anstatt es erst auf den Stack zu pushen) \(^6\).

**GCC** + \textbf{German text placeholder}

GCC 4.4.1 macht das selbe, aber ohne externe Funktions aufrufe.

Listing 1.37: GCC 4.7.3

```
.LC0:
  .string "hi! %d, %d, %d\n"
f:
  push  ebp
  mov  ebp, esp
  push  ebx
  sub  esp, 660
  lea  ebx, [esp+39]
  and  ebx, -16 ; align pointer by 16-bit border
  mov  DWORD PTR [esp], ebx ; s
  mov  DWORD PTR [esp+20], 3
  mov  DWORD PTR [esp+16], 2
  mov  DWORD PTR [esp+12], 1
  mov  DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
  mov  DWORD PTR [esp+4], 600 ; maxlen
  call __snprintf
  mov  DWORD PTR [esp], ebx ; s
  call _puts
  mov  ebx, DWORD PTR [ebp-4]
  leave 
  ret
```

\(^6\)Das liegt daran, das `alloca()` Verhalten Compiler intrinsisch bestimmt (\cite[10.3 on page 512]{msdn}) im Gegensatz zu einer normalen Funktion. Einer der Gründe dafür das man braucht eine separate Funktion braucht. statt ein paar Code Instruktionen im Code, ist weil die \textbf{MSCV} \(^6\) `alloca()` Implementierung ebenfalls Code hat welcher aus dem gerade allozierten Speicher gelesen wird. Damit in Folge das Betriebssystem!\(^6\) physikalischen Speicher in dieser VM \(^6\) Region zu allozieren. Nach dem `alloca()` Aufruf, zeigt ESP auf den Block von 600 Bytes der nun als Speicher für das buf Array dienen kann.
GCC + German text placeholder

Nun der gleiche Code, aber in AT&T Syntax:

Listing 1.38: GCC 4.7.3

```
.LC0:
    .string "hi! %d, %d, %d\n"

f:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $660, %esp
    leal 39(%esp), %ebx
    andl $-16, %ebx
    movl %ebx, (%esp)
    movl $3, 20(%esp)
    movl $2, 16(%esp)
    movl $1, 12(%esp)
    movl $.LC0, 8(%esp)
    call _snprintf
    movl %ebx, (%esp)
    call puts
    movl -4(%ebp), %ebx
    leave
    ret
```

Der Code ist der gleiche wie im vorherigen listig.

Übrigens, `movl $3, 20(%esp)` in AT&T Syntax wird zu `mov DWORD PTR [esp+20], 3` in Intel-syntax. In der AT&T Syntax, sehen Register+Offset Formatierungen einer Adresse so aus: `offset(%register)`

(Windows) SEH

Automatisches deallokieren der Daten auf dem Stack

Vielleicht ist der Grund warum man lokale Variablen und SEH Einträge auf dem Stack speichert, weil sie beim verlassen der Funktion automatisch aufgeräumt werden. Man braucht dabei nur eine Instruktion um die Position des Stackpointers zu korrigieren (oftmals ist es die ADD Instruktion). Funktions Argumente, könnte man sagen werden auch am Ende der Funktion deallokiert. Im Kontrast dazu, alles was auf dem heap gespeichert wird muss explizit deallokiert werden.

1.6.4 Ein typisches Stack Layout

Ein typisches Stacklayout auf einer 32-Bit Umgebung sieht am Anfang der ausführung einer Funktion, noch bevor der ausführung der ersten Instruktion wie folgt aus:

| ESP-0x8C | lokale Variable#2, German text placeholder var_8 |
| ESP-0x8  | lokale Variable#1, German text placeholder var_4 |
| ESP-0x4  | German text placeholderEBP |
| ESP      | Rücksprungadresse |
| ESP+0x4  | German text placeholder#1, German text placeholder arg_0 |
| ESP+0x8  | German text placeholder#2, German text placeholder arg_4 |
| ESP+0x8C | German text placeholder#3, German text placeholder arg_8 |

1.6.5 Rauschen auf dem Stack

When one says that something seems random, what one usually means in practice is that one cannot see any regularities in it.

Stephen Wolfram, A New Kind of Science.

```c
#include <stdio.h>

void f1()
{
    int a=1, b=2, c=3;
};

void f2()
{
    int a, b, c;
    printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
    f1();
    f2();
};
```

Compilieren ...

Listing 1.39: German text placeholder MSVC 2010

```
SG2752 DB ' %d, %d, %d', 0AH, 00H

_c$ = -12 ; size = 4
_b$ = -8 ; size = 4
_a$ = -4 ; size = 4

_f1 PROC
    push ebp
    mov ebp, esp
    sub esp, 12
    mov DWORD PTR _a$[ebp], 1
    mov DWORD PTR _b$[ebp], 2
    mov DWORD PTR _c$[ebp], 3
    mov esp, ebp
    pop ebp
    ret 0

_f1 ENDP

_c$ = -12 ; size = 4
_b$ = -8 ; size = 4
_a$ = -4 ; size = 4

_f2 PROC
    push ebp
    mov ebp, esp
    sub esp, 12
    mov eax, DWORD PTR _c$[ebp]
    push eax
    mov ecx, DWORD PTR _b$[ebp]
    push ecx
    mov edx, DWORD PTR _a$[ebp]
    push edx
    push OFFSET $SG2752 ; '%d, %d, %d'
    call DWORD PTR __imp__printf
    add esp, 16
    mov esp, ebp
    pop ebp
    ret 0

_f2 ENDP

_main PROC
    push ebp
    mov ebp, esp
    call _f1
    call _f2
    xor eax, eax
    pop ebp
```

37
Hier wird sich der Compiler ein bisschen beschweren...

```plaintext
c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

st.c
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation. All rights reserved.
/out:st.exe
st.obj
```

Aber wenn wir das compilierte Programm laufen lassen...

```plaintext
c:\Polygon\c>st
1, 2, 3
```

sieh an! Wir haben keine Variablen gesetzt in f2(). Das sind „Geister“ Werte, welche noch immer auf dem Stack rumliegen.
Lasst uns das Beispiel in OllyDbg laden:

Abbildung 1.6: OllyDbg: f1()

Wenn f1() den Variablen \(a\), \(b\) und \(c\) ihre Werte zuordnet, wird ihre Adresse bei 0x1FF860 gespeichert und so weiter.
Und wenn f2() ausgeführt wird:

Abbildung 1.7: OllyDbg: f2()

... liegen $a$, $b$ und $c$ von f2() an den gleichen Adressen! Nichts hat bis jetzt Ihre Werte überschrieben und sie sind bisher unberührt geblieben. Also, damit diese seltsame Situation eintritt, müssen mehrere Funktionen nacheinander aufgerufen werden und SP muss gleich sein für jede Funktions Instruktion (z.B. die Funktionen haben die gleiche Anzahl an Argumenten). Dann werden die lokalen Variablen an den gleichen Positionen im Stack liegen. Zusammen fassend kann man sagen, alle Werte auf dem Stack (und Speicherzellen im allgemeinen) beinhalten Werte von vorhergehenden Funktions aufrufen. Diese Werte sind nicht zufällig im klassischem Sinn, eher unvorhersehbar. Es wäre wahrscheinlich möglich Teile des Stacks auf zu räumen vor jedem Funktions Aufruf, aber das wäre zu viel zusätzliche (und unnötige) Arbeit.

**MSVC 2013**

Das Beispiel wurde compiliert mit dem MSVC 2010 Compiler. Allerdings haben die Leser dieses Buch auch schon geschafft das Beispiel mit MSVC 2013 zu compilieren, sie haben es geschafft es zum laufen zu bringen und alle drei Nummern zu reversen.

```
c:\Polygon\c>st
3, 2, 1
```

Warum? Ich habe das Beispiel auch mit MSCV 2013 compiliert und habe folgendes beobachtet:

Listing 1.40: MSVC 2013

```
a$ = -12 ; size = 4
_b$ = -8 ; size = 4
_c$ = -4 ; size = 4
_f2 PROC
...
_f2 ENDP
_c$ = -12 ; size = 4
```

1.6.6 Übungen

• http://challenges.re/51
• http://challenges.re/52

1.7 German text placeholder

An dieser Stelle wird das Hallo, Welt! (1.5 on page 8)-Beispiel ein wenig erweitert, indem printf() in der main()-Funktion durch folgendes ersetzt wird:

```c
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
    return 0;
};
```

1.7.1 x86

x86: 3 Argumente

MSVC

Beim Kompilieren mit MSVC 2010 Express wird folgende Ausgabe erzeugt:

```asm
$SG3830 DB 'a=%d; b=%d; c=%d', 00H
...
push 3
push 2
push 1
push OFFSET $SG3830
call printf
add esp, 16 ; 00000010H
```

Fast die gleiche Ausgabe, allerdings sind jetzt die printf()-Argumente in umgekehrter Reihenfolge auf dem Stack hinterlegt. Das erste Argument kommt zuerst.

Die Variablen vom Typ int sind in einer 32-Bit-Umgebung 32 Bit breit, was 4 Byte entspricht.

In diesem Fall sind 4 Argumente vorhanden. $4 \times 4 = 16$ —diese benötigen exakt 16 Byte auf dem Stack einen 32-Bit-Zeiger auf eine Zeichenkette und 3 Zahlen des Typs int.

Wenn der Stapel-Zeiger (ESP-Register) nach einem Funktionsaufruf durch die Anweisung ADD ESP, X wiederhergestellt wird, wird die Anzahl Argumente oft einfach durch die Division von X durch 4 abgeleitet.
Natürlich ist dies eine Eigenheit der cdecl-Aufrufkonvention und nur für 32-Bit-Umgebungen gültig (siehe auch Abschnitt über Aufrufkonventionen (6.1 on page 448)).

In bestimmten Fällen in denen mehrere Funktionen direkt hintereinander beendet werden, kann der Compiler mehrere „ADD ESP, X“-Anweisungen in einer zusammenfassen:

```
push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

Hier ist ein Beispiel aus einer realen Applikation:

Listing 1.41: x86

```
.text:100113E7 push 3
.text:100113E9 call sub_100018B0 ; erwartet ein Argument (3)
.text:100113EE call sub_100019D0 ; erwartet kein Argument
.text:100113F3 call sub_10006A90 ; erwartet kein Argument
.text:100113F8 push 1
.text:100113FA call sub_100018B0 ; erwartet ein Argument (1)
.text:100113FF add esp, 8 ; Addiert zwei Argumente auf einmal auf den Stack
```
MSVC und OllyDbg


Jetzt muss die main()-Funktion gefunden werden, die sich ganz oben im Code befindet (MSVC allokiert die main()-Funktion ganz zu Beginn der Code-Sektion):

Abbildung 1.8: OllyDbg: Der Start der main()-Funktion

Um den CRT-Code zu überspringen (der hier nicht von Interesse ist) müssen folgende Schritte ausgeführt werden: Klicken auf die PUSH EBP-Anweisung, Drücken von F2 (Breakpoint setzen) und Drücken von F9 (Starten)
Sechsmaliges Drücken von F8 (German text placeholder) überspringt sechs Anweisungen:

Jetzt zeigt der PC auf die CALL printf-Anweisung. OllyDbg hebt, wie andere Debugger auch, den Wert des Registers hervor, welches sich geändert hat. Bei jedem Drücken von F8 ändert sich also EIP und der Wert wird in rot angezeigt. ESP ändert sich ebenfalls, weil die Werte der Argumente auf dem Stack gesichert werden.

Wo befinden sich die Werte auf dem Stack? Ein Blick auf das untere, rechte Fenster des Debuggers liefert die Antwort:

Es ist möglich einen Rechtsklick auf den Formatstring und anschließend auf „Follow in dump“ zu klicken. Der Formatstring wird in dem linken-unteren Debugger erscheinen, welches immer einen Teil des Speichers zeigt. Diese Speicherwerte können editiert werden. Es ist möglich den Formatstring zu verändern, was die Ausgabe dieses Beispiels ebenfalls verändern würde. In diesem Fall ist das nicht sehr nützlich, aber es könnte eine gute Übung sein um ein Gefühl dafür zu bekommen wie hier alles funktioniert.
Es erscheint die folgende Ausgabe auf der Konsole:

\[ a = 1; \quad b = 2; \quad c = 3 \]

Nachfolgend die Änderungen in den Registern und der Status des Stacks:

Das Register EAX enthält nun \( 0xD \) (13). Dies ist auch richtig, weil `printf()` die Anzahl der ausgegeben Zeichen zurück gibt. Der Wert von EIP hat sich verändert: er enthält nun die Adresse der Anweisung, die nach `CALL printf` kommt. Die Werte von ECX und EDX haben sich ebenfalls geändert. Offensichtlich nutzt die `printf()`-Funktion diese für interne Zwecke.


**Abbildung 1.11: OllyDbg nach der Ausführung von `printf()`**

Das Register EAX enthält nun \( 0xD \) (13). Dies ist auch richtig, weil `printf()` die Anzahl der ausgegebenen Zeichen zurück gibt. Der Wert von EIP hat sich verändert: er enthält nun die Adresse der Anweisung, die nach `CALL printf` kommt. Die Werte von ECX und EDX haben sich ebenfalls geändert. Offensichtlich nutzt die `printf()`-Funktion diese für interne Zwecke.

Durch erneutes Drücken von F8 wird die Anweisung ADD ESP, 10 ausgeführt:

Abbildung 1.12: OllyDbg: Nach der Ausführung von ADD ESP, 10

ESP hat sich geändert, aber die Werte sind immer noch auf dem Stack! Das liegt natürlich daran, dass es keine Notwendigkeit gibt diese Werte auf Null zu setzen oder Ähnliches. Alle oberhalb des Stack-Pointers (SP) ist Rauschen oder German text placeholder und hat keine Bedeutung. Es wäre zeitintensiv und unnötig die ungenutzten Stack-Einträge zurück zu setzen.

GCC

Nun wird das gleiche Programm unter Linux mit GCC 4.4.1 kompiliert und in IDA untersucht:

```
main proc near
var_10 = dword ptr -10h
var_C = dword ptr -0Ch
var_8 = dword ptr -8
var_4 = dword ptr -4

push ebp
mov ebp, esp
and esp, 0FFFFFF0h
sub esp, 10h
mov eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
mov [esp+10h+var_4], 3
mov [esp+10h+var_8], 2
mov [esp+10h+var_C], 1
mov [esp+10h+var_10], eax
call printf
mov eax, 0
leave
retn
endp
```

Es ist erkennbar, dass der Unterschied zwischen MSVC- und GCC-Code lediglich die Art ist, auf der die Argumente auf dem Stack gespeichert werden. Hier arbeitet GCC direkt mit dem Stack ohne Benutzung von PUSH/PUSH.

GCC und GDB
Nachfolgend das Beispiel mit **GDB** unter Linux. Die Option `-g` weist den Compiler an, Debugging-Informationen in die ausführbare Datei einzufügen.

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
... Reading symbols from /home/dennis/polygon/1...done.
```

Listing 1.42: Setzen eines Breakpoints auf `printf()`

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Ausführen. Im Quellcode ist keine `printf()`-Funktion, also kann GDB sie nicht anzeigen.

```
(gdb) run
Starting program: /home/dennis/polygon/1
```

```
Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29    printf.c: No such file or directory.
```

```
Ausgeben von 10 Stack-Einträgen. Die Spalte ganz links enthält die Adressen auf dem Stack.
```

```
(gdb) x/10w $esp
0xbfffffff1c: 0x0804844a 0x080484f0 0x00000001 0x00000002
0xbfffffff12c: 0x00000003 0x08048460 0x00000000 0x00000000
0xbfffffff13c: 0xb7e29905 0x00000001
```

Das allererste Element ist RA (0x0804844a). Dies kann durch das Disassemblieren des Speichers an dieser Stelle überprüft werden:

```
(gdb) x/5i 0x0804844a
  0x804844a <main+45>: mov $0x0,%eax
  0x804844f <main+50>: leave
  0x8048450 <main+51>: ret
  0x8048451: xchg %ax,%ax
  0x8048453: xchg %ax,%ax
```

Die beiden XCHG-Anweisungen sind Idle-Anweisungen, analog zu NOP. Das zweite Element (0x080484f0) ist die Adresse des Formatstrings:

```
(gdb) x/s 0x080484f0
0x80484f0: "a=%d; b=%d; c=%d"
```

Die nächsten drei Elemente (1, 2, 3) sind die `printf()`-Argumente. Der Rest der Elemente kann lediglich „garbage“ auf dem Stack sein, oder auch Werte von anderen Funktionen, deren lokale Variablen usw. An dieser Stelle können sie ignoriert werden.

Ausführen von „finish“. Dieses Kommando führt dazu, dass GDB „alle Anweisungen bis zum Ende der Funktion ausführt“. In diesem Fall: Ausführen bis zum Ende von `printf()`.

---

65 GNU Debugger

47
GDB zeigt was printf() in EAX zurück gibt (13). Die ist die Anzahl der ausgegebenen Zeichen, genau wie im OllyDbg-Beispiel.

Sichtbar ist auch „return 0;“ und die Information das dieser Ausdruck in der Datei 1.c in Zeile 6 ist. 1.c befindet sich in aktuellen Verzeichnis und GDB kann die Zeichenkette dort finden. Wie erkennt GDB welche C-Zeile gerade ausgeführt wird? Dies kommt von der Tatsache, dass der Compiler beim Generieren der Debugging-Informationen auch eine Tabelle mit Verweisen zwischen dem Quellcode-Zeilen und den Anweisungsadressen anlegt. GDB ist also ein Debugger auf Quellcode-Ebene.

Schauen wir uns die Register an. 13 in EAX:

Nachfolgend das Disassemblieren der aktuellen Anweisung. Der Pfeil zeigt auf die nächste Anweisung die ausgeführt wird.

GDB nutzt standardmäßig den AT&T-Syntax. Es ist aber möglich auf den Intel-Syntax zu wechseln:
Ausführen der nächsten Anweisung. **GDB** zeigt Ende-Klammern, welche den Block schließt.

```c
(gdb) step
7  
```

Das Ansehen der Register nach der `MOV EAX, 0`-Anweisung zeigt, dass das EAX-Register an dieser Stelle Null ist.

```c
(gdb) info registers
eax  0x0  0
ecx  0x0  0
edx  0x0  0
ebx  0xb7fc0000 -1208221696
esp  0xbffff120 0xbffff120
ebp  0xbffff138 0xbffff138
esi  0x0  0
edi  0x0  0
eip  0x804844f 0x804844f <main+50>
...
```

### 1.7.2 ARM

### 1.7.3 Fazit

Hier ist der grobe Aufbau der Aufruffunktion:

```c
Listing 1.43: x86
...
PUSH Drittes Argument
PUSH Zweites Argument
PUSH Erstes Argument
CALL Funktion
; gegebenenfalls den Stackpointer modifizieren
```

```c
Listing 1.44: x64 (MSVC)
MOV RCX, Erstes Argument
MOV RDX, Zweites Argument
MOV R8, Drittes Argument
MOV R9, Viertes Argument
...
PUSH fünftes, sechstes Argument, usw. (falls notwendig)
CALL Funktion
; gegebenenfalls den Stackpointer modifizieren
```

```c
Listing 1.45: x64 (GCC)
MOV RDI, Erstes Argument
MOV RSI, Zweites Argument
MOV RDX, Drittes Argument
MOV RCX, Viertes Argument
MOV R8, Fünftes Argument
MOV R9, Sechstes Argument
...
PUSH Siebtes, Achten Argument, usw. (falls notwendig)
CALL Funktion
; gegebenenfalls den Stackpointer modifizieren
```
1.7.4 Übrigens…

Übrigens ist der Unterschied der Art der Argumenten Übergabe in x86, x64, fastcall, ARM und MIPS eine
gute Darstellung der Tatsache, dass die CPU nicht weiß wie die Argumente an die Funktion übergeben
werden. Es ist auch möglich einen hypothetischen Compiler zu erstellen, der die Möglichkeit hat Argumente
mittels einer speziellen Struktur, ohne den Stack an die Funktionen zu übergeben.

MIPS $A0 ...$A3-Register sind aus Bequemlichkeitsgründen auf diese Weise beschriftet (O32 Aufrufkonvention).
Programmierer können auch andere Register (vielleicht außer $ZERO) nutzen um Daten zu übergeben oder
eine andere Aufrufkonvention zu nutzen.

Die CPU hatte jedoch keinerlei Kenntnisse über die Aufrufkonvention.

Man sieht hier auch wie Neulinge der Assemblersprache Argumente an andere Funktionen übergeben: in
der Regel per Register ohne explizite Reihenfolge oder globale Variablen. Natürlich funktioniert das ebenso
gut.

1.8 scanf()

1.8.1 Ein einfaches Beispiel

```c
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");
```
Es ist nicht ratsam scanf() heutzutage noch für User Interaktionen zu verwenden. Aber dennoch können wir hier die Übergabe eines Pointers an eine Variable vom Typ int betrachten.

**Pointer**


Wenn die aufgerufene Funktion außerdem das große Array oder Struct verändern muss und das gesamte Object zurückgeben muss, ist die Situation beinahe absurd. Das einfachste ist also die Adresse eines Arrays oder Struts an die aufgerufene Funktion zu übergeben und sie dann die notwendigen Veränderungen durchführen zu lassen.

Ein Pointer ist in C/C++ nichts anderes als die Adresse einer Speicherstelle.

In x86 wird die Adresse als 32-Bit-Zahl dargestellt, d.h. sie benötigt 4 Byte, während in x86-64 eine Darstellung durch 64 Bit (d.h. 8 Byte) erfolgt. Dies ist übrigens der Grund dafür, dass einige Leute den Wechsel zu x86-64 ablehnen–alle Pointer in der x64-Architektur erfordern doppelt soviel Speicherplatz, inklusive Speicher in Cache, der ein sehr teurer Speicher ist.

Es ist möglich lediglich mit untypisierten Pointern zu arbeiten, wenn man ein wenig zusätzlichen Aufwand betreibt; z.B. in der Standard-C-Funktion memcpy(), die einen Datenblock von einer Speicherstelle zu einer anderen kopiert, werden zwei Pointer vom Typ void* als Argumente verwendet, da es nicht vorhersagbar ist, welchen Datentyp die Funktion kopieren soll. Datentypen sind hier nicht wichtig, entscheidend ist hier nur die Größe des Speicherblocks.

Pointer werden außerdem häufig verwendet, wenn eine Funktion mehr als einen Wert zurückgeben muss. (Darauf kommen wir später in (1.10 on page 90) zurück.)

Die Funktion scanf() ist solch ein Fall: Neben der Tatsache, dass die Funktion angeben muss wie viele Werte erfolgreich gelesen wurden, muss sie auch alle diese Werte zurückliefern.

In C/C++ wird der Pointertyp nur für Typüberprüfungen zur Compilezeit benötigt.

Intern steckt im kompilierten Code keinerlei Information über die Typen der enthaltenen Pointer.

**x86**

**MSVC**

Den folgenden Code erhalten wir nach dem Kompilieren mit MSVC 2010:

```
CONSTR SEGMENT
$$SG3831 DB 'Enter X:', 0aH, 00H
$$SG3832 DB '%d', 00H
$$SG3833 DB 'You entered %d...', 0aH, 00H
CONST   ENDS
PUBLIC _main
EXTRN _scanf:PROC
EXTRN _printf:PROC
; Compiler Flags der Funktion: /Odtp
_TEXT SEGMENT
_x$ = -4 ; size = 4
_main PROC
    push ebp
    mov ebp, esp
    push ecx
    push OFFSET $$SG3831 ; 'Enter X:'
call _printf
    add esp, 4
    lea eax, DWORD PTR _x$[ebp]
push eax
```
x ist eine lokale Variable.

Gemäß dem C/C++-Standard darf diese nur innerhalb dieser Funktion sichtbar sein und nicht aus einem anderen, äußeren Scope. Traditionell werden lokale Variablen auf dem Stack gespeichert. Es gibt möglicherweise andere Wege sie anzulegen, aber in x86 geschieht es auf diese Weise.

Das Ziel des Befehls direkt nach dem Funktionsprolog, PUSH ECX, ist es nicht, den Status von ECX zu sichern (man beachte, dass Fehlen eines entsprechenden POP ECX im Funktionsepilog). Tatsächlich reserviert der Befehl 4 Byte auf dem Stack, um die Variable x speichern zu können.

Auf x wird mithilfe des _x$ Makros (es entspricht -4) und des EBP Registers, das auf den aktuellen Stack Frame zeigt, zugegriffen. Während der Dauer der Funktionsausführung zeigt EBP auf den aktuellen Stack Frame, wodurch mittels EBP+offset auf lokalen Variablen und Funktionsargumente zugegriffen werden kann.

x is to be accessed with the assistance of the _x$ macro (it equals to -4) and the EBP register pointing to the current frame.

Es ist auch möglich, das ESP Register zu diesem Zweck zu verwenden, aber dies ist unüblich, da es sich häufig verändert. Der Wert von EBP kann als eingefrorener Wert des Wertes von ESP zu Beginn der Funktionsausführung verstanden werden.

It is also possible to use ESP for the same purpose, although that is not very convenient since it changes frequently. The value of the EBP could be perceived as a frozen state of the value in ESP at the start of the function’s execution.

Hier ist ein typisches Layour eines Stack Frames in einer 32-Bit-Umgebung:

<table>
<thead>
<tr>
<th>EBP-8</th>
<th>local variable #2, German text placeholder var_8</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBP-4</td>
<td>local variable #1, German text placeholder var_4</td>
</tr>
<tr>
<td>EBP</td>
<td>saved value of EBP</td>
</tr>
<tr>
<td>EBP+4</td>
<td>return address</td>
</tr>
<tr>
<td>EBP+8</td>
<td>German text placeholder #1, German text placeholder arg_0</td>
</tr>
<tr>
<td>EBP+0xC</td>
<td>German text placeholder #2, German text placeholder arg_4</td>
</tr>
<tr>
<td>EBP+0x10</td>
<td>German text placeholder #3, German text placeholder arg_8</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Die Funktion scanf() in unserem Beispiel hat zwei Argumente.

Das erste ist ein Pointer auf den String %d und das zweite ist die Adresse der Variablen x.

Zunächst wird die Adresse der Variablen x durch den Befehl lea eax, DWORD PTR _x$[ebp] in das EAX Register geladen.

LEA steht für load effective address und wird häufig benutzt, um eine Adresse zu erstellen (.16 on page 534). In diesem Fall speichert LEA einfach die Summe des EBP Registers und des _$ Makros im Register EAX. Dies entspricht dem Befehl lea eax, [ebp-4].

Es wird also 4 von Wert in EBP abgezogen und das Ergebnis in das Register EAX geladen. Danach wird der Wert in EAX auf dem Stack abgelegt und scanf() wird aufgerufen.

Anschließend wird printf() mit einem Argument aufgerufen--einen Pointer auf den String: You entered %d...

\n
Das zweite Argument wird mit `mov ecx, [ebp-4]` vorbereitet. Dieser Befehl speichert den Wert der Variablen \( x \) (nicht seine Adresse) im Register ECX.

Schließlich wird der Wert in ECX auf dem Stack gespeichert und das letzte `printf()` wird aufgerufen.
MSVC + OllyDbg

Schauen wir uns diese Beispiel in OllyDbg an. Wir laden es und drücken F8 (set a breakpoint), dann F9 (Run). Der Breakpoint wird ausgelöst, wenn die Funktion main() beginnt.

Wir klicken auf den ersten Befehl (PUSH EB0), drücken F2 (setabreakpoint), dann F9 (Run). Der Breakpoint wird ausgelöst, wenn die Funktion main() beginnt.

Verfolgen wir den Ablauf bis zu der Stelle, an der die Adresse der Variablen x berechnet wird:


Wir machen einen Rechtsklick auf EAX in Registerfenster und wählen „Follow in stack“.

Diese Adresse wird im Stackfenster erscheinen. Der rote Pfeil wurde nachträglich hinzugefügt; er zeigt auf die Variable im lokalen Stack. Im Moment enthält diese Speicherstelle Zufallswerte (0x6E494714). Jetzt wird mithilfe des PUSH Befehls die Adresse dieses Stackelements auf demselben Stack an der folgenden Position gespeichert. Verfolgen wir den Ablauf mit F8 bis die Ausführung von scanf() abgeschlossen ist. Während der Ausführung von scanf() geben wir beispielsweise 123 in der Konsole ein:

Enter X:
123
scanf() ist bereits beendet:

Abbildung 1.14: OllyDbg: scanf() wurde ausgeführt

scanf() liefert 1 im EAX Register zurück, was aussagt, dass die Funktion einen Wert erfolgreich eingelesen hat. Wenn wir wiederum auf das zugehörige Stackelement für die lokale Variable schauen, enthält diese nun den Wert 0x7B (dez. 123).
Im weiteren Verlauf wird dieser Wert vom Stack in das ECX Register kopiert und an printf() übergeben:

**Abbildung 1.15:** OllyDbg: Wert für Übergabe an printf() vorbereiten.

**GCC**

Kompilieren wir diesen Code mit GCC 4.4.1 unter Linux:

```c
main proc near
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_4  = dword ptr -4

push    ebp
mov  ebp, esp
and  esp, 0FFFFFF0h
sub   esp, 20h
mov  [esp+20h+var_20], offset aEnterX ; "Enter X:"
call   _puts
mov  eax, offset aD ; "%d"
lea   edx, [esp+20h+var_4]
mov  [esp+20h+var_1C], edx
mov  [esp+20h+var_20], eax
call   __isoc99_scanf
mov  edx, [esp+20h+var_4]
mov  eax, offset aYouEnteredD___ ; "You entered %d...
mov  [esp+20h+var_1C], edx
mov  [esp+20h+var_20], eax
call   _printf
mov  eax, 0
leave
retn

main endp
```

GCC ersetzt den Aufruf von printf() durch einen Aufruf von puts(). Der Grund hierfür wurde bereits in (1.5.3 on page 21) erklärt.

Genau wie im MSVC Beispiel werden die Argumente mithilfe des Befehls MOV auf dem Stack abgelegt.

**By the way**

Dieses einfache Beispiel ist übrigens eine Demonstration der Tatsache, dass der Compiler eine Liste von Ausdrücken in einem C/C++-Block in eine sequentielle Liste von Befehlen übersetzt. Es gibt nichts
zwischen zwei C/C++-Anweisungen und genauso verhält es sich auch im Maschinencode. Der Control Flow geht von einem Ausdruck direkt an den folgenden über.

**x64**

Hier zeigt sich ein ähnliches Bild mit dem Unterschied, dass die Register anstelle des Stacks für die Übergabe der Funktionsargumente verwendet werden.

**MSVC**

Listing 1.49: MSVC 2012 x64

```assembly
_DATA SEGMENT
$SG1289 DB 'Enter X:', 0AH, 00H
$SG1291 DB '%d', 00H
$SG1292 DB 'You entered %d...', 0AH, 00H
_DATA ENDS

_TEXT SEGMENT
x$ = 32
main PROC

$LN3:
    sub    rsp, 56
    lea    rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
    call   printf
    lea    rdx, QWORD PTR x$[rsp]
    lea    rcx, OFFSET FLAT:$SG1291 ; '%d'
    call   scanf
    mov    edx, DWORD PTR x$[rsp]
    lea    rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
    call   printf

; return 0
xor    eax, eax
add    rsp, 56
ret 0

main ENDP

_TEXT ENDS
```

**GCC**

Listing 1.50: German text placeholder GCC 4.4.6 x64

```assembly
.LC0:   .string "Enter X:"
.LC1:   .string "%d"
.LC2:   .string "You entered %d...

main:
    sub    rsp, 24
    mov    edi, OFFSET FLAT:.LC0 ; "Enter X:"
    call   puts
    lea    rsi, [rsp+12]
    mov    edi, OFFSET FLAT:.LC1 ; "%d"
    xor    eax, eax
    call   __isoc99_scanf
    mov    esi, DWORD PTR [rsp+12]
    mov    edi, OFFSET FLAT:.LC2 ; "You entered %d...
    xor    eax, eax
    call   printf

; return 0
xor    eax, eax
add    rsp, 24
ret
```

57
Damit scanf() Elemente einlesen kann, benötigt die Funktion einen Parameter—einen Pointer vom Typ \textit{int}. \textit{int} hat die Größe 32 Bit, wir benötigen also 4 Byte, um den Wert im Speicher abzulegen, und passt daher genau in ein 32-Bit-Register. Auf dem Stack wird Platz für die lokale Variable \textit{x} reserviert und IDA bezeichnet diese Variable mit \textit{var\_8}. Eigentlich ist aber an dieser Stelle gar nicht notwendig, Platz auf dem Stack zu reservieren, da \textit{SP} (Stapel-Zeiger) bereits auf die Adresse zeigt und auch direkt verwendet werden kann.

Der Wert von \textit{SP} wird also in das R1 Register kopiert und zusammen mit dem Formatierungsstring an scanf() übergeben.

Später wird mithilfe des LDR Befehls dieser Wert vom Stack in das R1 Register verschoben um an printf() übergeben werden zu können.

**ARM64**

```
Listing 1.51: German text placeholder GCC 4.9.1 ARM64

1. LC0:
2. .string "Enter X:"
3. LC1:
4. .string "%d"
5. LC2:
6. .string "You entered %d...
"
7. scanf_main:
8. ; subtrahiere 32 von SP, speichere dann FP und LR im Stack Frame:
9.   stp x29, x30, [sp, -32]!
10. ; setze Stack Frame (FP=SP)
11.   add x29, sp, 0
12. ; ladens Pointer auf den "Enter X:" String:
13.   adrp x0, .LC0
14.   add x0, x0, :lo12:.LC0
15. ; X0=Pointer auf den "Enter X:" String
16. ; print it:
17.   bl puts
18. ; ladens Pointer auf den "%d" String:
19.   adrp x0, .LC1
20.   add x0, x0, :lo12:.LC1
21. ; finde Platz im Stack Frame für die Variable "x" (X1=FP+28):
22.   add x1, x29, 28
23. ; X1=Adresse der Variablen "x"
24. ; übergeben die Adresse an scanf() und rufe auf:
25.   bl __isoc99_scanf
26. ; ladens 32-Bit-Wert aus der Variable in den Stack Frame:
27.   ldr w1, [x29,28]
28. ; W1=x
29. ; ladens Pointer auf den "You entered %d...
"
30. ; printf() nimmt den Textstring aus X0 und die Variable "x" aus X1 (oder W1)
31.   adrp x0, .LC2
```
Im Stack Frame werden 32 Byte reserviert, was deutlich mehr als benötigt ist. Vielleicht handelt es sich um eine Frage des Aligning (dt. Angleichens) von Speicheradressen. Der interessanteste Teil ist, im Stack Frame einen Platz für die Variable \( x \) zu finden (Zeile 22). Warum 28? Irgendwie hat der Compiler entschieden die Variable am Ende des Stack Frames anstatt an dessen Beginn abzulegen. Die Adresse wird an scanf() übergeben; diese Funktion speichert den Userinput an der genannten Adresse im Speicher. Es handelt sich hier um einen 32-Bit-Wert vom Typ int. Der Wert wird in Zeile 27 abgeholt und dann an printf() übergeben.

**MIPS**

Auf dem lokalen Stack wird Platz für die Variable \( x \) reserviert und als $sp+24 referenziert. Die Adresse wird an scanf() übergeben und der Userinput wird mithilfe des Befehls LW („Load Word“) geladen und dann an printf() übergeben.

Listing 1.52: *German text placeholder GCC 4.4.5 (German text placeholder)*

```mips
$LC0: 
    .ascii "Enter X:\000"
$LC1: 
    .ascii "%d\000"
$LC2: 
    .ascii "You entered %d...\12\000"
main:
    ; Funktionsprolog:
    lui $28, hi(__gnu_local_gp)
    addiu $sp, $sp, -40
    addiu $28, $28, lo(__gnu_local_gp)
    sw $31, 36($sp)
    ; Aufruf von puts():
    lw $25, call16.puts($28)
    lui $4, hi($LC0)
    jalr $25
    addiu $4, $4, lo($LC0); branch delay slot
    ; Aufruf von scanf():
    lw $28, 16($sp)
    lui $4, hi($LC1)
    lw $25, call16(__isoc99_scanf($28)
    ; setze 2. Argument von scanf(), $a1=$sp+24:
    addiu $5, $sp, 24
    jalr $25
    addiu $4, $4, lo($LC1); branch delay slot
    ; Aufruf von printf():
    lw $28, 16($sp)
    ; setze 2. Argument von printf(),
    ; lade Wort nach Adresse $sp+24:
    lw $5, 24($sp)
    lw $25, call16(printf($28)
    lui $4, hi($LC2)
    jalr $25
    addiu $4, $4, lo($LC2); branch delay slot
    ; Funktionsepilog:
    lw $31, 36($sp)
    ; setze Rückgabewert auf 0:
    move $2, $0
    ; return:
    j $31
    addiu $sp, $sp, 40; branch delay slot
```
Listing 1.53: IDA stellt das Stack Layout wie folgt dar:

```assembly
.text:00000000 main:
.text:00000000 var_18 = -0x18
.text:00000000 var_10 = -0x10
.text:00000000 var_4 = -4
;
Funktionsprolog:
.text:00000000 lui $gp, (__gnu_local_gp >> 16)
.text:00000004 addiu $sp, -0x28
.text:00000008 la $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C sw $ra, 0x28+var_4($sp)
.text:00000010 sw $gp, 0x28+var_18($sp)
;
Aufruf von puts():
.text:00000014 lw $t9, (puts & 0xFFFF)($gp)
.text:00000018 lui $a0, ($LC0 >> 16) # "Enter X:"
.text:0000001C jalr $t9
.text:00000020 la $a0, ($LC0 & 0xFFFF) # "Enter X:" ; branch delay slot
;
Aufruf von scanf():
.text:00000024 lw $gp, 0x28+var_18($sp)
.text:00000028 lui $a0, ($LC1 >> 16) # "%d"
.text:0000002C lw $t9, (__isoc99_scanf & 0xFFFF)($gp)
; setze 2. Argument von scanf(), $a1=$sp+24:
.text:00000030 addiu $a1, $sp, 0x28+var_10
.text:00000034 jalr $t9 ; branch delay slot
.text:00000038 la $a0, ($LC1 & 0xFFFF) # "%d"
;
Aufruf von printf():
.text:0000003C lw $gp, 0x28+var_18($sp)
; setze 2. Argument von printf(),
; lade Word nach Adresse $sp+24:
.text:00000040 lw $a1, 0x28+var_10($sp)
.text:00000044 lw $t9, (printf & 0xFFFF)($gp)
.text:00000048 lui $a0, ($LC2 >> 16) # "You entered %d...
.text:0000004C jalr $t9
.text:00000050 la $a0, ($LC2 & 0xFFFF) # "You entered %d...\n" ; branch delay slot
;
Funktionsepilog:
.text:00000054 lw $ra, 0x28+var_4($sp)
; setze Rückgabewert auf 0:
.text:00000058 move $v0, $zero
; return:
.text:0000005C jr $ra
.text:00000060 addiu $sp, 0x28 ; branch delay slot
```

### 1.8.2 Häufiger Fehler

Ein häufiger (Tipp)-Fehler besteht darin, den Wert von `x` anstatt eines Pointers auf `x` zu übergeben.

```c
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");
    scanf ("%d", x); // BUG
    printf ("You entered %d...\n", x);
    return 0;
}
```

Was geschieht hier? `x` ist nicht uninitialisiert und enthält Zufallswerte vom lokalen Stack. Wenn `scanf()` aufgerufen wird, nimmt es den eingegebenen String vom Benutzer, wandelt ihn in eine Zahl um und versucht ihn nach `x` zu schreiben, wobei `x` wie eine Speicheradresse behandelt wird. Aber hier liegen
Zufallswerte vor, sodass scanf() versucht an eine zufällige Speicherstelle zu schreiben. Höchstwahrscheinlich wird der Prozess dadurch abstürzen.

Bemerkenswert ist, dass manche CRT-Bibliotheken im Debug Build gut erkennbare Muster in den gerade reservierten Speicher schreiben, wie z.B. 0xCCCCCCCC oder 0xBADF00D usw. In diesem Fall könnte x den Wert 0xCCCCDCCCCC enthalten und scanf() würde versuchen in die Adresse 0xCCCCDCCCCC zu schreiben. Wenn man nun bemerkt, dass irgendein Code im Prozess in die Adresse 0xCCCCDCCCCC schreiben möchte, weiß man, dass eine uninitialisierte Variable (oder ein Pointer) verwendet werden. Dies ist besser als wenn der frisch reservierte Speicher einfach gelöscht würde.

1.8.3 Globale Variablen

Was passiert, wenn die x-Variable aus dem letzten Beispiel nicht lokal sondern global ist? In dem Fall wäre sie von jeder Stelle aus zugreifbar, nicht nur aus dem Funktions-Rumpf. Globale Variablen gelten als

```
#include <stdio.h>

// x ist jetzt eine globale Variable
int x;

int main()
{
    printf ("Wert für x:\n");
    scanf ("%d", &x);
    printf ("Sie haben %d... eingegeben\n", x);
    return 0;
}
```

MSVC: x86

```
_DATA SEGMENT
COMM_x:DWORD
$SG2456  DB 'Enter X:', 0AH, 00H
$SG2457  DB ' %d', 00H
$SG2458  DB 'You entered %d... ', 0AH, 00H
_DATA ENDS
PUBLIC _main
EXTERN _scanf:PROC
EXTERN _printf:PROC
; Function compile flags: /Odtp
_TEXTURE SEGMENT
_main PROC
    push ebp
    mov ebp, esp
    push OFFSET $SG2456
    call _printf
    push OFFSET _x
    push OFFSET $SG2457
    call _scanf
    add esp, 8
    mov eax, DWORD PTR _x
    push eax
    push OFFSET $SG2458
    call _printf
    add esp, 8
    xor eax, eax
    pop ebp
    ret 0
_main ENDP
_TEXTURE ENDS
```

In diesem Falle wird die Variable x im _DATA Segment definiert und kein Speicher auf dem lokalen Stack wird reserviert. Es erfolgt ein direkter Zugriff ohne Umweg über den Stack. Nicht initialisierte globale
Variablen verbrauchen keinen Platz in der ausführbaren Datei (und wirklich: warum sollte man Speicher reservieren, wenn die Variable auf Anfang auf 0 gesetzt wird?), aber wenn jemand auf ihre Adresse zugreift, wird das BS einen Block Nullen reservieren\(^{66}\).

Weisen wir nun der Variablen ausdrücklich einen Wert zu:

```c
int x=10; // Defaultwert
```

Wir erhalten:

```plaintext
_DATA SEGMENT
_x DD 0AH
...
```

Wir sehen hier einen Wert 0xA vom Typ DWORD (DD steht für DWORD = 32 Bit) für diese Variable.

Wenn man die kompilierte .exe in IDA öffnet, sieht man, dass die Variable \(x\) am Anfang des _DATA Segments angelegt wird und danach sehen wir Textstrings.

Here we see a value 0xA of DWORD type (DD stands for DWORD = 32 bit) for this variable.

Wenn man die kompilierte .exe aus dem vorangegangenen Beispiel in IDA öffnet, in dem der Wert von \(x\) nicht gesetzt wurde, sieht man etwa das Folgende:

```
Listing 1.54: IDA
```

```
.data:0040FA80 _x dd ? ; DATA XREF: _main+10
.data:0040FA80 ........ ; _main+22
.data:0040FA84 dword_40FA84 dd ? ; DATA XREF: _memset+1E
.data:0040FA84 ........ ; unknown libname 1+28
.data:0040FA88 dword_40FA88 dd ? ; DATA XREF: ___sbh_find_block+5
.data:0040FA88 ........ ; ___sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem dd ? ; DATA XREF: ___sbh_find_block+B
.data:0040FA8C ........ ; ___sbh_free_block+2CA
.data:0040FA90 dword_40FA90 dd ? ; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90 ........ ; __calloc_impl+72
.data:0040FA94 dword_40FA94 dd ? ; DATA XREF: ___sbh_free_block+2FE
```


\(^{66}\)That is how a VM behaves
MSVC: x86 + OllyDbg

Hier sehen die Dinge noch einfacher aus:

Abbildung 1.16: OllyDbg: nach Ausführung von `scanf()`


Warum ist das erste Byte `7B`? Logisch gedacht müsste dort `00 00 00 7B` sein. Der Grund dafür ist die sogenannte Endianness und x86 verwendet little Endian. Dies bedeutet, dass das niedrigwertigste Byte zuerst und das höchswertigste zuletzt geschrieben werden. Für mehr Informationen dazu siehe: ?? on page ?? Zurück zu Beispiel: der 32-Bit-Wert wird von dieser Speicheradresse nach EAX geladen und an `printf()` übergeben.

Die Speicheradresse von `x` ist `0x00C53394`. 
In OllyDbg können wir die Speicherzuordnung des Prozesses nachvollziehen (Alt-M) und wir erkennen, dass sich diese Adresse innerhalb des `.data` PE-Segments von unserem Programm befindet:

![OllyDbg: Speicherzuordnung](image)

**Abbildung 1.17: OllyDbg: Speicherzuordnung**

**GCC: x86**

Unter Linux zeigt sich ein ähnliches Bild, mit dem Unterschied, dass die nicht initialisierten Variablen im `_bss` Segment gehalten werden. In der ELF-Datei hat dieses Segment die folgenden Attribute:

```plaintext
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

Wenn man nun der Variablen einen Wert zuweist, z.B. 10, muss diese im `_data` Segment abgelegt werden, das die folgenden Attribute aufweist:

```plaintext
; Segment type: Pure data
; Segment permissions: Read/Write
```

**MSVC: x64**

Listing 1.55: MSVC 2012 x64

```plaintext
DATA SEGMENT
COMM x:DWORD
SG2924 DB 'Enter X:', 0AH, 00H
SG2925 DB '%d', 00H
SG2926 DB 'You entered %d...', 0AH, 00H
```

---

67 German text placeholder
main PROC
$LN3:
    sub    rsp, 40
    lea    rcx, OFFSET FLAT:$SG2924; 'Enter X:'
    call   printf
    lea    rdx, OFFSET FLAT:x
    lea    rcx, OFFSET FLAT:$SG2925; '%d'
    call   scanf
    mov    edx, DWORD PTR x
    lea    rcx, OFFSET FLAT:$SG2926; 'You entered %d...'
    call   printf
    ; return 0
    xor    eax, eax
    add    rsp, 40
    ret
main ENDP

Der Code ist beinahe der gleich wie in x86. Man beachte, dass die Adresse der Variable \( x \) mittels des Befehls \texttt{LEA} an die Funktion \texttt{scanf()} übergeben wird, wohingegen der Wert der Variablen an das zweite printf() mit einem MOV Befehl übergeben wird. DWORD PTR—ist ein Teil der Assemblersprache (mit keinerlei Verbindung zum Maschinencode) und zeigt an, dass die Variablengröße 32 Bit beträgt und der MOV Befehl entsprechend aufgebaut sein muss.

**ARM: German text placeholder Keil 6/2013 (German text placeholder)**

Listing 1.56: IDA

```
.text:00000000 ; Segment type: Pure code
.text:00000000 AREA .text, CODE
...
.text:00000000 main
.text:00000000 PUSH {R4,LR}
.text:00000002 ADR R0, aEnterX ; "Enter X:\n"
.text:00000004 BL _2printf
.text:00000008 LDR R1, =x
.text:0000000A ADR R0, aD ; "%d"
.text:0000000C BL _0scanf
.text:00000010 LDR R0, =x
.text:00000012 LDR R1, [R0]
.text:00000014 ADR R0, aYouEnteredD___ ; "You entered %d...\n"
.text:00000016 BL _2printf
.text:00000018 MOVS R0, #0
.text:0000001C POP {R4,PC}
...
.text:00000020 aEnterX DCB "Enter X:\",0xA,0 ; DATA XREF: main+2
.text:0000002A DCB 0
.text:0000002B DCB 0
.text:0000002C off_2C DCD x ; DATA XREF: main+8
.text:0000002C main+10
.text:0000002C aD DCB "%d",0 ; DATA XREF: main+A
.text:00000033 DCD 0
.text:00000034 aYouEnteredD___ DCD "You entered %d...",0xA,0 ; DATA XREF: main+14
.text:00000047 DCD 0
.text:00000047 ; .text ends
.text:00000047 ...
.data:00000048 ; Segment type: Pure data
.data:00000048 AREA .data, DATA
.data:00000048 ORG 0x48
.data:00000048 EXPORT x
.data:00000048 x DCD 0xA ; DATA XREF: main+8
.data:00000048 ; main+10
65
Nun ist x eine globale Variable und aus diesem Grund in einem anderen Segment, nämlich dem Datensegment (.data) angelegt. Man könnte sich fragen, warum die Textstrings sich im Codesegment (.text) befinden und nicht x. Das liegt daran, dass x eine Variable ist und sich damit per definitionem ihr Wert ändern kann; mehr noch, er kann sich möglicherweise oft ändern. Textstrings dagegen sind Konstanten, können also nicht geändert werden, und befinden sich deshalb im .text Segment. Das Codesegment kann manchmal in einem ROM! Chip liegen (erinnern wir uns, dass wir es hier mit eingebetteter Mikroelektronik zu tun haben und Speicherknappheit ein häufiges Problem ist) und änderbare Variablen —im RAM! Es ist nicht besonders ökonomisch konstante Variablen im RAM zu speichern, wenn ein ROM verfügbar ist.

Mehr noch, Konstanten im RAM müssen initialisiert werden, denn nach dem Einschalten enthält der RAM naturgemäß zufällige Informationen.

Im Folgenden sehen wir einen Pointer auf die Variable x (off_2C) im Codesegment und bemerken, dass alle Operationen auf der Variable über den Pointer laufen.

Dies liegt daran, dass die Variable x auch an einem weit entfernten Codefragment liegen könnte, weshalb ihre Adresse irgendwo in der Nähe des Codes gespeichert werden muss.

Der Befehl LDR im Thumb Mode kann nur Variablen in einer Reichweite von 1020 Bytes von seiner Speicherstelle adressieren und im ARM Mode —beträgt die Spannweite der Variablen ±4095 Bytes.

Also muss die Adresse der Variable x sich in der Nähe befinden, denn es gibt keine Garantie, dass der Linker in der Lage ist, die Variable in der Nähe des Codes zu platzieren, sie könnte sich sogar auf einem externen Memory Chip befinden.

Eine weitere Sache: wenn eine Variable as konstant deklariert wird, legt der Keil Compiler sie im .const data Segment ab.

Möglicherweise könnte der Linker dieses Segment anschließend gemeinsam mit dem Codesegment im ROM ablegen.

ARM64

Listing 1.57: German text placeholder GCC 4.9.1 ARM64

```
.data:00000048 ; .data ends

1   .comm   x,4,4
2   .LC0:
3       .string   "Enter X:"
4   .LC1:
5       .string   "%d"
6   .LC2:
7       .string   "You entered %d...
8      
9   f5:  ; speichere FP und LR im Stack Frame:
10      stp   x29, x30, [sp, -16]!
11      ; setze Stack Frame (FP=SP)
12      add   x29, sp, 0
13      ; lade Pointer auf den "Enter X:" String:
14      adrp   x0, .LC0
15      add   x0, x0, :lo12:.LC0
16      bl   puts
17      ; lade Pointer auf den "%d" String:
18      adrp   x0, .LC1
19      add   x0, x0, :lo12:.LC1
20      ; bilde Adresse der globalen Variable x:
21      adrp   x1, x
22      add   x1, x1, :lo12:x
23      bl   __isoc99_scant
24      ; bilde Adress der globalen Variable x erneut:
25      adrp   x0, x
26      add   x0, x0, :lo12:x
27      ; lade Wert aus dem Speicher an dieser Adresse:
28      ldr   w1, [x0]
29      ; lade Pointer auf den "You entered %d...
30      ...
```

68ROM!
69Random-Access Memory
In diesem Fall ist die Variable \( x \) als global deklariert und ihre Adresse wird mithilfe des Befehlspaars ADRP/ADD berechnet (Zeilen 21 und 25).

**MIPS**

**Nicht initialisierte globale Variable**

Nun ist \( x \) eine globale Variable. Wir kompilieren zu einer ausführbaren Datei anstelle eines Objectfiles und laden diese in IDA. IDA stellt die Variable \( x \) in der ELF Section `.sbss` dar (erinnern Sie sich an „Global Pointer“? 1.5.4 on page 24), denn die Variable ist zu Beginn nicht initialisiert.

Listing 1.58: German text placeholder GCC 4.4.5 (IDA)
IDA reduziert die Anzahl der Informationen; also erzeugen wir auch ein Listing mit objdump und kommentieren es:

Listing 1.59: German text placeholder GCC 4.4.5 (objdump)

Wir erkennen, dass die Adresse der Variablen \( x \) mit GP aus einem 64KiB Datenpuffer gelesen wird und ein negativer Offset hinzugefügt wird (Zeile 18). Mehr noch, die Adressen der drei externen Funktionen, die in unserem Beispiel verwendet werden (puts(), scanf(), printf()) werden auch mithilfe von GP aus einem 64KiB globalen Datenpuffer gelesen (Zeile 9, 16 und 26). GP zeigt auf die Mitte des Puffers und ein solches Offset gibt an, dass die Adressen aller drei Funktionen genau wie die Adresse von \( x \) irgendwo am Anfang des Puffers gespeichert werden. Das ergibt Sinn, denn unser Beispiel ist winzig.

Eine andere bemerkenswerte Sache ist, dass die Funktion mit zwei NOPs (MOVE $AT,$AT — Befehle ohne Auswirkung) schließt, damit der Beginn der nächsten Funktion auf eine 16-Byte-Grenze gelegt wird.

**Initialisierte globale Variable**

Verändern wir unser Beispiel dadurch, dass wir \( x \) einen Defaultwert geben:

```c
int x=10; // Defaultwert
```

IDA zeigt hier, dass die Variable \( x \) im .data Segment liegt:

Listing 1.60: German text placeholder GCC 4.4.5 (IDA)
Warum nicht .sdata? Vielleicht hängt es mit einer Option von GCC zusammen?

Hier ist also das objdump Listing für eine genaue Untersuchung:

```
Listing 1.61: German text placeholder GCC 4.4.5 (objdump)

004006a0 <main>:  
004006a0:  3c1c0042  lui   $gp, 0x42  
004006a4:  27bdffe0  addiu  $sp, -0x20  
004006a8:  279c8930  addiu  $gp, $gp, -30416  
004006ac:  afbf001c  sw   $ra, 0x20($sp)  
004006b0:  afb00018  sw   $s0, 0x24($sp)  
004006b4:  afbc0010  sw   $gp, 0x16($sp)  
004006b8:  8f998034  lw    $t9, -32716($gp)  
004006bc:  3c040040  lui   $a0, 0x40  
004006c0:  0320f809  jalr  $t9  
004006c4:  248408d0  addiu  $a0, $a0, 2256  
004006c8:  8fbc0010  lw    $gp, 0x16($sp)  
   ; bereite höherwertigen Teil der Adresse von x vor:  
004006cc:  3c1c0041  lui   $sp, 0x41  
004006d0:  279c893a  la    $a0, __isoc99_scanf  
004006d4:  3c040060  lui   $a0, 0x20  
   ; addiere niederwertigen Teil der Adresse von x:  
004006dc:  jalr  $t9 ; __isoc99_scanf  
004006e0:  279c893a  la    $a0, aEnterX  
004006e4:  3c040060  lui   $a0, 0x20  
   ; höhe ein Word aus dem Speicher:  
004006ea:  lw    $gp, 0x20+var_10($sp)  
   ; Wert von x liegt nun in $a1.  
004006ec:  la    $t9, printf  
004006f0:  3c040060  lui   $a0, 0x40  
004006f4:  jalr  $t9 ; printf  
004006f8:  279c893a  la    $a0, aYouEnteredD  
004006fc:  3c040060  lui   $a0, 0x20  
00400700:  move  $v0, $zero  
00400704:  lw    $s0, 0x20+var_8($sp)  
00400708:  jr     $ra  
0040070c:  addiu  $sp, 0x20  
```
Wir erkennen, dass die Adresse mit LUI und ADDIU gebildet wird, aber der höherwertige Teil der Adresse sich immer noch im $S0 Register befindet und es möglich ist, den Offset durch einen LW ("Load Word") Befehl anzugeben, sodass ein einzelnes LW ausreicht um den Wert aus der Variablen zu lesen und an printf() zu übergeben.

Register, die temporäre Daten halten, beginnen mit T-, aber wir sehen hier auch einige, die mit dem Präfix S- beginnen. Die Inhalte dieser Register müssen gespeichert werden, bevor sie in anderen Funktionen verwendet werden können.

Das ist der Grund warum der Wert von $S0 an der Adresse 0x4006cc gesetzt und dann wieder an der Adresse 0x4006e8 verwendet wurde, nachdem scanf() aufgerufen wurde. Die Funktion scanf() verändert den Wert nicht.

### 1.8.4 scanf()

Wie bereits angemerkt ist es ein wenig altdiagnostisch heutzutage noch scanf() zu verwenden. Wenn wir aber darauf angewiesen sind, müssen wir prüfen, ob scanf() ohne Fehler ausgeführt wurde.

```c
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");
    if (scanf ("%d", &x)==1)
        printf ("You entered %d...\n", x);
    else
        printf ("What you entered? Huh?\n");
    return 0;
}
```

Gemäß dem Standard, gibt die Funktion scanf()\(^70\) die Anzahl der erfolgreich gelesenen Argumente zurück. In unserem Fall also 1, falls alles fehlerfrei funktioniert und der User eine Zahl eingibt oder im Fehlerfall (oder bei EOF\(^71\)) — 0.

Fügen wir ein wenig C Code hinzu, um den Rückgabewert von scanf() zu überprüfen und im Fehlerfall eine Fehlermeldung auszugeben.

Dies funktioniert wie erwartet:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...
C:\...>ex3.exe
```

\(^70\)scanf, wscanf: MSDN

\(^71\)End of File
Enter X:
ouch
What you entered? Huh?

**MSVC: x86**

Wir erhalten den folgenden Assembleroutput (MSVC 2010):

```
lea   eax, DWORD PTR _x$[ebp]
push  eax
push  OFFSET $SG3833 ; '%d', 00H
call  _scanf
add   esp, 8
cmp   eax, 1
jne   SHORT $LN2@main
mov   ecx, DWORD PTR _x$[ebp]
push  ecx
push  OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
call  _printf
add   esp, 8
jmp   SHORT $LN1@main
$LN2@main:
push  OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
call  _printf
add   esp, 4
$LN1@main:
xor   eax, eax
```

Die aufrufende Funktion (main()) benötigt das Ergebnis der aufgerufenen Funktion (scanf()), weshalb diese es über das Register EAX zurückgibt.

Wir prüfen mithilfe des Befehls CMP EAX, 1 (CoMPare). Mit anderen Worten, wir vergleichen den Wert in EAX mit 1.

Ein bedingter JNE Sprung folgt auf den CMP Befehl. JNE steht für Jump if Not Equal.

Wenn also der Wert in EAX ungleich 1 ist, wird die CPU die Ausführung an der Stelle fortsetzen, die imOperanden von JNE steht, in unserem Fall $LN2@main. Den Control Flow an diese Adresse zu übergeben hat zur Folge, dass die Funktion printf() mit dem Argument What you entered? Huh? aufgerufen wird. Wenn aber alles funktioniert und der bedingte Sprung nicht ausgeführt wird, wird ein anderer Aufruf von printf() mit zwei Argumenten ausgeführt: 'You entered %d... ' und dem Wert von x.

Da in diesem Fall das zweite printf() nicht ausgeführt werden darf, befindet sich davor ein JMP (unbedingter Sprung). Dieser gibt den Control Flow ab an den Punkt nach dem zweiten printf(), genau vor dem XOR EAX EAX Befehl, welcher die Rückgabe von 0 implementiert.

Man kann also festhalten, dass der Vergleich von zwei Werten gewöhnlich durch ein CMP/Jcc Befehlspaar implementiert wird, wobei cc für condition code, also Sprungbedingung, steht. CMP vergleicht zwei Werte und setzt die Flags des Prozessors. Jcc prüft diese Flags und entscheidet entweder den Control Flow an die angegebene Adresse zu übergeben oder nicht.

Es klingt möglicherweise paradox, aber der CMP Befehl ist tatsächlich ein SUB (subtract). Alle arithmetischen Befehle setzen die Flags des Prozessors, nicht nur CMP. Wenn wir 1 und 1 vergleichen, ist 1–1 = 0 und daher wird das ZF Flag gesetzt (gleichbedeutend damit, dass das Ergebnis der letzten Berechnung 0 ergeben hat). ZF kann nur durch diesen Umstand gesetzt werden, nämlich, dass zwei Operanden gleich sind. JNE prüft nur das ZF Flag und springt nur, wenn dieses nicht gesetzt ist. JNE ist daher ein Synonym für JNZ (Jump if Not Zero). Der Assembler übersetzt JNE und JNZ in den gleichen Opcode. Der CMP Befehl kann also durch ein SUB ersetzt werden, aber mit dem Unterschied, dass SUB den Wert des ersten Operanden verändert. CMP bedeutet also SUB ohne Speichern des Ergebnisses, aber mit Setzen der Flags.

**MSVC: x86: IDA**

Es ist an der Zeit IDA auszuprobieren und etwas damit zu machen. Für Anfänger ist es übrigens eine gute Idee, die /MD Option in MSVC zu verwenden, da diese bewirkt, dass alle Standardfunktionen nicht mit der ausführbaren Datei verlinkt werden, sondern aus der Datei MSVCR*.DLL importiert werden. Dadurch ist es einfacher zu erkennen, welche Standardfunktionen verwendet werden und wo dies geschieht.

---

72zu x86 Flags, siehe auch: [wikipedia](https://de.wikipedia.org/wiki/Flags).
Bei der Codeanalyse in IDA ist es hilfreich Notizen für sich selbst (und andere) zu hinterlassen. Bei der Analyse dieses Beispiels sehen wir, dass JNZ im Falle eines Fehlers ausgeführt wird. Es ist nun möglich den Cursor auf das Label zu setzen, „n“ zu drücken und es in „error“ umzubenennen. Wir erstellen noch ein Label—in „exit“. Hier ist das Ergebnis:

```plaintext
.text:00401000  _main proc near
.text:00401000 .text:00401000 var_4 = dword ptr -4
.text:00401000 argc = dword ptr 8
.text:00401000 argv = dword ptr 0Ch
.text:00401000 envp = dword ptr 10h
.text:00401000 .text:00401000 push ebp
.text:00401001 .text:00401001 mov ebp, esp
.text:00401003 .text:00401003 push ecx
.text:00401004 .text:00401004 push offset Format ; "Enter X:\n"
.text:00401009 .text:00401009 call ds:printf
.text:0040100F .text:0040100F add esp, 4
.text:00401012 .text:00401012 lea eax, [ebp+var_4]
.text:00401015 .text:00401015 push eax
.text:00401016 .text:00401016 push offset aD ; "%d"
.text:0040101B .text:0040101B call ds:scanf
.text:00401021 .text:00401021 add esp, 8
.text:00401024 .text:00401024 cmp eax, 1
.text:00401027 .text:00401027 jnz short error
.text:00401029 .text:00401029 ; print result
.text:00401032 .text:00401032 call ds:printf
.text:0040103B .text:0040103B add esp, 8
.text:0040103D .text:0040103D jmp short exit
.text:0040103D .text:0040103D error: ; CODE XREF: _main+27
.text:0040103D .text:0040103D push offset aWhat ; "What you entered? Huh?\n"
.text:00401042 .text:00401042 call ds:printf
.text:00401048 .text:00401048 add esp, 4
.text:0040104B .text:0040104B xor eax, eax
.text:0040104D .text:0040104D mov esp, ebp
.text:0040104F .text:0040104F pop ebp
.text:00401050 .text:00401050 retn
.text:00401050 _main endp
```

So ist es etwas einfacher den Code zu verstehen. Natürlich ist es aber auch keine gute Idee, jeden Befehl zu kommentieren.

Man kann Teile einer Funktion in IDA auch einklappen. Um dies zu tun, markiert man den Block und drückt dann „–“ auf dem Zahlenblock der Tastatur und gibt den stattdessen anzuzeigenden Text ein.

Verstecken wir zwei Blöcke und geben ihnen Namen:

```plaintext
.text:00401000  _text segment para public 'CODE' use32
.text:00401000  .text:00401000 assume cs: _text
.text:00401000  .text:00401000 ;org 401000h
.text:00401000  .text:00401000 ; ask for X
.text:00401012  .text:00401012 ; get X
.text:00401024  .text:00401024 cmp eax, 1
.text:00401027  .text:00401027 jnz short error
.text:00401029  .text:00401029 ; print result
.text:0040103B  .text:0040103B jmp short exit
.text:0040103D  .text:0040103D error: ; CODE XREF: _main+27
.text:0040103D  .text:0040103D push offset aWhat ; "What you entered? Huh?\n"
.text:00401042  .text:00401042 call ds:printf
.text:00401048  .text:00401048 add esp, 4
.text:0040104B  .text:0040104B xor eax, eax
.text:0040104D  .text:0040104D mov esp, ebp
.text:0040104F  .text:0040104F pop ebp
```
Um eingeklappte Teile des Code wieder auszuklappen, verwendet man „+“ auf dem Zahlenblock der Tastatur.
Durch Drücken von der „Leertaste“ sehen wir, wie IDA die Funktion als Graph darstellt:

Abbildung 1.18: Graph Modus in IDA

Es gibt hinter jedem bedingten Sprung zwei Pfeile: einen grünen und einen roten. Der grüne Pfeil zeigt auf den Codeblock der ausgeführt wird, wenn der Sprung ausgeführt wird und der rote den Codeblock, der ausgeführt wird, falls nicht gesprungen wird.
In diesem Modus ist es möglich, Knoten einzuklappen und ihnen auch Namen zu geben („group nodes“). Wir probieren das mit 3 Blöcken aus:

```plaintext
; int __cdecl main()
main proc near

var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
push    ecx
push    offset Format ; "Enter X:\n"
call    ds:printf
add     esp, 4
lea     eax, [ebp+var_4]
push    eax
push    offset aD ; "\%d"
call    ds:scanf
add     esp, 8
cmp     eax, 1
jnz     short error
```

Abbildung 1.19: Graph Modus in IDA mit 3 eingeklappten Knoten

Das ist sehr nützlich. Man kann sagen, dass ein großer Teil der Arbeit eines Reverse Engineers (und eines jeden anderen Forsches) darin besteht, die Menge der zur Verfügung stehenden Informationen zu reduzieren.
MSVC: x86 + OllyDbg

Laden wir unser Programm in OllyDbg und zwingen es dazu zu glauben, dass `scanf()` stets ohne Fehler arbeitet. Wenn die Adresse einer lokalen Variablen an `scanf()` übergeben wird, enthält die Variable zu Beginn einen zufälligen Wert, in diesem Fall 0x6E494714:

Abbildung 1.20: OllyDbg: Adresse der Variablen an `scanf()` übergeben
Während scanf() ausgeführt wird, geben wir in der Konsole etwas ein, das definitiv keine Zahl ist, z.B. „asdasd“. scanf() beendet sich mit 0 in EAX, was anzeigt, dass ein Fehler aufgetreten ist:

Abbildung 1.21: OllyDbg: scanf() Rückgabewert bei Fehler

Wir können auch die lokale Variable auf dem Stack überprüfen und stellen fest, dass sie sich nicht verändert hat. Was könnte scanf() hier auch hineinschreiben? Die Funktion hat nichts getan außer 0 zurückzugeben.

Versuchen wir unser Programm zu modifizieren, d.i. zu „hacken“. Rechtsklick auf EAX, in den Optionen finden wir „Set to 1“. Das ist was wir brauchen.

Wir haben jetzt 1 in EAX, sodass die folgende Überprüfung wie gewünscht ausgeführt wird und printf() den Wert der Variablen auf dem Stack ausgibt.

Wenn wir das Programm laufen lassen (F9), sehen wir das Folgende im Konsolenfenster:

Listing 1.62: console window

Enter X: asdasd
You entered 1850296084...

Und tatsächlich ist 1850296084 die dezimale Darstellung der Zahl auf dem Stack (0x6E494714)!
Unser Programm kann auch als einfaches Beispiel für das Patchen einer Executable dienen. Wir könnten versuchen, die Executable so zu patchen, dass das Programm unabhängig vom Input diesen stets auszugeben.

Angenommen, dass die Executable mit externer MSVCR*.DLL (d.h. mit der Option MD) kompiliert wurde, finden wir die Funktion main() am Anfang des .text Segments. Öffnen wir die Executable in Hiew und schauen uns den Anfang des .text Segments an (Enter, F8, F6, Enter, Enter).

Wir sehen das Folgende:

![Hiew: main() Funktion](image)

Hiew erkennt ASCII Strings und die Namen importierter Funktionen und zeigt diese an.

---

73dieser Vorgang wird auch „dynamisches Verlinken genannt“

74ASCII Zero ( )

---

78
Setzen wir den Cursor auf die Adresse .00401027, an der sich der JNZ Befehl, den wir umgehen müssen, befindet, drücken F3 und fügen dann „9090“ (zwei NOPs\(^7\)) ein.

### Abbildung 1.23: Hiew: ersetzen von JNZ durch zwei NOPs

Wir drücken F9 (update). Die Executable wird gespeichert und verhält sich wie gewünscht.

Zwei NOPs sind wahrscheinlich nicht der ästhetischste Ansatz. Ein anderer Weg die Executable zu patchen besteht darin, das zweite Byte des Opcodes (den German text placeholder) auf 0 zu setzen, sodass JNZ immer zum nächsten Befehl springt.

Wir könnten auch das Gegenteil tun: das erste Byte durch EB ersetzen und das zweite (German text placeholder) unangetastet lassen. Wir würde einen unbedingten Sprung erhalten, der stets ausgeführt wird. In diesem Fall würde unabhängig vom Input stets die Fehlermeldung ausgegeben.

### MIPS

Listing 1.63: German text placeholder GCC 4.4.5 (IDA)

```
.text:004006A0  main:
    .text:004006A0
    .text:004006A0  var_18  = -0x18
    .text:004006A0  var_10  = -0x10
7\(^{\text{a}}\) NOPs!
```
German text placeholder


1.8.5 German text placeholder

• http://challenges.re/53

1.9 Zugriff auf übergebene Argumente

Nun haben wir heraus gefunden das die caller Funktion die Argumente zur callee Funktion über den Stack schiebt. Aber wie greift die callee Funktion auf sie zu?

Listing 1.64: simple example

```c
#include <stdio.h>

int f (int a, int b, int c)
{
    return a*b+c;
};

int main()
{
```
printf ("%d\n", f(1, 2, 3));
return 0;
}

1.9.1 x86

MSVC

Das ist das Ergebnis nach dem kompilieren (MSVC 2010 Express):

Listing 1.65: MSVC 2010 Express

<table>
<thead>
<tr>
<th>TEXT SEGMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>_a$ = 8 ; größe = 4</td>
</tr>
<tr>
<td>_b$ = 12 ; größe = 4</td>
</tr>
<tr>
<td>_c$ = 16 ; größe = 4</td>
</tr>
</tbody>
</table>

_f PROC
push ebp
mov ebp, esp
mov eax, DWORD PTR _a$[ebp]
imul eax, DWORD PTR _b$[ebp]
add eax, DWORD PTR _c$[ebp]
pop ebp
ret 0
_f ENDP

_main PROC
push ebp
mov ebp, esp
push 3 ; drittes Argument
push 2 ; zweites Argument
push 1 ; erstes Argument
call _f
add esp, 12
push eax
push OFFSET $SG2463 ; 'd', 0Ah, 00H
call _printf
add esp, 8
; return 0
xor eax, eax
pop ebp
ret 0
_main ENDP

Was wir hier sehen ist das die main() Funktion drei Zahlen auf den Stack schiebt und f(int,int,int). aufruft

Der Argument zugriff innerhalb von f() wird organisiert mit der Hilfe von Makros wie zum Beispiel: _a$ = 8, auf die gleiche weise wie Lokale Variablen allerdings mit positiven Offsets (adressiert mit plus).

Also adressieren wir die äussere Seite des Stack frame indem wir _a$ Makros zum Wert des EBP Registers addieren

Dann wird der Wert von _a in EAX gespeichert. Nachdem die IMUL Instruktion ausgeführt wurde, ist der Wert in EAX ein Produkt des Wertes aus EAX und dem Inhalt von _b.

Nun addiert ADD den Wert in _c auf EAX

Der Wert in EAX muss nicht verschoben werden: Der Wert von EAX befindet sich schon wo er sein muss

Beim zurück kehren zur caller Funktion, wird der Wert aus EAX genommen und als Argument für den printf() Aufruf benutzt.

MSVC + OllyDbg

Lasst uns die Darstellung in OllyDbg betrachten

Wenn wir die erste Instruktion tracen in f() das auf eines der Argumente zugreift (das erste), können wir sehen das EBP auf den German text placeholder zeigt, dieser Frame wird mit dem roten Rechteck markiert dargestellt.
Das erste Element des gespeicherten Wert von EBP ist RA, das zweite Element ist das erste Funktions Argument, dann folgt das zweite und dritte Funktions Argument.

Um auf das erste Funktions Argument zu zugreifen, muss man lediglich exakt 8 (2 32-Bit Wörter) zu EBP addieren.

OllyDbg erkennt diesen Umstand, und Kommentare zu den entsprechenden Stack Elementen hinzugefügt zum Beispiel:

„RETURN from“ und „Arg1 = ...“, etc.

Beachte: Funktions Argumente sind keine Mitglieder des Funktions Stack Frame, sie sind eher Mitglieder des Stack Frame der caller Funktion.

Deswegen, hat OllyDbg die „Arg“ Elemente als Mitglied eines anderen Stackframes identifiziert.

Abbildung 1.24: OllyDbg: inside of f() function

**GCC**

Lasst uns das gleiche in GCC kompilieren und die Ergebnisse in IDA betrachten:

Listing 1.66: GCC 4.4.1

```assembly
public f
f proc near
arg_0 = dword ptr 8
arg_4 = dword ptr 0Ch
arg_8 = dword ptr 10h
push ebp
mov ebp, esp
mov eax, [ebp+arg_0]; erstes Argument
imul eax, [ebp+arg_4]; zweites Argument
add eax, [ebp+arg_8]; drittes Argument
pop ebp
retn
f endp

public main
main proc near
var_10 = dword ptr -10h
var_C = dword ptr -0Ch
var_8 = dword ptr -8
```

82
Das Ergebnis ist fast das gleiche aber mit kleineren Unterschieden die wir bereits früher besprochen haben.

Der **Stapel-Zeiger** wird nicht zurück gesetzt nach den beiden Funktion aufrufen (f und printf), weil die vorletzte LEAVE Instruktion (.1.6 on page 534) sich um das zurück setzen kümmert.

### 1.9.2 x64

Die Geschichte bei x86-64 Funktions Argumenten ist ein wenig anders (zumindest für die ersten vier bis sechs) sie werden über die Register übergeben z.b. der **callee** liest direkt aus den Registern anstatt vom Stack zu lesen.

#### MSVC

<table>
<thead>
<tr>
<th>German text placeholder MSVC:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Listing 1.67:</strong> German text placeholder MSVC 2012 x64</td>
</tr>
</tbody>
</table>

```
main PROC
    sub    rsp, 40
    mov    edx, 2
    lea    r8d, QWORD PTR [rdx+1] ; R8D=3
    lea    ecx, QWORD PTR [rdx-1] ; ECX=1
    call   f
    lea    rcx, OFFSET FLAT:$SG2997 ; '%'d'
    mov    edx, eax
    call   printf
    xor    eax, eax
    add    rsp, 40
    ret    0
main ENDP

f PROC
    ; ECX - 1st argument
    ; EDX - 2nd argument
    ; R8D - 3rd argument
    imul   ecx, edx
    lea    eax, DWORD PTR [r8+rcx]
    ret    0
f ENDP
```

Wie wir sehen können, die compact Funktion f() nimmt alle Argumente aus den Registern.

Die LEA Instruktion wird hier für Addition benutzt, scheinbar hat der Compiler die Instruktion für schneller befunden als die MOV Instruktion.

LEA wird auch benutzt in der main() Funktion um das erste und das dritte f() Argument vor zu bereiten. Der Compiler muss entschieden haben das dies schneller abgearbeitet wird als die Werte in die Register zu laden mit der MOV Instruktion.
Lasst uns einen Blick auf nicht optimierte MSVC Ausgabe werfen:

**Listing 1.68: MSVC 2012 x64**

```assembly
f proc near
 ; shadow space:
arg_0 = dword ptr 8
arg_8 = dword ptr 10h
arg_10 = dword ptr 18h

; ECX - 1st argument
; EDX - 2nd argument
; R8D - 3rd argument
mov [rsp+arg_10], r8d
mov [rsp+arg_8], edx
mov [rsp+arg_0], ecx
imul eax, [rsp+arg_0]
add eax, [rsp+arg_8]
retn
f endp

main proc near
sub rsp, 28h
mov r8d, 3 ; 3rd argument
mov edx, 2 ; 2nd argument
mov ecx, 1 ; 1st argument
call f
mov edx, eax
lea rcx, $SG2931 ; "%d\n"
call printf

; return 0
xor eax, eax
add rsp, 28h
retn
main endp
```

Es sieht ein bisschen wie ein Puzzle aus, weil alle drei Argumente aus den Registern auf dem Stack gespeichert werden aus irgend einem Grund. Dies bezeichnet man als „shadow space“

76: So wird sich wahrscheinlich jede Win64 EXE verhalten und alle vier Register Werte auf dem Stack speichern.

Das wird aus zwei Gründen so gemacht:

1) Es ist ziemlich übertrieben ein ganzes Register (oder gar vier Register) zu Reservieren für eine Argument Übergabe, also werden die Argumente über den Stack zugänglich gemacht. 2) Der Debugger weiß immer wo die Funktions Argumente zu finden sind bei einem breakpoint77.

Also, so können größere Funktionen ihre Eingabe Argumente im „shadow space“ speichern wenn die Funktion auf die Argumente während der Laufzeit zugreifen will, kleinere Funktionen (wie unsere) zeigen dieses Verhalten nicht.

Es liegt in der Verantwortung vom caller den „shadow space“ auf dem Stack zu allozieren.

**GCC**

Optimierter GCC generiert mehr oder minder verständlichen Code:

**Listing 1.69: German text placeholder GCC 4.4.6 x64**

```assembly
f:
 ; EDI - 1st argument
 ; ESI - 2nd argument
 ; EDX - 3rd argument
imul esi, edi
```

76MSDN
77MSDN
Listing 1.70: GCC 4.4.6 x64

```c
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
```

Bei System V *NIX Systemen ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, System V Application Binary Interface. AMD64 Architecture Processor Supplement, (2013)] 78) ist kein „shadow space“ nötig, aber der callee will vielleicht seine Argumente irgendwo speichern im Fall das keine oder zu wenig Register frei sind.

**GCC: uint64_t statt int**

Unser Beispiel funktioniert mit 32-Bit int, weshalb auch 32-Bit Register Bereiche benutzt werden (mit dem Präfix E-).

Es lassen sich auch ohne Probleme 64-Bit Werte benutzen:

```c
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
```

---

78German text placeholder https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf
```c
{ return a*b+c; }

int main()
{
    printf("%lld\n", f(0x1122334455667788,
                     0x1111111122222222,
                     0x3333333344444444));
    return 0;
}
```

Listing 1.71: German text placeholder GCC 4.4.6 x64

```asm
f proc near
    imul    rsi, rdi
    lea     rax, [rdx+rsi]
    retn
f endp

main proc near
    sub     rsp, 8
    mov     rdx, 3333333344444444h ; 3rd argument
    mov     rsi, 1111111122222222h ; 2nd argument
    mov     rdi, 1122334455667788h ; 1st argument
    call    f
    mov     edi, offset format ; "%lld\n"
    mov     rsi, rax
    xor     eax, eax ; number of vector registers passed
    call    __printf
    xor     eax, eax
    add     rsp, 8
    retn
main endp
```

Der Code ist der gleiche, aber diesmal werden die full size 64-Bit Register benutzt (mit dem R- Präfix).

### 1.9.3 ARM

German text placeholder Keil 6/2013 (German text placeholder)

```assembly
.text:000000A4 00 30 A0 E1 MOV R3, R0
.text:000000A8 93 21 20 E0 MLA R0, R3, R1, R2
.text:000000AC 1E FF 2F E1 BX LR
...
.text:000000B0 10 40 2D E9 STMD SP!, {R4,LR}
.text:000000B4 03 20 A0 E3 MOV R2, #3
.text:000000B8 02 10 A0 E3 MOV R1, #2
.text:000000BC 01 00 A0 E3 MOV R0, #1
.text:000000C0 F7 FF FF EB BL f
.text:000000C4 00 40 A0 E1 MOV R4, R0
.text:000000C8 04 10 A0 E1 MOV R1, R4
.text:000000CC 5A 0F 8F E2 ADR R0, aD 0 ; "%d\n"
.text:000000D0 E3 18 00 EB BL __2printf
.text:000000D4 00 00 A0 E3 MOV R0, #0
.text:000000D8 10 80 BD E8 LMDM SP!, {R4,PC}
```

Die main() Funktion ruft einfach zwei weitere Funktionen auf, mit diesen drei werten die dann der ersten Funktion übergeben werden.

Wie bereits angemerkt, auf ARM werden die erste 4 Werte in den ersten vier Registern übergeben (R0-R3).

Die f() Funktion, benutzt augenscheinlich die ersten drei Register (R0-R2) als Argumente.

Die MLA (Multiplikation Akkumulierung) Instruktion multipliziert den ersten der beiden Operanden (R3 und R1), addiert den dritten Operanden (R2) zum Produkt und speichert das Ergebnis in das nullte Register (R0), wohin per Standard definiert Funktionen ihre Rückgabe werte speichern.
Multiplikation und Addition in einem (*Fused multiply-add*) ist eine sehr nützliche Instruktion. Nebenbei bemerkt gab es eine solche Instruktion auf x86 nicht, bis FMA-Instruktionen in SIMD implementiert wurden.\(^{79}\)

Die erste Instruktion `MOV R3, R0` ist anscheinen redundant (es hätte anstatt eine einzelne MLA Instruktion benutzt werden können). Der Compiler hat die Instruktion nicht weg optimiert, da das Programm ohne Optimierungen kompiliert wurde.


---

**German text placeholder Keil 6/2013 (German text placeholder)**

<table>
<thead>
<tr>
<th>.text:00000098</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:00000098</td>
<td>91 20 20 E0</td>
</tr>
<tr>
<td>.text:0000009C</td>
<td>1E FF 2F E1</td>
</tr>
</tbody>
</table>

Und hier wurde die `f()` Funktion mit dem Kreil Compiler und allen Optimierungen (/03) kompiliert.

Die `MOV` Instruktion wurde weg optimiert und jetzt benutzt MLA alle Eingabe Register und schreibt die Ergebnisse direkt nach R0. Genau da wo die aufrufende Funktion das Ergebnis auslesen und benutzen wird.

**German text placeholder Keil 6/2013 (German text placeholder)**

<table>
<thead>
<tr>
<th>.text:0000005E</th>
<th>48 43</th>
<th>MULS R0, R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:00000060</td>
<td>80 18</td>
<td>ADDS R0, R0, R2</td>
</tr>
<tr>
<td>.text:00000062</td>
<td>70 47</td>
<td>BX LR</td>
</tr>
</tbody>
</table>

Die MLA Instruktion ist im Thumb Modus nicht verfügbar, also muss der Compiler den Code für diese beiden Instruktionen (Multiplikation und Addition) separat generieren.

Zu erst multipliziert die `MULS` Instruktion R0 mit R1 und platziert das Ergebnis im Register R0. Die zweite Instruktion (`ADD`) addiert das Ergebnis mit R2 und platziert das Ergebnis wieder im R0 Register.

**ARM64**

**German text placeholder GCC (Linaro) 4.9**

Hier ist alles ganz einfach. MADD ist einfach eine Instruktion die Multiplikation/Addition verschmelzt (ähnlich wie wir es bei MLA gesehen haben)

Alle drei Argumente werden über den 32-Bit Part des X-Registers übergeben. In der tat, die Argumente sind 32-Bit int’s. Das Ergebnis wird in W0 gespeichert.

---

**Listing 1.72: German text placeholder GCC (Linaro) 4.9**

```plaintext
f:
    madd w0, w0, w1, w2
    ret

main:
    ; save FP and LR to stack frame:
    stp x29, x30, [sp, -16]!
    mov w2, 3
    mov w1, 2
    add x29, sp, 0
    mov w0, 1
    bl f
    mov w1, w0
```
Lasst uns nun alle Datentypen nach 64-Bit `uint64_t` konvertieren und testen:

```c
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
    return a*b+c;
};

int main()
{
    printf ("%lld\n", f(0x1122334455667788,
                      0x1111111122222222,
                      0x3333333344444444));
    return 0;
};
```

Die `f()` Funktion ist die gleiche, nur das jetzt alle 64-Bit X-Register benutzt werden. Lange 64-Bit Werte werden Stückweise in die Register geladen, genauer beschrieben hier: 1.27.3 on page 402.

**German text placeholder GCC (Linaro) 4.9**

Der nicht optimierte Compiler lauf ist redundant:

```c
f:
    madd  x0, x0, x1, x2
    ret
main:
    mov   x1, 13396
    adrp  x0, .LC8
    stp   x29, x30, [sp, -16]!
    movk  x1, 0x27d0, lsl 16
    add   x0, x0, :lo12:.LC8
    movk  x1, 0x122, lsl 32
    add   x29, sp, 0
    movk  x1, 0x58be, lsl 48
    bl    printf
    mov   w0, 0
    ldp   x29, x30, [sp], 16
    ret
.LC8:
    .string "%lld\n"
```
Der Code speichert seine Eingabe Argumente auf dem lokalen Stack, für den Fall das die Funktion die W0..W2 Register benutzen muss das verhindert das überschreiben der Original Argumente, die vielleicht noch in Zukunft gebraucht werden.

Das bezeichnet man auch als Register Save Area. ([Procedure Call Standard for the ARM 64-bit Architecture (AArch64), (2013)](http://go.yurichev.com/17287)). Der Callee, ist hier nicht in der Pflicht die Werte zu speichern. So Ähnlich wie beim „Shadow Space“: 1.9.2 on page 84.

Warum hat der optimierte GCC 4.9 Aufruf dieses Argument weg gelassen? Weil der Compiler in dem Fall zusätzliche Optimierungen gemacht hat. Und erkannt hat das die zusätzlichen Argumente in der weiteren Ausführung des Codes nicht mehr benötigt werden. Und auch das die Register W0..W2 auch nicht weiter benötigt werden.

Wir können auch ein MUL/ADD Instruktions paar sehen anstatt einem einzelnen MADD.

1.9.4 MIPS

Listing 1.73: German text placeholder GCC 4.4.5

Die ersten vier Funktions Argumente werden in vier Register übergeben die das A- Präfix haben.

Es gibt zwei spezial Register in MIPS: HI und LO die das 64-Bit Multiplikation Ergebnis der Ausführung der MULT Instruktion enthalten.

80German text placeholder http://go.yurichev.com/17287
Auf diese Register sind nur zugreifbar durch die MFL0 und die MFHI Instruktionen. MFL0 enthält hier die niedrigen Bits aus dem Multiplikations Ergebnis und speichert diese in $V0. Also wird der höhere Wert des 32-Bit Teils der multiplikation einfach verworfen ( der HI Register in halt wird nicht verwendet ) . In der Tat: Wir operieren hier auf 32-Bit int Daten Typen.

Zum Schluss addiert ADDU („Add Unsigned“) den Wert des dritten Argumentes zum Ergebnis.

Es gibt zwei unterschiedliche Addition Instruktionen auf der MIPS Plattform: ADD und ADDU. Der unterschied zwischen den beiden Instruktionen bezieht sich nicht auf das Vorzeichen (+/-) sondern auf die exceptions. ADD kann eine exception werfen bei einem overflow, was manchmal nützlich sein kann und wird auch bei Ada PS unterstützt, zum Beispiel:

ADDU wirft keine exception bei einem overflow.

Da C/C++ keine Unterstützung hierfür bietet, sehen wir in unserem Beispiel ADDU statt ADD.

Das 32-Bit Ergebnis bleibt übrig in $V0.

In main() existiert nun eine neue Instruktion, die interessant für uns ist: JAL „Jump an Link“.

Der unterschied zwischen JAL und JALR ist das in der ersten Instruktion ein relatives offset hart codiert ist, während JALR zur absoluten Adresse gespeichert in einem Register springt („Jump und Link Register“).

Beide f() und die main() Funktionen liegen innerhalb der gleichen Objekt Datei, also ist die relative Adresse von f() bekannt und fix.

1.10 Pointer
1.10.1 Werte zurückgeben

Pointer werden oft verwendet um Funktionsergebnisse zurückzuliefern (siehe der Fall scanf() (1.8 on page 50)).

Zum Beispiel dann, wenn eine Funktion zwei Werte zurückgeben soll.

Beispiel mit globalen Variablen

```c
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
    *sum=x+y;
    *product=x*y;
};

int sum, product;

void main()
{
    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d\n", sum, product);
};
```

Dies kompiliert zu:

Listing 1.74: German text placeholder MSVC 2010 (/Ob0)

```
COMM _product:DWORD
COMM sum:DWORD
$SG2803 DB 'sum=%d, product=%d', 0aH, 00H

_x$ = 8 ; size = 4
_y$ = 12 ; size = 4
_sum$ = 16 ; size = 4
_product$ = 20 ; size = 4

f1 PROC
    mov ecx, DWORD PTR _y$[esp-4]
    mov eax, DWORD PTR _x$[esp-4]

81http://go.yurichev.com/17326
82Programmiersprache
```
lea edx, DWORD PTR [eax+ecx]
imul eax, ecx
mov ecx, DWORD PTR _product$[esp-4]
push esi
mov esi, DWORD PTR _sum$[esp]
mov DWORD PTR [esi], edx
mov DWORD PTR [ecx], eax
pop esi
ret 0
_f1 ENDP

_main PROC
push OFFSET _product
push OFFSET _sum
push 456 ; 000001c8H
push 123 ; 0000007bH
call _f1
mov eax, DWORD PTR _product
mov ecx, DWORD PTR _sum
push eax
push ecx
push OFFSET $SG2803
call DWORD PTR __imp__printf
add esp, 28
xor eax, eax
ret 0
_main ENDP
Schauen wir es uns in OllyDbg an:

Abbildung 1.25: OllyDbg: Adressen der globalen Variablen werden an f1() übergeben

Zuerst werden die Adressen der globalen Variablen an f1() übergeben. Wir klicken auf „Follow in dump“ beim Stackelement und wir sehen den Platz, der im Datensegment für die beiden Variablen angelegt wird.
Diese Variablen werden auf Null gesetzt, denn nicht initialisierte Daten (aus BSS) werden gelöscht, bevor die Ausführung beginnt, [siehe ISO/IEC 9899:TC3 (C C99 standard), (2007) 6.7.8p10].

Sie bleiben im Datensegment, was wir durch Drücken von Alt-M und untersuchen der Speicherzuordnung verifizieren können:

Abbildung 1.26: OllyDbg: Speicherzuordnung
Verfolgen wir den Ablauf (F7) bis zum Start von f1():

Abbildung 1.27: OllyDbg: f1() beginnt

Zwei Werte sind auf dem Stack sichtbar: 456 (0x1C8) und 123 (0x7B) und außerdem die Adressen der beiden globalen Variablen.
Verfolgen wir den Ablauf bis zum Ende von f1(). Im linken unteren Fenster sehen wir wie die Ergebnisse der Berechnung in den globalen Variablen erscheinen:

Abbildung 1.28: OllyDbg: Ausführung von f1() beendet
Jetzt werden die Werte der globalen Variablen in Register geladen, um dann an `printf()` übergeben zu werden (über den Stack):

Abbildung 1.29: OllyDbg: Adressen der globalen Variablen werden an `printf()` übergeben

Beispiel mit lokalen Variablen

Verändern wir unser Beispiel ein wenig:

```c
void main()
{
    int sum, product; // die beiden sind nun lokale Variablen
    f1(123, 456, &sum, &product);
    printf(\"sum=\%d, product=\%d\n\", sum, product);
}
```

Der Code von `f1()` wird sich nicht verändern. Nur den Code von `main()` wird sich verändern:

```c
_main PROC
; Line 10
sub     esp, 8
; Line 13
lea     eax, DWORD PTR _product$[esp+8]
push    eax
lea     ecx, DWORD PTR _sum$[esp+12]
push    ecx
push    456 ; 000001c8
push    123 ; 0000007b
call    _f1
; Line 14
mov     edx, DWORD PTR _product$[esp+24]
mov     eax, DWORD PTR _sum$[esp+24]
push    edx
push    eax
push    OFFSET $SG2803
call    DWORD PTR __imp__printf
; Line 15
xor     eax, eax
add     esp, 36
```

Listing 1.76: German text placeholder MSVC 2010 (/Ob0)
Schauen wir es uns erneut mit OllyDbg an. Die Adressen der lokalen Variablen auf dem Stack sind 0x2EF854 und 0x2EF858. Wir erkennen wie diese auf dem Stack abgelegt werden:

Abbildung 1.30: OllyDbg: Adressen der lokalen Variablen werden auf dem Stack abgelegt
f1() beginnt. Bis hierher befinden sich nur zufällige Werte auf dem Stack an den Adressen 0x2EF854 und 0x2EF858.

Abbildung 1.31: OllyDbg: f1() beginnt
Wir finden nun 0xDB18 und 0x243 an den Adressen 0x2EF854 und 0x2EF858. Diese Werte sind die Ergebnisse von f1().

**Fazit**

f1() kann Pointer auf jede beliebige Speicherstelle zurückgeben. Dies ist die Quintessenz der Nützlichkeit von Pointern.

C++ references funktionieren übrigens genau auf die gleiche Weise. Lesen Sie mehr dazu: (?? on page ??).

### 1.10.2 Eingabewerte vertauschen

Dies erledigt die Aufgabe für uns:

```c
#include <memory.h>
#include <stdio.h>

void swap_bytes (unsigned char* first, unsigned char* second)
{
    unsigned char tmp1;
    unsigned char tmp2;
    
    tmp1=*first;
    tmp2=*second;
    
    *first=tmp2;
    *second=tmp1;
}

int main()
{
    // copy string into heap, so we will be able to modify it
    char *s=strdup("string");
    
    // swap 2nd and 3rd characters
    swap_bytes (s+1, s+2);
    
    printf ("%s\n", s);
}
```
Wie wir erkennen, werden mit MOVZX Bytes in die niederen 8 Bit von ECX und EBX geladen (sodass die höherwertigen Teile dieser Register gelöscht werden) und danach werden die Bytes in umgekehrter Reihenfolge zurückgeschrieben.

Listing 1.77: Optimizing GCC 5.4

```assembly
swap_bytes:
    push  ebx
    mov  edx, DWORD PTR [esp+8]
    mov  eax, DWORD PTR [esp+12]
    movzx ecx, BYTE PTR [edx]
    movzx ebx, BYTE PTR [eax]
    mov  BYTE PTR [edx], bl
    mov  BYTE PTR [eax], cl
    pop  ebx
    ret
```

Die Adressen der beiden Bytes, die von Argumenten und Ausführung von der Funktion stammen, befinden sich in EDX und EAX.

Wenn wir Pointer verwenden: möglicherweise gibt es keinen besseren Wert diese Aufgabe ohne zu lösen.

### 1.11 GOTO Operator


Hier ist ein sehr einfaches Beispiel:

```c
#include <stdio.h>

int main()
{
    printf("begin\n");
    goto exit;
    printf("skip me!\n");
exit:
    printf("end\n");
}
```

Dies erhalten wir in MSVC 2012:

Listing 1.78: MSVC 2012

```assembly
_main PROC
    push ebp
    mov  ebp, esp
    push OFFSET $SG2934 ; 'begin'
    call _printf
    add  esp, 4
    jmp  SHORT $exit$3
    push OFFSET $SG2936 ; 'skip me'
    call _printf
    add  esp, 4
$exit$3:
    push OFFSET $SG2937 ; 'end'
    call _printf
    add  esp, 4
    xor  eax, eax
    pop  ebp
    ret 0
_main ENDP
```

[Dennis Yurichev, C/C++ programming language notes](http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf) enthält auch einige Beispiele.
Der Goto Ausdruck wurde einfach durch einen JMP Befehl ersetzt, der den gleichen Effekt hat: einen unbedingten Sprung an eine andere Stelle. Das zweite printf() kann nur durch menschlichen Eingriff ausgeführt werden, z.B. durch einen Debugger oder Patchen des Codes.
Dies könnte auch als einfache Übung zum Patchen von Nutzen sein. Öffnen wir die Executable in Hiew:

Abbildung 1.33: Hiew
Wir setzen den Cursor auf die Adresse JMP (0x410), drücken F3 (edit), drücken zweimal Null, sodass der Opcode zu EB 00 verändert wird:

![Image showing the editor with the updated opcode]

Abbildung 1.34: Hiew

Das zweite Byte des JMP Opcodes gibt den relativen Offset für den Sprung an; 0 entspricht dabei der Stelle direkt hinter dem aktuellen Befehl.

Auf diese Weise wird JMP den zweiten Aufruf von printf() nicht überspringen.

Wir drücken F9 (save) und verlassen den Editor. Wenn wir die Executable jetzt ausführen, sehen wir dies:

Listing 1.79: Patched executable output

```
C:\...>goto.exe
begin
skip me!
end
```

Das gleiche Ergebnis kann erreicht werden, wenn der JMP Befehl durch 2 NOP Befehle ersetzt wird.

NOP hat einen Opcode von 0x90 und die Länge 1 Byte, sodass wir 2 Befehle als Ersatz für JMP, welcher eine Größe von 2 Byte hat, benötigen.

### 1.11.1 Dead code

Der zweite Aufruf von printf() wird fachsprachlich auch „dead code“ genannt. Dies bedeutet, dass der Code nie ausgeführt wird. Wenn wir dieses Beispiel mit aktivierter Optimierung kompilieren, entfernt der Compiler den „dead code“ und es gibt keine Spur mehr von ihm:

Listing 1.80: German text placeholder MSVC 2012

```
$SG2981 DB 'begin', 0AH, 00H
$SG2983 DB 'skip me!', 0AH, 00H
$SG2984 DB 'end', 0AH, 00H

_main PROC
    push OFFSET $SG2981 ; 'begin'
    call printf
    push OFFSET $SG2984 ; 'end'
$exit$4:
    call printf
    add esp, 8
    xor eax, eax
```
Trotzdem hat der Compiler vergessen, den „skip me“ String ebenfalls zu entfernen.

### 1.11.2 German text placeholder

Versuchen Sie das gleiche Ergebnis mit ihrem bevorzugten Compiler und Debugger zu erzielen.

### 1.12 Bedingte Sprünge

#### 1.12.1 einfaches Beispiel

```c
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
}

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
}

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
}
```

**x86**

**x86 + MSVC**

Die Funktions f_signed() sieht folgendermaßen aus:

Listing 1.81: German text placeholder MSVC 2010

```
.a$ = 8
_b$ = 12
_f_signed PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    cmp eax, DWORD PTR _b$[ebp]
    jle SHORT $LN3@f_signed
    push OFFSET $SG737 ; 'a>b'
    call _printf
    add esp, 4
$LN3@f_signed:
    mov ecx, DWORD PTR _a$[ebp]
    cmp ecx, DWORD PTR _b$[ebp]
    jne SHORT $LN2@f_signed
    push OFFSET $SG739 ; 'a==b'
```

105
Der erste Befehl, JLE steht für *Jump if Less or Equal*. Mit anderen Worten, wenn der zweite Operand größer gleich dem ersten ist, wird der Control Flow an die angegebene Adresse bzw. das angegebene Label übergeben. Wenn die Bedingung falsch ist, weil der zweite Operand kleiner ist als der erste, wird der Control Flow nicht verändert und das erste printf() wird ausgeführt.

Der zweite Check ist JNE: *Jump if Not Equal*. Der Control Flow wird nicht verändert, wenn die Operanden gleich sind.

Der dritte Check ist JGE: *Jump if Greater or Equal*—springe, falls der erste Operand größer gleich dem zweiten ist. Wenn also alle drei bedingten Sprünge ausgeführt werden, wird also kein Aufruf von printf() ausgeführt. Dies ist ohne manuellen Eingriff unmöglich. Werfen wir nun einen Blick auf die Funktion f_unsigned(). Diese Funktion macht prinzipiell das gleiche wie f_signed() mit der Ausnahme, dass die Befehle JBE und JAE anstelle von JLE und JGE wie folgt verwendet werden:

Wie bereits erwähnt unterscheiden sich die Verzweigungsbefehle: JBE—*Jump if Below or Equal* und JAE—*Jump if Above or Equal*. Diese Befehle (JA/JAE/JB/JBE) unterscheiden sich von JG/JGE/ JL/JLE dadurch, dass sie mit vorzeichenlosen Zahlen arbeiten.

Siehe hierzu auch den Abschnitt über Darstellung vorzeichenbehafteter Zahlen (2.1 on page 408). Das ist der Grund warum wir, wenn wir JG/JL anstelle von JA/JB und umgekehrt finden, fast mit Gewissheit sagen können, dass die Variablen vorzeichenbehaftet bzw. vorzeichenlos sind. Hier befindet sich auch die Funktion main(), welche für uns nichts Neues bereithält:
_main  PROC
    push   ebp
    mov    ebp, esp
    push   2
    push   1
    call   _f_signed
    add    esp, 8
    push   2
    push   1
    call   _f_unsigned
    add    esp, 8
    xor    eax, eax
    pop    ebp
    ret    0
_main  ENDP
We can see how flags are set by running this example in OllyDbg. Let’s begin with `f_unsigned()`, which works with unsigned numbers.

CMP is executed thrice here, but for the same arguments, so the flags are the same each time.

**Result of the first comparison:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>disp</th>
<th>ASCII (ANSI)</th>
<th>OPR</th>
<th>STR</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110A65</td>
<td>11</td>
<td>20</td>
<td>112.63</td>
<td>01</td>
<td>11</td>
<td>00</td>
</tr>
</tbody>
</table>

So, the flags are: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0.

They are named with one character for brevity in OllyDbg.

OllyDbg gives a hint that the (JBE) jump is to be triggered now. Indeed, if we take a look into Intel manuals (11.1.4 on page 520), we can read there that JBE is triggering if CF=1 or ZF=1. The condition is true here, so the jump is triggered.

**Abbildung 1.35: OllyDbg: f_unsigned(): first conditional jump**

![CPU - main thread, module ex](image)

The jump is taken.

Source: Ex. 00110A65

Last Exit 00000000 ERROR_SUCCESS

PEL 0000297
The next conditional jump:

OllyDbg gives a hint that JNZ is to be triggered now. Indeed, JNZ triggering if ZF=0 (zero flag).

Abbildung 1.36: OllyDbg: f_unsigned(): second conditional jump

OllyDbg gives a hint that JNZ is to be triggered now. Indeed, JNZ triggering if ZF=0 (zero flag).
The third conditional jump, JNB:

Abbildung 1.37: OllyDbg: f_unsigned(): third conditional jump

In Intel manuals (11.1.4 on page 520) we can see that JNB triggers if CF=0 (carry flag). That is not true in our case, so the third printf() will execute.
Now let's review the `f_signed()` function, which works with signed values, in OllyDbg. Flags are set in the same way: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. The first conditional jump JLE is to be triggered:

In Intel manuals (11.1.4 on page 520) we find that this instruction is triggered if ZF=1 or SF≠OF. SF≠OF in our case, so the jump triggers.
The second JNZ conditional jump triggering: if ZF=0 (zero flag):

Abbildung 1.39: OllyDbg: f_signed(): second conditional jump
The third conditional jump \texttt{JGE} will not trigger because it would only do so if SF=OF, and that is not true in our case:

\textbf{Abbildung 1.40:} OllyDbg: \texttt{f\_signed()}: third conditional jump
Wir können versuchen, die Executable so zu verändern, dass die Funktion `f_unsigned()` stets „a==b“ ausgibt, egal was wir eingeben. So sieht das ganze in Hiew aus:

Abbildung 1.41: Hiew: Funktion `f_unsigned()`

Grundsätzlich haben wir drei Dinge zu erzwingen:

- den ersten Sprung stets ausführen;
- den zweiten Sprung nie ausführen;
- den dritten Sprung stets ausführen.

Dadurch können wir den Code Flow so manipulieren, dass das zweite `printf()` immer ausgeführt wird und „a==b“ ausgibt. Drei Befehle (oder Bytes) müssen verändert werden:

- Der erste Sprung wird zu JMP verändert, aber der German text placeholder bleibt gleich.
- Der zweite Sprung könnte manchmal ausgeführt werden, wird aber in jedem Fall zum nächsten Befehl springen, denn wir setzen den German text placeholder auf 0. Bei diesen Befehlen wird der German text placeholder zu der Adresse der nächsten Befehls addiert. Wenn der Offset 0 ist, wird die Ausführung also beim nächsten Befehl fortgesetzt.
- Den dritten Sprung ersetzen wie genau wie den ersten durch JMP, damit er stets ausgeführt wird.
Hier ist der veränderte Code:

Abbildung 1.42: Hiew: Veränderte Funktion f_unsigned()

Wenn wir es verpassen, einen dieser Sprünge zu verändern, könnten mehrere Aufrufe von printf() ausgeführt werden; wir wollen aber nur genau einen Aufruf ausführen.

**German text placeholder GCC**

**German text placeholder** GCC 4.4.1 erzeugt fast identischen Code, aber mit `puts()` (1.5.3 on page 21) anstelle von printf().

**German text placeholder GCC**

Der aufmerksame Leser fragt sich vielleicht, warum CMP mehrmals ausgeführt wird, wenn doch die Flags nach jeder Ausführung dieselben Werte haben.

Vielleicht kann der optimierende MSVC dies nicht leisten, aber der optimierende GCC 4.8.1 gräbt tiefer:

Listing 1.84: GCC 4.8.1 f_signed()

```
f_signed:
mov eax, DWORD PTR [esp+8]
cmp DWORD PTR [esp+4], eax
jg .L6
je .L7
jge .L1
mov DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
jmp puts
.L6:
mov DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
jmp puts
.L1:
rep ret
```
Wir finden auch hier JMP puts anstelle von CALL puts / RETN.
Dieser Trick wird später erklärt: 1.13.1 on page 134.
Diese Sorte x86 Code ist trotzdem selten. MSVC 2012 kann wie es scheint solchen Code nicht erzeugen. Andererseits sind Assemblerprogrammierer sich natürlich der Tatsache bewusst, dass Jcc Befehle gestackt werden können. Wenn man solche gestackten Befehle findet, ist es sehr wahrscheinlich, dass der entsprechende Code von Hand geschrieben wurde. Die Funktion f_unsigned() ist nicht so ästhetisch:

```
Listing 1.85: GCC 4.8.1 f_unsigned()
f_unsigned:
push esi
push ebx
sub esp, 20
mov esi, DWORD PTR [esp+32]
mov ebx, DWORD PTR [esp+36]
cmp esi, ebx
ja .L13
cmp esi, ebx ; dieser Befehl könnte entfernt werden
je .L14
.
.L10:
jb .L15
add esp, 20
pop ebx
pop esi
ret
.
.L15:
mov DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
add esp, 20
pop ebx
pop esi
jmp puts
.
.L13:
mov DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
call puts
cmp esi, ebx
jne .L10
.
.L14:
mov DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
add esp, 20
pop ebx
pop esi
jmp puts
```

Trotzdem werden hier immerhin nur zwei statt drei CMP Befehle verwendet. Die Optimierungsalgorithmen von GCC 4.8.1 sind möglicherweise noch nicht so ausgereift.

ARM

32-bit ARM

German text placeholder Keil 6/2013 (German text placeholder)
Viele Befehle im ARM mode können nur ausgeführt werden, wenn spezielle Flags gesetzt sind. Dies ist beispielsweise oft beim Vergleich von Zahlen der Fall.

Der ADD Befehl zum Beispiel heißt hier intern ADDAL, wobei AL für Always (dt. immer) steht, d.h. er wird immer ausgeführt. Die Prädikate werden in den 4 höchstwertigsten Bits des 32-Bit-ARM-Befehls kodiert, dem condition field.

Der Befehl ADRB für einen unbedingten Sprung ist tatsächlich doch bedingt und genau wie jeder andere bedingte Sprung kodiert, nutzt dass er AL im condition field hat und dadurch die Flags ignoriert und immer ausgeführt wird.

Der Befehl ADRG BT arbeitet wie ADR, wird aber nur ausgeführt, wenn das vorangehende CMP ergeben hat, dass eine der beiden Eingabezahlen größer war als die andere.

Der folgende BLGT Befehl verhält sich genau wie BL und wird nur dann ausgeführt, wenn das Ergebnis des Vergleichs das gleiche war (d.h. größer als). ADRGT schreibt einen Pointer auf den String a>b\n nach R0 und BLGT ruft printf() auf. Das heißt, Befehl mit dem Suffix -GT werden nur ausgeführt, wenn der Wert in R0 (das ist a) größer ist als der Wert in R4 (das ist b).

Im Folgenden finden wir die Befehle ADREQ und BLEQ. Sie verhalten sich wie ADR und BL, werden aber nur ausgeführt, wenn die beiden Operanden des letzten Vergleichs gleich waren. Ein weiteres CMP befindet sich davor (denn die Ausführung von printf() könnte die Flags verändert haben).

Dann finden wir LDMGEFD; dieser Befehl arbeitet genau wie LDMFD\(^86\), wird aber nur ausgeführt, wenn einer der Werte größer gleich dem anderen ist. Der Befehl LDMGEFD SP!, {R4-R6, PC} fungiert als Funktionsepilog, wird aber nur ausgeführt, ehnw a >= b und nur dann wird die Funktionsausführung beendet. Wenn aber diese Bedingung nicht erfüllt ist, d.h. a < b, wird der Control Flow zum nächsten „LDMFD SP!, {R4-R6, LR}“ springen, der ebenfalls einen Funktionsepilog darstellt. Dieser Befehl stellt nicht nur den Zustand der R4-R6 Register wieder her, sondern auch LR anstatt PC, dadurch gibt es nichts aus der Funktion zurück. Die letzten beiden Befehle rufen printf() mit dem String «a<b\n» als einzigen Argument auf. Wir haben bereits einen unbedingten Sprung zur Funktion printf() anstelle einer Funktionsrückgabe im Abschnitt «German text placeholder» (?? on page ??) untersucht.

f_unsigned ist ähnlich, nur die Befehle ADRHI, BLHL und LDMCSFD werden hier verwendet. Deren Prädikaten (HI = Unsigned higher, CS = Carry Set (greater than or equal)) sind analog zu den eben betrachteten, nur eben für vorzeichenlose Werte.

In der Funktion main() finden wir nicht viel Neues:

Listing 1.87: main()

| .text:000000128 | EXPORT main |
| .text:000000128 | main |
| .text:000000128 10 40 2D E9 | STMFD SP!, {R4,LR} |
| .text:000000012C 02 10 A0 E3 | MOV R1, #2 |
| .text:000000130 01 00 A0 E3 | MOV R0, #1 |
| .text:000000134 DF FF FF EB | BL f_signed |
| .text:000000138 02 10 A0 E3 | MOV R1, #2 |
| .text:00000013C 01 00 A0 E3 | MOV R0, #1 |
| .text:000000140 EA FF FF EB | BL f_unsigned |
| .text:000000144 00 00 A0 E3 | MOV R0, #0 |
| .text:000000148 10 80 BD E8 | LDMFD SP!, {R4,PC} |
| .text:000000148 ; End of function main |

Auf diese Weise kann man bedingte Sprünge im ARM mode entfernen.

Für eine Begründung warum dies vorteilhaft ist, siehe: ?? on page ??.

---

\(^86\) LDMFD
In x86 gibt es kein solches Feature, außer dem CMOVcc Befehl, der genau wie MOV funktioniert, aber nur ausgeführt wird, wenn spezielle Flags - normalerweise durch CMP gesetzt sind.

German text placeholder Keil 6/2013 (German text placeholder)

Nur der B Befehl im Thumb mode kann mit condition codes versehen werden, sodass der Thumb Code gewöhnlicher aussieht.
BLE ist ein normaler bedingter Sprung Less than or Equal, BNE—Not Equal, BGE—Greater than or Equal. f_unsigned ist ähnlich, nur dass andere Befehle verwendet werden, wenn mit vorzeichenlosen Werten umgegangen wird: BLS (Unsigned lower or same) und BCS (Carry Set (Greater than or equal)).

ARM64: German text placeholder GCC (Linaro) 4.9

Listing 1.89: f_signed()

Listing 1.90: f_unsigned()
German text placeholder
Versuchen Sie die Funktionen manuell hinsichtlich Größe und Entfernen redundanter Befehle zu optimieren.

MIPS
Ein wesentliches Feature von MIPS ist das Fehlen von Flags. Der Grund dafür ist offenbar, dass die Analyse von Datenabhängigkeiten dadurch vereinfacht wird.

Es gibt Befehle, die Ähnlichkeit mit SETcc in x86 haben: SLT („Set on Less Than“: vorzeichenbehaftete Version) und SLTU (Version ohne Vorzeichen). Diese Befehle setzen das Zielregister auf den Wert 1, falls die Bedingung wahr ist und ansonsten auf 0.

Das Zielregister wird dann mit BEQ („Branch on Equal“) oder BNE („Branch on Not Equal“) überprüft und gegebenenfalls ein Sprung ausgeführt. Dieses Befehlspaar muss also in MIPS für Vergleiche und Verzweigungen verwendet werden. Beginnen wir mit der vorzeichenbehafteten Version unserer Funktion:

Listing 1.91: German text placeholder GCC 4.4.5 (IDA)

```assembly
.globl f_signed
f_signed:
    sub $sp, $sp, 48
    sw $ra, 0($sp)
    sw $fp, 0($sp)
    sub $sp, $sp, 48

    addiu $sp, $sp, 48
    li $v0, 1
    syscall

.L23:
    bcc .L27 ; verzweige, falls Carry Clear (kleiner) (a<b)
    beq .L23 ; verzweige, falls gleich (a==b)
    bne .L23 ; verzweige, falls ungleich

.L23:
    addiu $sp, $sp, 48
    li $v0, 1
    syscall

.L23:
    addiu $sp, $sp, 48
    li $v0, 1
    syscall
```

Die Kommentare stammen vom Autor. Erstaunlich ist hier, dass der Compiler nicht bemerkt, dass einige Bedingungen unmöglich zu erfüllen sind, sodass Dead Code vorliegt, der nie ausgeführt werden kann.
move $fp, $sp
la $gp, __gnu_local_gp
sw $gp, 0x20+var_10($sp)

; speichere Eingabewerte auf lokalem Stack:
sw $a0, 0x20+arg_0($fp)
sw $a1, 0x20+arg_4($fp)
; reload them.
lw $v1, 0x20+arg_0($fp)
lw $v0, 0x20+arg_4($fp)

$v0=b
$v1=a

or $sat, $zero ; NOP
Pseudo-Befehl, entspricht "slt $v0,$v0,$v1"
setze $v0 auf 1 , falls $v0<$v1 (b<a) oder ansonsten auf 0:

slt $v0, $v1

; springe zu loc_5c, falls die Bedingung nicht wahr ist
Pseudo-Befehl, entspricht "beq $v0,$zero,loc_5c":

beqz $v0, loc_5c ;
gibt "a>b" aus und verlasse

or $sat, $zero ; branch delay slot, NOP

or $sat, $zero ; NOP
;

lw $v1, 0x20+arg_0($fp)
lw $v0, 0x20+arg_4($fp)
or $sat, $zero ; NOP
or $sat, $zero ; branch delay slot, NOP
;

lui $v0, (unk_238 >> 16) # "a>b"
addiu $a0, $v0, (unk_238 & 0xFFFF) # "a=b"
lw $v0, (puts & 0xFFFF)($gp)
or $sat, $zero ; NOP
move $t9, $v0
jalr $t9
or $sat, $zero ; branch delay slot, NOP

lw $v1, $v0, loc_90
or $sat, $zero ; branch delay slot, NOP
;

lui $v0, (aAB) >> 16) # "a=b"
addiu $a0, $v0, (aAB & 0xFFFF) # "a=a"
lw $v0, (puts & 0xFFFF)($gp)
or $sat, $zero ; NOP
move $t9, $v0
jalr $t9
or $sat, $zero ; branch delay slot, NOP

lw $gp, 0x20+var_10($fp)

lw $v1, 0x20+arg_0($fp)
lw $v0, 0x20+arg_4($fp)
or $sat, $zero ; NOP

; prüfe, ob $v1<$v0 (a<b), setze $v0 auf 1 , falls die Bedingung wahr ist.
slt $v0, $v1, $v0

; falls die Bedingung nicht wahr ist (d.h., $v0==0), springe nach loc_c8:
beqz $v0, loc_c8
or $sat, $zero ; branch delay slot, NOP
;

lui $v0, (aAB_0) >> 16) # "a<b"
addiu $a0, $v0, (aAB_0 & 0xFFFF) # "a<b"
lw $v0, (puts & 0xFFFF)($gp)
or $sat, $zero ; NOP
move $t9, $v0
jalr $t9
or $sat, $zero ; branch delay slot, NOP

lw $gp, 0x20+var_10($fp)

; alle 3 Bedingungen waren falsch, also nur verlassen

move $sp, $fp

lw $ra, 0x20+arg_4($sp)
lw $fp, 0x20+var_8($sp)
SLT REG0, REG0, REG1 wird von IDA auf seine kürzere Form reduziert:
SLT REG0, REG1. Wir finden dort auch den Pseudo-Befehl BEQZ („Branch if Equal to Zero“), die BEQ, $ZERO, LABEL entspricht.
Die vorzeichenlose Version ist identisch, aber

Listing 1.92: German text placeholder GCC 4.4.5 (IDA)
1.12.2 Betrag berechnen

Eine einfache Funktion:

```c
int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
}
```

German text placeholder MSVC

Normalerweise wird folgender Code erzeugt:

```
Listing 1.93: German text placeholder MSVC 2012 x64

i$ = 8
my_abs PROC
; ECX = input
    test ecx, ecx
; prüfe Vorzeichen des Eingabewertes
    jns SHORT $LN2@my_abs
; negiere Wert
    neg ecx
$LN2@my_abs:
; berechne Ergebnis in EAX:
    mov eax, ecx
    ret
my_abs ENDP
```

GCC 4.9 macht ungefähr das gleiche.

German text placeholder Keil 6/2013: German text placeholder

```
Listing 1.94: German text placeholder Keil 6/2013: German text placeholder

my_abs PROC
    CMP r0,#0
; ist Eingabewert größer gleich 0?
    BGE  |L0.6|
; falls ja, überspringe RSBS Befehl
    RSBS |L0.6|
; subtrahiere Eingabewert von 0:
    R0, r0,#0
|L0.6|
    BX  lr
ENDP
```

Germantextplaceholder Keil 6/2013: German text placeholder

Es ist im ARM mode möglich, einigen Befehlen condition codes hinzuzufügen und genau das tut der Keil Compiler:

```
my_abs PROC
  CMP r0,#0
  ; führe "Reverse Subtract" Befehl nur aus, falls Eingabewert kleiner als 0 ist:
  RSBLT r0,r0,#0
  BX lr
ENDP
```

Jetzt sind keine bedingten Sprünge mehr übrig und das ist vorteilhaft: ?? on page ??.

Germantextplaceholder GCC 4.9 (ARM64)

ARM64 kennt den Befehl NEG zum Negieren:

```
my_abs:
  sub sp, sp, #16
  str w0, [sp,12]
  ldr w0, [sp,12]
  ; vergleiche Eingabewert mit dem Inhalt des WZR Registers
  ; (welches immer 0 enthält)
  cmp w0, wzr
  bge .L2
  neg w0, w0
  .b .L3
  ldr w0, [sp,12]
  .L3:
  add sp, sp, 16
ret
```

MIPS

```
my_abs:
  ; springe, falls $a0<0:
  bltz $a0, locret_10
  ; gib Eingabewert ($a0) in $v0 zurück:
  move $v0, $a0
  jr $ra
  or $at, $zero ; branch delay slot, NOP
locret_10:
  ; negiere Eingabewert und speichere ihn in $v0:
  jr $ra
  ; dies ist ein Pseudo-Befehl. Er entspricht "subu $v0,$zero,$a0" ($v0=0-$a0)
  negu $v0, $a0
```


Verzweigungslose Version?

Man kann auch eine verzweigungslose Version dieses Codes erzeugen. Dies werden wir später betrachten: ?? on page ??.
### 1.12.3 Ternärer Vergleichsoperator

Der ternäre Vergleichsoperator in C/C++ hat die folgende Syntax:

```
expression ? expression : expression
```

Hier ist ein Beispiel:

```c
const char* f (int a)
{
    return a==10 ? "it is ten" : "it is not ten";
}
```

#### x86

Alte und nicht optimerende Compiler erzeugen Assemblercode als wenn ein if/else Ausdruck verwendet wurde:

**Listing 1.98: German text placeholder MSVC 2008**

```assembly
$SG746 DB 'it is ten', 00H
$SG747 DB 'it is not ten', 00H
tv65 = -4 ; wird als temporäre Variable benutzt
_a$ = 8
_f PROC
    push ebp
    mov ebp, esp
    push ecx

; vergleiche Eingabewert mit 10
    cmp DWORD PTR _a$[ebp], 10
    ; springe zu $LN3@f, falls ungleich
    jne SHORT $LN3@f

; speichere Pointer auf den String in temporärer Variable:
    mov DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
    ; springe zum Ende
    jmp SHORT $LN4@f
$LN3@f:
    ; speichere Pointer auf den String in temporärer Variable:
    mov DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
    ; beenden. Kopiere Pointer auf den String aus temporärer Variable nach EAX.
    mov eax, DWORD PTR tv65[ebp]
    mov esp, ebp
    pop ebp
    ret 0
_f ENDP
```

**Listing 1.99: German text placeholder MSVC 2008**

```assembly
$SG792 DB 'it is ten', 00H
$SG793 DB 'it is not ten', 00H
_a$ = 8 ; size = 4
_f PROC
; vergleiche Eingabewert mit 10
    cmp DWORD PTR _a$[esp-4], 10
    ; springe zu $LN4@f, falls gleich
    je SHORT $LN4@f

$LN4@f:
    ; beenden. Kopiere Pointer auf den String aus temporärer Variable nach EAX.
    mov eax, OFFSET $SG792 ; 'it is ten'
    ; springe zum Ende
    jmp SHORT $LN4@f
$LN4@f:
    ; speichere Pointer auf den String in temporärer Variable:
    mov DWORD PTR tv65[ebp], OFFSET $SG792 ; 'it is ten'
    ; springe zum Ende
    jmp SHORT $LN4@f

$LN4@f:
    ; beenden. Kopiere Pointer auf den String aus temporärer Variable nach EAX.
    mov eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
    ; beenden. Kopiere Pointer auf den String aus temporärer Variable nach EAX.
    mov eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
    ; beenden. Kopiere Pointer auf den String aus temporärer Variable nach EAX.
    mov eax, OFFSET $SG793 ; 'it is not ten'

Neuere Compiler sind ein wenig präziser:
Listing 1.100: German text placeholder MSVC 2012 x64

```assembly
a$ = 8
f PROC
    ; lade Pointer auf beide Strings
    lea rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
    lea rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
    ; vergleiche Eingabewert mit 10
    cmp ecx, 10
    ; falls gleich, kopiere Wert aus RDX ("it is ten")
    ; falls nicht, tue nichts. Der Pointer auf den String "it is not ten" ist immernoch in RAX.
    cmove rax, rdx
    ret 0
f ENDP
```

German text placeholder GCC 4.8 für x86 verwendet ebenfalls den CMOVcc Befehl, wohingegen der nicht optimierende GCC 4.8 bedingte Sprünge verwendet.

**ARM**

German text placeholder Keil im ARM mode verwendet ebenfalls bedingte Sprungbefehle ADRcc:

Listing 1.101: German text placeholder Keil 6/2013 (German text placeholder)

```assembly
f PROC
    ; vergleiche Eingabewert mit 10
    CMP r0,#0xa
    ; falls Operanden gleich, kopiere Pointer auf den "it is ten" String nach R0
    ADREQ r0,|L0.16| ; "it is ten"
    ; falls Operanden ungleich, kopiere Pointer auf den "it is not ten" String nach R0
    ADRNE r0,|L0.28| ; "it is not ten"
    BX lr
    ENDP

|L0.16|
DCB "it is ten",0

|L0.28|
DCB "it is not ten",0
```

Ohne manuellen Eingriff können die beiden Befehle ADREQ und ADRNE nicht in einem Durchlauf ausgeführt werden.

German text placeholder Keil für Thumb mode muss bedingte Sprungbefehle verwenden, da es keine Ladebefehle gibt, die Bedingungsflags unterstützen.

Listing 1.102: German text placeholder Keil 6/2013 (German text placeholder)

```assembly
f PROC
    ; vergleiche Eingabewert mit 10
    CMP r0,#0xa
    ; springe zu |L0.8|, falls gleich
    BEQ |L0.8|
    ADR r0,|L0.12| ; "it is not ten"
    BX lr
    |L0.8|
    ADR r0,|L0.28| ; "it is ten"
    BX lr
    ENDP

|L0.12|
DCB "it is not ten",0

|L0.28|
DCB "it is ten",0
```

**ARM64**

German text placeholder GCC (Linaro) 4.9 für ARM64 verwendet auch bedingte Sprünge:

**MIPS**

Leider ist GCC 4.4.5 für MIPS auch nicht besser:

```
$LC0: 
.ascii "it is not ten\000"

$LC1: 
.ascii "it is ten\000"

f: 
li $2,10 # 0xa
; vergleiche $a0 und 10, springe, falls gleich:
  beq $4,$2,$L2
  nop ; branch delay slot

; lasse Adresse des "it is not ten" Strings in $v0 und beende:
  lui $2,%hi($LC0)
  j $31
  addiu $2,$2,%lo($LC0)

$L2: 
; lasse Adresse des "it is ten" Strings in $v0 und beende:
  lui $2,%hi($LC1)
  j $31
  addiu $2,$2,%lo($LC1)
```

**Schreiben wir es mit if/else**

```c
const char* f (int a)
{
    if (a==10)
        return "it is ten";
    else
        return "it is not ten";
}
```

Interessanterweise war der optimierende GCC 4.8 für x86 ebenfalls in der Lage CMOVcc hier zu verwenden:

```
.LC0: 
.string "it is ten"

.LC1: 
.string "it is not ten"
```

126
German text placeholder Keil im ARM mode erzeugt identischen Code zu Listing 1.101. Der optimierende MSVC 2012 ist hingegen (noch) nicht so gut.

Fazit

Warum versuchen optimierende Compiler bedingte Sprünge zu entfernen? Mehr dazu finden Sie hier: ?? on page ??.

1.12.4 Minimale und maximale Werte berechnen

32-bit

```c
int my_max(int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
};

int my_min(int a, int b)
{
    if (a<b)
        return a;
    else
        return b;
};
```

Listing 1.106: German text placeholder MSVC 2013

```asm
_a$ = 8
_b$ = 12
_my_min PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    ; vergleiche A und B:
    cmp eax, DWORD PTR _b$[ebp]
    ; springe, falls A größer gleich B:
    jge SHORT $LN2@my_min
    ; lade A ansonsten erneut nach EAX und springe zum Ende
    mov eax, DWORD PTR _a$[ebp]
    jmp SHORT $LN3@my_min
    jmp SHORT $LN3@my_min ; redundantes JMP
$LN2@my_min:
    ; return B
    mov eax, DWORD PTR _b$[ebp]
$LN3@my_min:
    pop ebp
    ret
_my_min ENDP

_a$ = 8
_b$ = 12
_my_max PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
```
Diese beiden Funktionen unterscheiden sich nur hinsichtlich der bedingten Sprungbefehle: JGE („Jump if Greater or Equal“) wird in der ersten verwendet und JLE („Jump if Less or Equal“) in der zweiten. Hier gibt es jeweils einen unnötigen JMP Befehl pro Funktion, den MSVC wahrscheinlich fehlerhafterweise dort belassen hat.

**Verzweigungslos**

ARM im Thumb mode erinnert uns an den x86 Code:

Listing 1.107: German text placeholder Keil 6/2013 (German text placeholder)

```assembly
my_max PROC
; R0=A
; R1=B
; vergleiche A und B:
    CMP r0, r1
; verzweige, falls A größer B:
    BGT |L0.6|
; ansonsten (A<=B) liefere R1 (B) zurück:
    MOVS r0, r1
|L0.6|
; return
    BX lr
ENDP

my_min PROC
; R0=A
; R1=B
; vergleiche A und B:
    CMP r0, r1
; verzweige, falls A kleiner B:
    BLT |L0.14|
; ansonsten (A>=B) liefere R1 (B) zurück:
    MOVS r0, r1
|L0.14|
; Rückgabe
    BX lr
ENDP
```

Die Funktionen unterscheiden sich in den Verzweigebefehlen: BGT und BLT. Es ist möglich im ARM mode conditional codes zu verwenden, sodass der Code kürzer ist.

MOVcc wird nur ausgeführt, wenn die Bedingung erfüllt (d.h. wahr) ist:

Listing 1.108: German text placeholder Keil 6/2013 (German text placeholder)

```assembly
my_max PROC
; R0=A
; R1=B
; vergleiche A und B:
    CMP r0, r1
; gib B anstatt A zurück, indem B nach R0 geschrieben wird
; dieser Befehl wird nur ausgeführt, falls A<=B (deshalb, LE - Less or Equal)
ENDP
```
wenn der Befehl nicht ausgeführt wird (im Falle von A>B), ist A immer noch im R0 Register

MOVE r0,r1
BX lr
ENDP

my_min PROC
; R0=A
; R1=B
; vergleiche A und B:
CMP r0,r1
; gib B anstatt A zurück, indem B nach R0 geschrieben wird
; dieser Befehl wird nur ausgeführt, falls A>=B (deshalb, GE - Greater or Equal)
; wenn der Befehl nicht ausgeführt wird (im Falle von A<B), ist A immer noch im R0 Register
MOVE r0,r1
BX lr
ENDP

German text placeholder GCC 4.8.1 und der optimierende MSVC 2013 können den CMOVcc Befehl verwenden, der analog zu MOVcc in ARM funktioniert:

Listing 1.109: German text placeholder MSVC 2013

64-bit

#include <stdint.h>

int64_t my_max(int64_t a, int64_t b)
{
    if (a>b)
        return a;
    else
        return b;
};

int64_t my_min(int64_t a, int64_t b)
{
    if (a<b)
        return a;
    else
        return b;
};

Hier findet ein unnötiges Verschieben von Variablen statt, aber der Code ist verständlich:
Verzweigungslös
Die Funktionargumente müssen nicht vom Stack geladen werden, da sie sich bereits in den Registern befinden:

MSVC 2013 tut beinahe das gleiche:
ARM64 verfügt über den CSEL Befehl, der genau wie MOVcc in ARM oder CMOVcc in x86 arbeitet; er hat lediglich einen anderen Namen: „Conditional SELet“.

my_max:
sub   sp, sp, #16
str   x0, [sp,8]
str   x1, [sp]
ldr   x1, [sp]
ldr   x0, [sp]
cmp   x1, x0
ble   .L2
ldr   x0, [sp,8]
b     .L3
.L2:
    ldr   x0, [sp]
.L3:
    add   sp, sp, 16
    ret

my_min:
sub   sp, sp, #16
str   x0, [sp,8]
str   x1, [sp]
ldr   x1, [sp]
ldr   x0, [sp]
cmp   x1, x0
bge   .L5
ldr   x0, [sp,8]
b     .L6
.L5:
    ldr   x0, [sp]
.L6:
    add   sp, sp, 16
    ret
; vergleiche A und B:
cmp x0, x1
; setze X0 (A) auf X0, falls X0>=X1 oder A>=B (größer gleich)
; setze X1 (B) auf X0, falls A<B
csel x0, x0, x1, ge
ret

my_min:
; X0=A
; X1=B
; vergleiche A und B:
cmp x0, x1
; setze X0 (A) auf X0, falls X0<=X1 oder A<=B (kleiner gleich)
; setze X1 (B) auf X0, falls A>B
csel x0, x0, x1, le
ret

MIPS

Leider ist GCC 4.4.5 für MIPS nicht so gut:

Listing 1.113: German text placeholder GCC 4.4.5 (IDA)

my_max:
; setze $v1 auf 1 , falls $a1<$a0, ansonsten (falls $a1>$a0) lösche:
slt $v1, $a1, $a0
; springe, falls $v1 ist 0 (oder $a1>$a0):
beqz $v1, locret_10
; dies ist der branch delay slot
; bereite $a1 in $v0 vor, falls verzweigt wird
move $v0, $a1
; wird nicht verzweigt, bereite $a0 in $v0 vor:
move $v0, $a0

locret_10:
    jr $ra
    or $at, $zero ; branch delay slot, NOP

; die min() Function ist identisch, aber die Eingabeoperanden des SLT Befehls sind vertauscht.
my_min:
    slt $v1, $a0, $a1
    beqz $v1, locret_28
    move $v0, $a1
    move $v0, $a0

locret_28:
    jr $ra
    or $at, $zero ; branch delay slot, NOP

Vergessen Sie nicht die branch delay slots: der erste MOVE wird vor BEQZ ausgeführt, der zweite MOVE wird nur dann ausgeführt, wenn die Verzweigung nicht genommen wird.

1.12.5 Fazit

x86

Hier ist der grobe Aufbau eines bedingten Sprungs:

Listing 1.114: x86

CMP register, register/value
Jcc true ; cc=condition code
false:
... dieser Code wird ausgeführt, wenn der Vergleich falsch ergibt ...
JMP exit
ture:
... dieser Code wird ausgeführt, wenn der Vergleich wahr ergibt ...
exit:
ARM

Listing 1.115: ARM

```
CMP register, register/value
Bcc true ; cc=condition code
false:
... dieser Code wird ausgeführt, wenn der Vergleich falsch ergibt ...
JMP exit
true:
... dieser Code wird ausgeführt, wenn der Vergleich wahr ergibt ...
exit:
```

MIPS

Listing 1.116: prüfe auf Null

```
BEQZ REG, label
...
```

Listing 1.117: Prüfe auf kleiner Null

```
BLTZ REG, label
...
```

Listing 1.118: Prüfe auf Gleichheit

```
BEQ REG1, REG2, label
...
```

Listing 1.119: Prüfe auf Ungleichheit

```
BNE REG1, REG2, label
...
```

Listing 1.120: Prüfe auf größer, größer als (vorzeichenbehaftet)

```
BEQ REG1, label
...
```

Listing 1.121: Prüfe auf kleiner, kleiner als (vorzeichenlos)

```
BEQ REG1, label
...
```

Ohne Verzweigung

Wenn der Rumpf eines bedingten Ausdrucks sehr kurz ist, kann der bedingten Move-Befehl verwendet werden: MOVcc in ARM (in ARM mode), CSEL in ARM64, CMOVcc in x86.

ARM

Es ist im ARM mode möglich, Bedingungssuffixe (engl. condition code) für manchen Befehle zu verwenden:

```
CMP register, register/value
instr1_cc ; dieser Befehl wird ausgeführt, wenn der condition code falsch ergibt
instr2_cc ; dieser Befehl wird ausgeführt, wenn der condition code wahr ergibt
...
etc...
```

Ohne Verzweigung

Wenn der Rumpf eines bedingten Ausdrucks sehr kurz ist, kann der bedingten Move-Befehl verwendet werden: MOVcc in ARM (in ARM mode), CSEL in ARM64, CMOVcc in x86.
Natürlich gibt es keine Obergrenze für die Anzahl an Befehlen mit conditional codes, solange diese die CPU Flags nicht verändern.

Im Thumb mode gibt es den IT Befehl, der es erlaubt, zusätzliche Suffixe an die vier folgenden Befehle anzuheften. Mehr dazu unter: 1.17.7 on page 238.

Listing 1.123: ARM (German text placeholder)

CoMP register, register/value
ITEEE EQ ; setze folgende Suffixe: if-then-else-else-else
instr1 ; Befehl wird ausgeführt, wenn Bedingung wahr ist
instr2 ; Befehl wird ausgeführt, wenn Bedingung falsch ist
instr3 ; Befehl wird ausgeführt, wenn Bedingung falsch ist
instr4 ; Befehl wird ausgeführt, wenn Bedingung falsch ist

1.12.6 German text placeholder

(ARM64) Versuchen Sie den Code in Listing.1.103 so neu zu schreiben, dass alle Befehle mit bedingten Sprüngen durch den CSEL Befehl ersetzt werden.

1.13 switch()/case/default

1.13.1 Kleine Anzahl von Fällen

#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default: printf ("something unknown\n"); break;
    
    };

int main()
{
    f (2); // test
};

x86

German text placeholder MSVC

Ergebnis (MSVC 2010):

Listing 1.124: MSVC 2010

    tv64 = -.4 ; size = 4
   _a$ = 8 ; size = 4
-f  PROC
  push ebp
  mov ebp, esp
  push ecx
  mov eax, DWORD PTR _a$[ebp]
  mov DWORD PTR tv64[ebp], eax
  cmp DWORD PTR tv64[ebp], 0
  je SHORT $LN4@f
  cmp DWORD PTR tv64[ebp], 1
  je SHORT $LN3@f
  cmp DWORD PTR tv64[ebp], 2
  je SHORT $LN2@f
  jmp SHORT $LN1@f
Unsere Funktionen mit ein paar Fällen in switch() ist analog zu dieser Konstruktion:

```c
void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
}
```

Wenn wir mit einem switch() mit einigen wenigen Fällen arbeiten, ist es unmöglich sicher zu sein, dass es sich tatsächlich im Quellcode um ein switch() handelt und nicht um eine Reihe von if()-Anweisungen. Das bedeutet, dass switch() ein syntaktischer Zucker für eine große Anzahl von verschachtelten if()-Anweisungen ist.

Hier ist nichts Neues für uns im erzeugten Code, mit der Ausnahme, dass der Compiler die Eingabevariable `a` in einer temporären Variable `tv64` speichert87.

Wenn wir diesen Code mit GCC 4.4.1 kompilieren, erhalten wir fast das gleiche Ergebnis, sogar unter Verwendung maximaler Optimierung (Option -O3).

**German text placeholder MSVC**

Aktivieren wir nun Optimierung in MSVC (/Ox): `cl 1.c /Fa1.asm /Ox`

**Listing 1.125: MSVC**

87Lokale Variablen im Stack haben den Präfix tv—so benennt MSVC interne Variablen für seine Zwecke
Hier finden wir ein paar schmutzige Tricks.

Zunächst: der Wert von \( a \) wird in EAX abgelegt und 0 wird davon abgezogen. Klingt absurd, muss aber getan werden, um zu prüfen, ob der Wert in EAX null ist. Falls ja, wird das ZF Flag gesetzt (denn 0 minus 0 ist 0) und der erste bedingte Sprung JE (Jump if Equal) oder synonym JZ —Jump if Zero) wird ausgeführt und der Control Flow wird an das Label $LN4@f$ übergeben, an dem die Nachricht 'zero' ausgegeben wird. Wenn der erste Sprung nicht ausgeführt wird, wird 1 vom Eingabewert abgezogen und wenn an irgendeinem Punkt der Ausführung ein Ergebnis von 0 auftritt, wird der zugehörige Sprung ausgeführt.

Falls aber keiner der Sprünge ausgeführt wird, wird der Control Flow mit dem String Argument 'something unknown' an printf() übergeben.

Zweitens: wir sehen etwas für uns Ungewohntes: ein Pointer auf einen String wird in der Variablen \( a \) abgelegt und anschließend wird printf() nicht über CALL, sondern per JMP aufgerufen. Es gibt hierfür eine einfache Erklärung: Der Aufrufer legt einen Wert auf dem Stack ab und ruft unsere Funktion über CALL auf. CALL selbst legt die Rücksprungadresse (RA) auf dem Stack ab und macht einen unbedingten Sprung zur Adresse unserer Funktion. Unsere Funktion hat an jedem Punkt der Ausführung (da sie keine Befehle enthält, die den Stackpointer verändern) das folgende Stacklayout:

- ESP—zeigt auf RA
- ESP+4—zeigt auf die Variable \( a \)

Wenn wir andererseits printf() aufrufen benötigen wir hier exakt das gleiche Stacklayout mit dem Unterschied des ersten Arguments von printf(), welches auf den auszugebenden String zeigt. Unser Code tut hier das Folgende:

Er ersetzt das erste Argument der Funktion mit der Adresse (d.i. dem Pointer) auf den String und springt zu printf() als ob wir nicht unsere Funktion f() (wie direkt printf() aufrufen würden. Die Funktion printf() gibt einen String auf German text placeholder aus und führt dann den RET Befehl aus, der die RA vom Stack holt. Der Control Flow wird nicht an f() übergeben, sondern an den Aufrufer von f(), womit effektiv die Funktion f() umgangen wird.

All dies ist möglich, da printf() in allen Fällen ganz am Ende der Funktion f() aufgerufen wird. In gewisser Weise besteht Ähnlichkeit zur Funktion longjmp()

Ein ähnlicher Fall mit dem ARM Compiler wird im Abschnitt „German text placeholder“ beschrieben: ( ?? on page ??)
Da dieses Beispiel kompliziert ist, untersuchen wir es mit OllyDbg.

OllyDbg kann solche switch() Konstruktionen erkennen und einige nützliche Kommentare hinzufügen. EAX ist zu Beginn auf 2 gesetzt, das entspricht dem Eingabewert der Funktion:

Abbildung 1.43: OllyDbg: EAX enthält jetzt das erste (und einzige) Functionsargument
0 wird in vor der 2 in EAX abgezogen. Natürlich enthält EAX danach immer noch den Wert 2, aber das ZF-Flag ist jetzt 0, da das Ergebnis der letzten Berechnung nicht 0 ergeben hat.

Abbildung 1.44: OllyDbg: SUB wurde ausgeführt
DEC wird ausgeführt und EAX enthält nun 1. Da 1 ein von null verschiedener Wert ist, ist das ZF Flag immer noch 0:

Abbildung 1.45: OllyDbg: erstes DEC wurde ausgeführt
Das nächste DEC wird ausgeführt. EAX enthält jetzt 0 und das ZF Flag wird gesetzt, da eine Berechnung 0 ergeben hat.

Abbildung 1.46: OllyDbg: zweites DEC wurde ausgeführt

OllyDbg zeigt an, dass dieser Sprung jetzt getätigt wird.
Ein Pointer auf den String „two“ wird jetzt auf den Stack geschrieben:

Abbildung 1.47: OllyDbg: Pointer auf den String wird an die Stelle des ersten Arguments geschrieben

Man beachte: das aktuelle Argument der Funktion ist 2 und diese 2 befindet sich im Stack nun an der Adresse 0x001EF850.
MOV schreibt den Pointer auf den String an die Adresse 0x001EF850 (siehe Stackfenster). Dann wird der Sprung ausgeführt. Dies ist der erste Befehl der Funktion `printf()` in MSVCR100.DLL. (Dieses Beispiel wurde mit der Option /MD kompiliert.)

Abbildung 1.48: OllyDbg: erster Befehl von `printf()` in MSVCR100.DLL

Die Funktion `printf()` behandelt den String an der Adresse 0x00FF3010 als (einziges) Argument und gibt ihn aus.
Dies ist der letzte Befehl von printf():

Abbildung 1.49: OllyDbg: letzter Befehl von printf() in MSVCRI100.DLL

Der String „two“ wurde gerade auf der Konsole ausgegeben.
Wir drücken nun F7 oder F8 (German text placeholder) und kehren...nicht zu f(), sondern zu main() zurück:

Abbildung 1.50: OllyDbg: zurück zu main()

Dieser Sprung wird direkt von printf() zu main() durchgeführt, da RA im Stack nicht auf eine Stelle in f(), sondern auf main() zeigt. Der Befehl CALL 0x00FF1000 war der eigentliche Befehl, der f() aufgerufen hat.

ARM: German text placeholder Keil 6/2013 (German text placeholder)

Auch hier können wir bei Untersuchung des Code nicht sagen, ob im Quellcode ein switch() oder eine Folge von if()-Ausdrücken vorliegt.

Wir finden hier Befehle mit Prädikaten wieder (wie ADREQ (Equal)), welcher nur dann ausgeführt wird, wenn R0 = 0 und dann die Adresse des Strings "zero\n" nach R0 lädt.

Der folgende BEQBefehl überlässt den Control Flow an loc_170, falls R0 = 0. Ein aufmerksamer Leser könnte sich fragen, ob BEQ korrekt ausgelöst wird, da ADREQ das R0 Register bereits mit einem anderen Wert befüllt hat. Es wird korrekt ausgelöst, da BEQ die Flags, die vom CMP Befehl gesetzt wurden, prüft und ADREQ die Flags nicht verändert.

Die übrigen Befehle kennen wir bereits. Es gibt nur einen Aufruf von printf() am Ende und wir haben diesen Trick bereits hier kennengelernt (?? on page ??). Am Ende gibt es drei Wege zur Ausführung von printf().

89(PowerPC, ARM) Branch if Equal
Der letzte Befehl, CMP R0, #2, wird benötigt, um zu prüfen, ob \( a = 2 \). Wenn dies nicht der Fall ist, lädt ADRNE einen Pointer auf den String «something unknown \n» nach R0, da \( a \) bereits auf Gleichheit mit 0 oder 1 geprüft wurde und wir können sicher sein, dass die Variable \( a \) an dieser Stelle keinen dieser beiden Werte enthält. Falls \( R0 = 2 \) ist, lädt ADREQ einen Pointer auf den String «two\n» nach R0.

**ARM: German text placeholder Keil 6/2013 (German text placeholder)**

```
.text:000000D4
  f1:  
.text:000000D4 10 B5  PUSH  {R4,LR}  
.text:000000D6 00 28  CMP  R0, #0  
.text:000000D8 05 D0  BEQ  zero_case  
.text:000000DA 01 28  CMP  R0, #1  
.text:000000DC 05 D0  BEQ  one_case  
.text:000000DE 02 28  CMP  R0, #2  
.text:000000E0 05 D0  BEQ  two_case  
.text:000000E2 91 A0  ADR  R0, aSomethingUnknown ; "something unknown\n"  
.text:000000E4 04 E0  B  default_case  

.zero_case:  ; CODE XREF: f1+4  
.text:000000E6 95 A0  ADR  R0, aZero ; "zero\n"  
.text:000000E8 02 E0  B  default_case  

.one_case:  ; CODE XREF: f1+8  
.text:000000EA 96 A0  ADR  R0, aOne ; "one\n"  
.text:000000EC 00 E0  B  default_case  

.two_case:  ; CODE XREF: f1+C  
.text:000000EE 97 A0  ADR  R0, aTwo ; "two\n"  
.text:000000F0 00 E0  B  default_case  ; CODE XREF: f1+10  

.default_case  ; f1+14  
.text:000000F4 10 BD  POP  {R4,PC}  
```

Wie bereits erwähnt ist es bei den meisten Befehlen im Thumb mode nicht möglich Prädikate für Bedingungen hinzuzufügen, sodass der Thumb-Code hier dem leicht verständlichen x86 CISC-style Code sehr ähnlich ist.

**ARM64: German text placeholder GCC (Linaro) 4.9**

```
.LC12:  .string "zero"  
.LC13:  .string "one"  
.LC14:  .string "two"  
.LC15:  .string "something unknown"  

f12:  stp  x29, x30, [sp, -32]!  
     add  x29, sp, 0  
     str  w0, [x29,28]  
     ldr  w0, [x29,28]  
     cmp  w0, 1  
     beq  .L34  
     cmp  w0, 2  
     beq  .L35  
     cmp  w0, wzr  
     bne  .L38  ; jump to default label  
     adrp  x0, .LC12  ; "zero"  
     add  x0, x0, :lo12:.LC12  
     bl  puts  
     b  .L32  

.L34:  
     adrp  x0, .LC13  ; "one"  
     add  x0, x0, :lo12:.LC13  
     bl  puts  
     b  .L32  

.L35:  ```
Der Datentyp des Eingabewertes ist int, deshalb wird das Register W0 anstatt des X0 Registers verwendet, um ihn aufzunehmen.

Die Pointer auf die Strings werden an puts() mit einem ADRP/ADD Befehlspaar übergeben, genauso wie wir es im „Hallo, Welt!” Beispiel gezeigt haben: 1.5.3 on page 23.

ARM64: German text placeholder GCC (Linaro) 4.9

Ein besser optimiertes Stück Code. Der Befehl CBZ (Compare and Branch on Zero) springt, falls W0 gleich null ist. Es gibt auch einen direkten Sprung zu puts() anstelle eines Aufrufs, so wie bereits hier erklärt: 1.13.1 on page 134.

MIPS

Listing 1.126: German text placeholder GCC 4.4.5 (IDA)


**Fazit**

*German text placeholder* Listing.1.13.1.

### 1.13.2 Viele Fälle

Wenn ein switch() Ausdruck viele Fälle enthält, ist es für den Compiler nicht günstig sehr großen Code mit vielen JE/JNE Befehlen zu erzeugen.

```c
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    case 3: printf ("three\n"); break;
    case 4: printf ("four\n"); break;
    default: printf ("something unknown\n"); break;
    }
}

int main()
{
    f (2); // test

```
Wir erhalten (MSVC 2010):

Listing 1.127: MSVC 2010

```assembly
.tv64 = -4 ; size = 4
.a$ = 8 ; size = 4
_f          PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 4
    ja      SHORT $LN1@f
    mov     ecx, DWORD PTR tv64[ebp]
    jmp     DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push    OFFSET $SG739 ; 'zero', 0AH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN5@f:
    push    OFFSET $SG741 ; 'one', 0AH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN4@f:
    push    OFFSET $SG743 ; 'two', 0AH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN3@f:
    push    OFFSET $SG745 ; 'three', 0AH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN2@f:
    push    OFFSET $SG747 ; 'four', 0AH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN1@f:
    push    OFFSET $SG749 ; 'something unknown', 0AH, 00H
    call    _printf
    add     esp, 4
$LN9@f:
    mov     esp, ebp
    pop     ebp
    ret     0
    npad    2 ; Padding für nächstes Label
$LN11@f:
    DD      $LN6@f ; 0
    DD      $LN5@f ; 1
    DD      $LN4@f ; 2
    DD      $LN3@f ; 3
    DD      $LN2@f ; 4
_f          ENDP
```

Was wir hier sehen ist eine Ansammlung von Aufrufen von `printf()` mit diversen Argumenten. Alle haben nicht Adressen im Speicher des Prozesses, sondern auch interne symbolische Labels, die ihnen vom Compiler zugewiesen werden. Alle diese Labels werden auch in der internen Tabelle $LN11@f aufgeführt.

Zu Beginn der Funktion wird der Control Flow an das Label $LN1@f abgegeben, wenn $a$ größer ist als 4. An diesem Label wird `printf()` mit dem Argument 'something unknown' aufgerufen.

Wenn aber der Wert von $a$ kleiner gleich 4 ist, dann wird dieser mit 4 multipliziert und mit der Tabellenadresse
$\text{LN11@f}$ addiert. Auf diese Weise wird die Adresse innerhalb der Tabelle konstruiert und zeigt genau auf das gewünschte Element. Nehmen wir zum Beispiel an, dass $a$ gleich 2 ist. $2 \cdot 4 = 8$ (alle Tabellenelemente sind Adressen in einem 32-Bit-Prozess und haben daher eine Breite von 4 Bytes). Die Adresse an der Stelle $\text{LN11@f + 8}$ ist das Tabellenelement, an dem das Label $\text{LN4@f}$ gespeichert ist. JMP holt die Adresse $\text{LN4@f}$ aus der Tabelle und springt dorthin.

Diese Tabelle wird manchmal Jumptable oder Verzweigungstabelle genannt\textsuperscript{90}.

Dann wird das zugehörige printf() mit dem Argument 'two' aufgerufen.

Der Befehl TTjmp DWORD PTR $\text{LN11@f[ecx*4]}$ bedeutet dabei springe zum an dieser Stelle gespeicherten DWORD$\text{LN11@f + ecx * 4}$.

\texttt{npad (.1.7 on page 542)} ist ein Assemblermakro, dass das nächste Label so angeordnet, dass es an einer 4 Byte (oder 16 Bit) Adressgrenze gespeichert wird. Das ist für den Prozessor sehr praktisch, da er die 32-Bit-Werte aus dem Speicher durch den Speicherbus, den Cache, etc. in effektiverer Weise laden kann.

\textsuperscript{90}Die ganze Methode wurde in früheren Versionen von Fortran berechnetes GOTO genannt: wikipedia. Heutzutage zwar nicht mehr relevant, aber welch ein Ausdruck!
Untersuchen wir das Beispiel in OllyDbg. Der Eingabewert der Funktion (2) wird nach EAX geladen:

Abbildung 1.51: OllyDbg: das Funktionsargument wird nach EAX geladen
Es wird geprüft, ob der Eingabewert größer als 4 ist. Falls nicht, wird der „default“ Sprung nicht ausgeführt:

Abbildung 1.52: OllyDbg: 2 ist nicht größer als 4: kein Sprung wird ausgeführt
Hier sehen wir eine Jumptable:

Wir haben „Follow in Dump“ → „Address constant“ geklickt, sodass wir jetzt die Jumptable im Datenfenster sehen. Hier sind 5 32-Bit-Werte. ECX ist jetzt 2, sodass das zweite Element (beginnend bei null) der Tabelle verwendet wird. Es ist auch möglich durch Klicken auf qFollow in Dump → „Memory address“ in OllyDbg das Element, das durch den JMP Befehl angesteuert wird, anzeigen zu lassen. Dieses Element ist hier 0x010B103A.

Abbildung 1.53: OllyDbg: Zieladresse mit Jumptable berechnen

91 Diese werden von OllyDbg unterstrichen, da es auch FIXUPS sind: 6.5.2 on page 474, wir kommen später darauf zurück.
Nach dem Sprung sind wir an der Stelle 0x010B103A: der Code zur Ausgabe von „two“ wird jetzt ausgeführt:

Abbildung 1.54: OllyDbg: jetzt sind wir am case: Label

German text placeholder GCC

Schauen wir was GCC 4.4.1 erzeugt:

Listing 1.128: GCC 4.4.1
Es ist bis auf eine Nuance das gleiche: das Argument arg_0 wird mit 4 multipliziert durch eine Verschiebung von 2 Bits nach links (dies entspricht einer Multiplikation mit 4) (1.16.2 on page 194). Dann wird die Adresse des Labels vom off_804855C genommen, die in EAX gespeichert wird, und dann wird mit JMP EAX der eigentliche Sprung durchgeführt.

**ARM: German text placeholder Keil 6/2013 (German text placeholder)**

Listing 1.129: German text placeholder Keil 6/2013 (German text placeholder)
Dieser Code verwendet das ARM mode Feature, das alle Befehle eine feste Länge von 4 Byte haben. Vergessen wir nicht, dass der Maximalwert für a beträgt und jeder größere Wert zur Ausgabe des Strings führt.

Der erste CMP R0, #5 Befehl vergleicht den Eingabewert a mit 5. Der nächste ADDCC PC, PC, R0, LSL#2 Befehl wird nur ausgeführt, falls R0 < 5 (CC=Carry clear/kleiner als). Wenn ADDCC nicht ausgeführt wird (d.h. R0 \geq 5), wird ein Sprung zum default_case Label ausgeführt.

Aber wenn R0 < 5 und ADDCC ausgeführt wird, wird das Folgende geschehen:

Der Wert in R0 wird mit 4 multipliziert. Der Suffix am Befehl steht dabei für „shift left by 2 bits“.

Im Moment der Ausführung von ADDCC ist der Wert von PC (0x180) 8 Bytes - oder mit anderen Worten: 2 Befehle - größer als die Adresse, an der sich der ADDCC Befehl befindet (0x178).

Jedes Mal wenn a um 1 erhöht wird, erhöht sich der PC um 4. Dabei ist 4 die Länge eines Befehls im ARM mode und auch die Länge jedes B Befehls, von denen sich hier 5 befinden.

Jeder dieser fünf B Befehle gibt den Control Flow weiter so wie es im switch() Ausdruck programmiert wurde.

Hier werden jeweils die Pointer auf die zugehörigen Strings geladen, etc.

\[^{92}ADD—Addition\]
Man kann sich nicht sicher sein, dass alle Befehle im Thumb und Thumb-2 mode dieselbe Größe haben. Man kann sogar sagen, dass die Befehle hier genau wie in x86 variable Längen haben.

Deshalb wird hier eine spezielle Tabelle verwendet, die Informationen darüber enthält wie viele Fälle vorliegen (ohne den Default-Case) und es wird für jeden Fall ein Label mit einem Offset für den Control Flow im zugehörigen Fall angegeben.
Hier taucht eine spezielle Funktion namens __ARM_common_switch8_thumb auf, die mit der Tabelle und der Übergabe des Control Flows umgeht. Sie beginnt mit BX PC, dessen Aufgabe es ist, den Prozessor in den ARM mode zu versetzen. Danach finden wir die Funktion für den Umgang mit der Tabelle.

Es ist hier zu fortgeschritten um weiter ins Details zu gehen, daher lassen wir es für den Moment hierbei bewenden.

Ist ist interessant festzustellen, dass die Funktion das LR Register als Pointer auf die Tabelle verwendet.

Tatsächlich enthält LR nach dem Aufruf der Funktion die Adresse nach dem Befehl BL __ARM_common_switch8_thumb, an dem die Tabelle beginnt.

Es ist auch bemerkenswert, dass der Code als eine separate Funktion erzeugt wird, um wiederverwendet werden zu können, sodass der Compiler nicht für jeden switch() Ausdruck den gleichen Code erzeugen muss.

IDA hat erfolgreich ermittelt, dass es sich um eine Servicefunktion und eine Tabelle handelt und hat Kommentare wie etwa jump table 000000FA case 0 zu den Labels hinzugefügt.

**MIPS**

Listing 1.131: German text placeholder GCC 4.4.5 (IDA)

```mips
f:
    lui $gp, (__gnu_local_gp >> 16)
    ; springe zu loc_24 , falls der Eingabewert kleiner als 5 ist:
    sltiu $v0, $a0, 5
    bnez $v0, loc_24
    la $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
    ; Eingabewert ist größer gleich 5.
    ; "something unknown" ausgeben und beenden:
    lui $a0, ($LC5 & 16) # "something unknown"
    lw $t9, (puts & 0xFFFF)($gp)
    or $at, $zero ; NOP
    jr $t9
    la $a0, ($LC5 & 0xFFFF) # "something unknown" ; branch delay slot

loc_24:           # CODE XREF: f+8
    ; lade Adresse der Jumptable
    ; LA ist ein Pseudo-Befehl, der für ein LUI und ADDIU Paar steht:
    la $v0, off_120
    ; multipliere Eingabewert mit 4:
    sll $a0, 2
    ; multiplizierten Wert und Adresse der Jumptable addieren:
    addu $a0, $v0, $a0
    ; lade Element aus Jumptable:
    lw $v0, 0($a0)
    or $at, $zero ; NOP
    ; Sprung zur Adresse in der Jumptable:
    jr $v0
    or $at, $zero ; branch delay slot, NOP

sub_44:           # DATA XREF: .rodata:0000012C
    ; "three" ausgeben und beenden
    lui $a0, ($LC3 & 16) # "three"
    lw $t9, (puts & 0xFFFF)($gp)
    or $at, $zero ; NOP
    jr $t9
    la $a0, ($LC3 & 0xFFFF) # "three" ; branch delay slot

sub_58:           # DATA XREF: .rodata:00000130
    ; "four" ausgeben und beenden
    lui $a0, ($LC4 & 16) # "four"
    lw $t9, (puts & 0xFFFF)($gp)
    or $at, $zero ; NOP
    jr $t9
    la $a0, ($LC4 & 0xFFFF) # "four" ; branch delay slot

sub_6C:           # DATA XREF: .rodata:off_120
    ; "zero" ausgeben und beenden
    lui $a0, ($LC0 & 16) # "zero"
```

156
lw  $t9, (puts & 0xFFFF)($gp)
or  $at, $zero ; NOP
jr   $t9
la   $a0, ($LC0 & 0xFFFF) # "zero" ; branch delay slot

sub_80:                                      # DATA XREF: .rodata:00000124
; "one" ausgeben und beenden
    lui  $a0, ($LC1 >> 16)   # "one"
    lw  $t9, (puts & 0xFFFF)($gp)
or  $at, $zero ; NOP
jr   $t9
la   $a0, ($LC1 & 0xFFFF) # "one" ; branch delay slot

sub_94:                                      # DATA XREF: .rodata:00000128
; "two" ausgeben und beenden
    lui  $a0, ($LC2 >> 16)   # "two"
    lw  $t9, (puts & 0xFFFF)($gp)
or  $at, $zero ; NOP
jr   $t9
la   $a0, ($LC2 & 0xFFFF) # "two" ; branch delay slot

; kann im .rodata Segment abgelegt werden:
off_120: .word  sub_6C
           .word  sub_80
           .word  sub_94
           .word  sub_44
           .word  sub_58

Der für uns neue Befehl ist SLTIU („Set on Less Than Immediate Unsigned“).
Dies ist das gleiche wie SLTU („Set on Less Than Unsigned“); das „I“ steht dabei für „immediate“, d.h. für den Befehl muss eine Zahl angegeben werden.
BNEZ ist „Branch if Not Equal to Zero“.
Der Code ist den anderen ISAs sehr ähnlich. SLL („Shift Word Left Logical“) führt eine Multiplikation mit 4 durch. Da MIPS eine 32-Bit CPU ist, sind auch die Adressen in der jumptable 32 Bit groß.

Fazit
Das grobe Gerüst eines switch():

Listing 1.132: x86

MOV REG, input
CMP REG, 4 ; maximale Anzahl von Fällen
JA  default
SHL REG, 2 ; finde Element in der Tabelle. 3 Bits schieben in x64.
MOV REG, jump_table[REG]
JMP REG
casel:   ; beliebiger Code
    JMP exit
case2:   ; beliebiger Code
    JMP exit
case3:   ; beliebiger Code
    JMP exit
case4:   ; beliebiger Code
    JMP exit
case5:   ; beliebiger Code
    JMP exit
default: ; beliebiger Code
    JMP exit

...
Der Sprung zur Adresse in der Jumptable kann auch durch den folgenden Befehl realisiert werden: 
JMP jump_table[REG*4] oder JMP jump_table[REG*8] in x64.

Eine Jumptable ist nur ein Array von Pointern, genau wie das hier beschriebene: 1.18.5 on page 262.

1.13.3 Wenn es mehrere case Ausdrücke in einem Block gibt

Hier ist eine weit verbreitete Konstruktion: mehrere case Ausdrücke für einen einzigen Block:

#include <stdio.h>
void f(int a)
{
    switch (a)
    {
    case 1:
    case 2:
    case 7:
    case 10:
        printf ("1, 2, 7, 10\n");
        break;
    case 3:
    case 4:
    case 5:
    case 6:
        printf ("3, 4, 5\n");
        break;
    case 8:
    case 9:
    case 20:
    case 21:
        printf ("8, 9, 21\n");
        break;
    case 22:
        printf ("22\n");
        break;
    default:
        printf ("default\n");
        break;
    }
};
int main()
{
    f(4);
};

Es ist zu verschwenderisch einen Block für jeden möglichen Fall zu erzeugen, sodass normalerweise ein Block und eine Art Dispatcher erzeugt werden.

MSVC

Listing 1.133: German text placeholder MSVC 2010

1 $SG2798 DB '1, 2, 7, 10', 0aH, 00H
2 $SG2800 DB '3, 4, 5', 0aH, 00H
3 $SG2802 DB '8, 9, 21', 0aH, 00H
4 $SG2804 DB '22', 0aH, 00H
5 $SG2806 DB 'default', 0aH, 00H

exit:
....
jump_table dd case1
    dd case2
    dd case3
    dd case4
    dd case5
Wir sehen hier zwei Tabellen: die erste Tabelle ($LN10@f$) ist eine Indextabelle und die zweite ($LN11@f$) ist ein Array von Pointern auf Blöcke.

Zuerst wird der Eingabewert als Index in der Indextabelle verwendet (Zeile 13).

Hier ist eine kurze Legende für die Werte in der Tabelle: 0 ist der erste case Block (für die Werte 1, 2, 7, 10), 1 ist der zweite (für die Werte 3, 4, 5), 2 ist der dritte (für die Werte 8, 9, 21), 3 ist der vierte (für die Werte 22), 4 ist der Defaultblock. Hier erhalten wir einen Index für die zweite Tabelle aus Pointern und springen zu einem solchen (Zeile 14).

Bemerkenswert ist auch, dass es keinen Case für den Eingabewert 0 gibt.

Aus diesem Grund haben wir den DEC Befehl in Zeile 10 und die Tabelle beginnt bei $a = 1$, da kein Tabellenelement für $a = 0$ angelegt werden muss.
Dies ist ein weitverbreitetes Muster. Warum ist dieses Vorgehen so ökonomisch? Warum ist es nicht möglich wie vorher in (1.13.2 on page 152) vorzugehen mit nur einer Tabelle aus Blockpointern? Der Grund hierfür ist, dass die Elemente in der Indextabelle nur 8 Bit groß sind und alles deshalb deutlich kompakter ist.

**GCC**

GCC erledigt seinen Job unter Verwendung von nur einer Pointertabelle wie bereits hier besprochen (1.13.2 on page 152).

**ARM64: German text placeholder GCC 4.9.1**

Es wird kein Code ausgeführt, wenn der Eingabewert 0 ist, weshalb GCC versucht, die Jumptable kleiner zu machen und erst beim Eingabewert 1 zu beginnen.

GCC 4.9.1 für ARM64 verwendeten einen noch ausgefeilteren Trick. Es ist möglich alle Offsets als 8 Bit Werte zu kodieren.

Erinnern wir uns, dass alle ARM64 Befehle eine Größe von 4 Bytes haben.


```
Listing 1.134: German text placeholder GCC 4.9.1 ARM64

f14:
; Eingabewert in W0
   sub   w0, w0, #1
   cmp   w0, 21
; verweigere, falls kleiner gleich (vorzeichenlos):
   bls   .L9
.L9:
; "default" ausgeben:
   adrp  x0, .LC4
   add   x0, x0, :lo12:.LC4
   b     puts

; lade Jumptableadresse von X1:
   adrp  x1, .L4
   add   x1, x1, :lo12:.L4
; W0=Eingabewert-1
; lade Byte aus der Tabelle:
   ldrb  w0, [x1,w0,uxtw]
; lade Adresse des Lrtx Labels:
   adr   x1, .Lrtx4
; multipliziere Tabellenelement mit 4 (durch Schieben von 2 Bits nach links) und addiere (oder
subtrahiere) es zur
Adresse von Lrtx:
   add   x0, x1, w0, sxtb #2
; springe zur berechneten Adresse:
   br    x0
; dieses Label zeigt auf das Code (Text) Segment:
.Lrtx4:
   .section   .rodata
; alles nach dem ".section" Ausdruck wird im Read-only (rodata) Segment angelegt:
.L4:
   .byte   (.L3 - .Lrtx4) / 4 ; case 1
   .byte   (.L3 - .Lrtx4) / 4 ; case 2
   .byte   (.L5 - .Lrtx4) / 4 ; case 3
   .byte   (.L5 - .Lrtx4) / 4 ; case 4
   .byte   (.L5 - .Lrtx4) / 4 ; case 5
   .byte   (.L5 - .Lrtx4) / 4 ; case 6
   .byte   (.L3 - .Lrtx4) / 4 ; case 7
   .byte   (.L6 - .Lrtx4) / 4 ; case 8
   .byte   (.L6 - .Lrtx4) / 4 ; case 9
   .byte   (.L3 - .Lrtx4) / 4 ; case 10
   .byte   (.L2 - .Lrtx4) / 4 ; case 11
   .byte   (.L2 - .Lrtx4) / 4 ; case 12
   .byte   (.L2 - .Lrtx4) / 4 ; case 13
   .byte   (.L2 - .Lrtx4) / 4 ; case 14
```
alles nach dem ".text" Ausdruck wird im Code (Text) Segment angelegt:

.L7:
; "22" ausgeben
  adrp  x0, .LC3
  add  x0, x0, :lo12:.LC3
  b    puts

.L6:
; "8, 9, 21" ausgeben
  adrp  x0, .LC2
  add  x0, x0, :lo12:.LC2
  b    puts

.L5:
; "3, 4, 5" ausgeben
  adrp  x0, .LC1
  add  x0, x0, :lo12:.LC1
  b    puts

.L3:
; "1, 2, 7, 10" ausgeben
  adrp  x0, .LC0
  add  x0, x0, :lo12:.LC0
  b    puts

.LC0:
.string "1, 2, 7, 10"

.LC1:
.string "3, 4, 5"

.LC2:
.string "8, 9, 21"

.LC3:
.string "22"

.LC4:
.string "default"

Kompilieren wir dieses Beispiel in eine Object-Datei und öffnen es ist IDA. Hier ist die Jumptable:

Listing 1.135: jumptable in IDA

.rodata:000000000000000064 AREA .rodata, DATA, READONLY
.rodata:000000000000000064 ; ORG 0x64
.rodata:000000000000000064 $d DCB 9    ; case 1
.rodata:000000000000000065 DCB 9    ; case 2
.rodata:000000000000000066 DCB 6    ; case 3
.rodata:000000000000000067 DCB 6    ; case 4
.rodata:000000000000000068 DCB 6    ; case 5
.rodata:000000000000000069 DCB 6    ; case 6
.rodata:00000000000000006A DCB 9    ; case 7
.rodata:00000000000000006B DCB 3    ; case 8
.rodata:00000000000000006C DCB 3    ; case 9
.rodata:00000000000000006D DCB 9    ; case 10
.rodata:00000000000000006E DCB 0xF7  ; case 11
.rodata:00000000000000006F DCB 0xF7  ; case 12
.rodata:000000000000000070 DCB 0xF7  ; case 13
.rodata:000000000000000071 DCB 0xF7  ; case 14
.rodata:000000000000000072 DCB 0xF7  ; case 15
.rodata:000000000000000073 DCB 0xF7  ; case 16
.rodata:000000000000000074 DCB 0xF7  ; case 17
.rodata:000000000000000075 DCB 0xF7  ; case 18
.rodata:000000000000000076 DCB 0xF7  ; case 19
.rodata:000000000000000077 DCB 3     ; case 20
.rodata:000000000000000078 DCB 3     ; case 21
.rodata:000000000000000079 DCB 0     ; case 22
.rodata:00000000000000007B ; .rodata ends
Im Fall von 1, 9 wird also mit 4 multipliziert und zur Adresse des Labels Lrtx4 addiert.

Im Fall von 22, wird 0 mit 4 multipliziert; mit dem Ergebnis 0.

Direkt hinter dem Lrtx4 Label befindet sich das L7 Label, an dem sich der Code befindet, der „22“ ausgibt.

Es gibt keine Jumptable im Codesegment; sie wird in einem getrennten .rodata Segment angelegt (es besteht keine Notwendigkeit, die Tabelle im Codesegment anzulegen).

Hier befinden sich auch negative Bytes (0xF7), die für das Zurückspringen im Code verwendet werden, um den „Default“ String am Label .L2 auszugeben.

1.13.4 Fallthrough

Eine andere übliche Verwendung des switch() Operators ist der sogenannte „Fallthrough“. Hier ist ein einfaches Beispiel:

```c
bool is_whitespace(char c) {
    switch (c) {
    case ' ': // fallthrough
    case '\t': // fallthrough
    case '\r': // fallthrough
    case '\n':
        return true;
    default: // not whitespace
        return false;
    }
}
```

Ein etwas komplizierteres Beispiel aus dem Linux Kernel:

```c
char nco1, nco2;

void f(int if_freq_khz)
{
    switch (if_freq_khz) {
    default:
        printf("IF=%d KHz is not supported, 3250 assumed\n", if_freq_khz);
        /* fallthrough */
    case 3250: /* 3.25Mhz */
        nco1 = 0x34;
        nco2 = 0x00;
        break;
    case 3500: /* 3.50Mhz */
        nco1 = 0x38;
        nco2 = 0x00;
        break;
    case 4000: /* 4.00Mhz */
        nco1 = 0x40;
        nco2 = 0x00;
        break;
    case 5000: /* 5.00Mhz */
        nco1 = 0x50;
        nco2 = 0x00;
        break;
    case 5380: /* 5.38Mhz */
        nco1 = 0x56;
        nco2 = 0x14;
        break;
    }
}
```

Listing 1.136: Optimizing GCC 5.4.0 x86

.1C0:

.string "IF=%d KHz is not supported, 3250 assumed\n"

f:

93Kopiert von https://github.com/azonalon/prgraas/blob/master/progllib/lecture_examples/is_whitespace.c
94Kopiert von https://github.com/torvalds/linux/blob/master/drivers/media/dvb-frontends/lgdt3306a.c

Jetzt verstehen wir auch, warum der switch() Ausdruck manchmal eine Quelle von Bugs ist: ein einziges vergessenes break verändert einen switch() Ausdruck in einen Fallthrough und mehrere Blöcke anstelle einer einzigen werden ausgeführt.

1.13.5 Übungen

German text placeholder #1

Der C-Code des Beispiels in 1.13.2 on page 146 soll so neu geschrieben werden, dass der Compiler die gleiche Funktionalität in noch kürzerem Code erreichen kann.

1.14 Schleifen

1.14.1 Einfaches Beispiel

x86

Es gibt einen speziellen LOOP Befehl im x86 Befehlsatz, der den Wert des Registers ECX prüft und falls dieser ungleich 0 ist, dekrementiert und danach die den control flow wieder an das Label des LOOP

In C/C++ werden Schleifen normalerweise mittels for()--, while()- oder do/while()-Ausdrücken erzeugt. Starten wir mit for().

Dieser Ausdruck definiert eine Schleifeninitialisierung (setzt den Zähler auf einen Startwert), definiert eine Schleifenbedingung (ist der Zähler größer als ein Grenzwert?), legt fest, was in jedem Durchlauf (Inkrement/Dekrement) geschieht und umschließt einen Schleifenkörper.

for (initialization; condition; at each iteration)  
{  
  loop_body;  
}

Der erzeugte Code besteht ebenfalls aus vier Teilen.

Beginnen wir mit einem einfachen Beispiel:

```c
#include <stdio.h>

void printing_function(int i)  
{  
  printf ("f(%d)\n", i);  
};

int main()  
{  
  int i;  
  for (i=2; i<10; i++)  
  {  
    printing_function(i);  
  }  
  return 0;  
};
```

Ergebnis (MSVC 2010):

```c
_i$ = -4
_main PROC
  push ebp
  mov ebp, esp
  push ecx
  mov DWORD PTR _i$[ebp], 2 ; loop initialization
  jmp SHORT $LN3@main
$LN2@main:  
  mov eax, DWORD PTR _i$[ebp] ; here is what we do after each iteration:  
  add eax, 1 ; add 1 to (i) value
  mov DWORD PTR _i$[ebp], eax
$LN3@main:
  cmp DWORD PTR _i$[ebp], 10 ; this condition is checked before each iteration
  jge SHORT $LN1@main ; if (i) is biggest or equals to 10, lets finish loop
  mov ecx, DWORD PTR _i$[ebp] ; loop body: call printing_function(i)
  push ecx
  call _printing_function
  add esp, 4
  jmp SHORT $LN2@main ; jump to loop begin
$LN1@main: ; loop end
  xor eax, eax
  mov esp, ebp
  pop ebp
  ret 0
_main ENDP
```

Hier gibt es nichts Besonderes zu sehen.

GCC 4.4.1 erzeugt einen fast identischen Code mit nur einen kleinen Unterschied:
Schauen wir uns nun an, was wir erhalten, wenn wir die Optimierung aktivieren (/Ox):

Was hier passiert ist, dass der Speicherplatz für die \( i \) Variable nicht mehr auf dem lokalen Stack bereitgestellt wird, sondern das extra ein Register, ESI, hierfür verwendet wird. Dies ist bei derartig kleinen Funktionen möglich, wenn nicht zu viele lokalen Variablen existieren.

Wichtig ist, dass die \( f() \) Funktion den Wert im Register ESI nicht verändern darf. Unser Compiler ist sich dieser Sache hier sicher. Und falls der Compiler entscheidet, das ESI auch innerhalb der Funktion \( f() \) zu verwenden, würde der Wert des Registers im Funktionsprolog gesichert und im Funktionsepilog wiederhergestellt werden; fast genauso wie im folgenden Listing. Man beachte das PUSH ESI/POP ESI bei Funktionsbeginn und -ende.

Probieren wir aus, was GCC 4.4.1 mit maximaler Optimierung (-O3 option) liefert:
Aha, GCC hat unsere Schleife unrolled (d.h. ausgerollt).

GERMAN TEXT PLACEHOLDER hat Vorteile in Fällen, in denen es nicht viele Schleifendurchläufe gibt und Ausführungszeit durch das Weglassen der Befehle für die Kontrollstrukturen der Schleife gewonnen werden kann. Andererseits ist der erzeugte Code natürlich deutlich länger.

Große Schleifen zu unrollen ist heutzutage nicht empfehlenswert, denn größere Funktionen erfordern einen größeren Cache-Fußabdruck.95.

Gut, nun wollen wir den Höchstwert der Variable $i$ auf 100 setzen und kompilieren erneut. GCC liefert:

```
Listing 1.141: GCC
```

Das Ergebnis ist sehr ähnlich dem, das MSVC 2010 mit Optimierung (/Ox) erzeugt, mit der Ausnahme, dass das EBX Register für die Variable $i$ verwendet wird.

GCC ist sicher, dass das Register innerhalb der f() Funktion nicht verändert wird und sollte dies doch der Fall sein, dass es im Funktionsprolog gesichert und im Funktionsepiolog wiederhergestellt werden wird, genau wie hier in der main() Funktion.

Wir kompilieren unser Beispiel in MSVC 2010 mit den Optionen /Ox und /Ob0 und laden es in OllyDbg. Es scheint, dass OllyDbg in der Lage ist, einfache Schleifen zu erkennen und in eckigen Klammern darzustellen, um die Übersichtlichkeit zu erhöhen:

Abbildung 1.55: OllyDbg: main() Einstieg

Verfolgen mit (F8 — German text placeholder) zeigt ESI incrementing. Hier ist zum Beispiel, $ESI = i = 6$:

Abbildung 1.56: OllyDbg: Schleifenkörper wird gerade ausgeführt für $i = 6$

9 ist der letzte Wert in der Schleife Deshalb triggert JL nach der increment-Anweisung nicht und die Funktion wird beendet:
Abbildung 1.57: OllyDbg: $ESI = 10$, Ende der Schleife

x86: tracer


Wir öffnen das kompilierte Beispiel in IDA, finden die Adresse mit dem Befehl `PUSH ESI` (das einzige Argument an `f()` übergebend), welche hier 0x401026 ist und aktivieren den Tracer:

```
tracer.exe -l:loops_2.exe bp=loops_2.exe!0x00401026
```

BPX setzt einen Breakpoint an der Adresse und der Tracer zeigt uns den momentanen Status der Register an. In `tracer.log` sehen wir das Folgende:

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x000a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x0000000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x000000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x000000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x000000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x000000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x000000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x000000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
```
Wir sehen wie der Wert des ESI Registers sich schrittweise von 2 zu 9 verändert.


Danach die ein IDA.idc-script erzeugt, das Kommentare hinzufügt. Wir haben also herausgefunden, dass die Adresse der main() Funktion 0x00401020 ist und wir führen nun das Folgende aus:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF setzt einen Breakpoint auf eine Funktion.

Als Ergebnis erahnten wir die Skripte loops_2.exe.idc und loops_2.exe_clear.idc.
Wir laden `loops_2.exe.idc` in IDA und erhalten:

Abbildung 1.58: IDA mit geladenem .idc-script

Wir sehen, dass der Wert von ESI zu Beginn der Schleife zwischen 2 und 9 und nach dem Inkrement zwischen 3 und 0xA (10) liegt. Wir sehen auch, dass die Funktion `main()` mit dem Rückgabewert 0 in EAX terminiert.

German text placeholder erzeugt ebenfalls die Datei `loops_2.exe.txt`, welche Informationen darüber enthält, welcher Befehl wie oft ausgeführt wurde, sowie zugehörige Registerwerte:

Listing 1.142: loops_2.exe.txt

<table>
<thead>
<tr>
<th>Adresse</th>
<th>E</th>
<th>Befehl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401020</td>
<td>e=1</td>
<td>[PUSH ESI] ESI=1</td>
</tr>
<tr>
<td>0x401021</td>
<td>e=1</td>
<td>[MOV ESI, 2]</td>
</tr>
<tr>
<td>0x401026</td>
<td>e=8</td>
<td>[PUSH ESI] ESI=2..9</td>
</tr>
<tr>
<td>0x401027</td>
<td>e=8</td>
<td>[CALL 8D1000h]</td>
</tr>
<tr>
<td>0x401028</td>
<td>e=8</td>
<td>[INC ESI]</td>
</tr>
<tr>
<td>0x401029</td>
<td>e=8</td>
<td>[ADD ESP, 4]</td>
</tr>
<tr>
<td>0x40102C</td>
<td>e=8</td>
<td>[CMP ESI, 0Ah]</td>
</tr>
<tr>
<td>0x401033</td>
<td>e=8</td>
<td>[JL short loc_401026]</td>
</tr>
<tr>
<td>0x401035</td>
<td>e=8</td>
<td>[XOR EAX, EAX]</td>
</tr>
<tr>
<td>0x401037</td>
<td>e=1</td>
<td>[POP ESI]</td>
</tr>
<tr>
<td>0x401038</td>
<td>e=1</td>
<td>[RETN] EAX=0</td>
</tr>
</tbody>
</table>

An dieser Stelle können wir grep verwenden.

**ARM**

German text placeholder Keil 6/2013 (German text placeholder)
Der Zähler \( i \) wird im Register \( R4 \) gespeichert. Der Befehl MOV \( R4, \#2 \) initialisiert \( i \). Die Befehle MOV \( R0, R4 \) und BL \( \text{printing_function} \) bilden den Körper der Schleife; der erste Befehl bereitet das Argument für die \( f() \) Funktion vor und der zweite ruft die Funktion auf.

Der Befehl ADD \( R4, R4, \#1 \) erhöht die \( i \) Variable in jedem Durchlauf um 1. CMP \( R4, \#0xA \) vergleicht \( i \) mit 0xA (10). Der nächste Befehl BLT (\text{Branch Less Than}) springt, falls \( i \) kleiner als 10 ist. Sonst wird 0 in das Register \( R0 \) geschrieben (unsere Funktion liefert den Wert 0 zurück) und die Funktionsausführung wird beendet.

**German text placeholder Keil 6/2013 (German text placeholder)**

```
_main
    PUSH    {R4,LR}
    MOV     R4, #2

loc_132
    MOV     R0, R4
    BL      \text{printing_function}
    ADDS    R4, R4, #1
    CMP     R4, #0xA
    BLT     loc_132
    MOV     R0, #0
    POP     {R4,PC}
```

Praktisch das gleiche.

**German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**

```
_main
    PUSH    {R4,R7,LR}
    MOVW    R4, #0x1124 ; "%d\n"
    MOV     R1, #2
    MOVT.W  R4, #0
    ADD     R7, SP, #4
    ADD     R4, PC
    MOV     R0, R4
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #3
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #4
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #5
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #6
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #7
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #8
    BLX     _printf
    MOV     R0, R4
    MOVS    R1, #9
    BLX     _printf
    MOV     R0, #0
    POP     {R4,R7,PC}
```
In meiner `f()` Funktion befand sich tatsächlich Folgendes:

```c
void printing_function(int i)
{
    printf("%d\n", i);
}
```

Also hat LLVM die Schleife nicht nur unrolled sondern auch die einfache Funktion `f()` inlined und den Körper der Schleife acht Mal generiert, anstatt die Schleife aufzurufen.

Dies ist möglich, wenn eine Funktion sehr einfach ist (wie meine) und wenn sie nicht allzu oft aufgerufen wird (wie hier).

**ARM64: German text placeholder GCC 4.9.1**

Listing 1.143: German text placeholder GCC 4.9.1

```assembly
printing_function:
; zweites Argument von printf() vorbereiten:
    mov    w1, w0
; lade Adresse des "f(%d)\n" Strings
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
; verzweige nur hier, anstatt Verzweigungen zu link und return:
    b      printf
main:
; speichern von FP und LR auf dem lokalen Stack:
    stp    x29, x30, [sp, -32]!
; setze Stack Frame:
    add    x29, sp, 0
; Inhalt des X19 Registers auf dem Stack ablegen
    str    x19, [sp,16]
; wir verwenden das W19 Register als Zähler.
; weise diesem den Startwert 2 zu:
    mov    w19, 2
.L3:
; erstes Argument von printing_function() vorbereiten:
    mov    w0, w19
; Zählerregister inkrementieren.
    add    w19, w19, 1
; W0 enthält hier immer noch den Wert des Zählers vor dem Inkrement.
    bl     printing_function
; am Ende?
    cmp    w19, 10
; nein, dann springe zum Anfang des Schleifenkörpers:
    bne    .L3
; return 0
    mov    w0, 0
; wiederherstellen des X19 Registers:
    ldr    x19, [sp,16]
; wiederherstellen von FP und LR:
    ldp    x29, x30, [sp], 32
ret .LC0:
    .string "f(%d)\n"
```

**ARM64: German text placeholder GCC 4.9.1**

Listing 1.144: German text placeholder GCC 4.9.1 -fno-inline

```assembly
.LC0:
    .string "f(%d)\n"
printing_function:
; speichere FP und LR auf dem lokalen Stack:
    stp    x29, x30, [sp, -32]!
```

172
MIPS

Eine Sache noch

Im generierten Code sehen wir folgendes: nach der Initialisierung von \( i \) wird der Körper der Schleife nicht ausgeführt, da die Bedingung für \( i \) zuerst geprüft wird und erst danach der Körper der Schleife ausgeführt werden kann. Und so ist es auch korrekt.

Denn, falls die Bedingung zu Beginn falsch ist, darf der Körper der Schleife nie ausgeführt werden. Dies ist z.B. im folgenden Fall möglich:

```mips
for (i=0; i<total_entries_to_process; i++)
    Schleifenkörper;
```

Wenn \textit{total\_entries\_to\_process} gleich 0 ist, darf der Körper der Schleife auf keinen Fall ausgeführt werden. Deshalb wird die Bedingung stets vor der Ausführung geprüft.

Ein optimierter Compiler kann jedoch das Prüfen der Bedingung und den Schleifenkörper vertauschen,
falls sichergestellt ist, dass die hier beschriebene Situation auf keinen Fall eintreten kann (wie in unserem sehr einfachen Beispiel und bei Keil, Xcode (LLVM), MSVC im optimierten Modus).

1.14.2 Funktion zum Kopieren von Speicherblöcken

Echte Funktionen zum Kopieren von Speicherblöcken kopieren in jedem Schritt 4 oder 8 Byte und verwenden SIMD\textsuperscript{97}, Vektorisierung, etc. Aber um einen Eindruck zu erhalten betrachte dieses einfachstmögliche Beispiel.

```c
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt) {
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
}
```

Grundlegende Implementierung

Listing 1.145: GCC 4.9 x64 optimized for size (-Os)

```
my_memcpy:
    ; RDI = Zieladresse
    ; RSI = Quelladresse
    ; RDX = Blockgröße
    ; initialisiere Zähler (i) mit 0
    xor   eax, eax
.L2:
    ; alles kopiert? dann verlassen:
    cmp    rax, rdx
    je .L5
    ; load byte at RSI+i:
    mov    cl, BYTE PTR [rsi+rax]
    ; store byte at RDI+i:
    mov    BYTE PTR [rdi+rax], cl
    inc    rax ; i++
    jmp    .L2
.L5:
    ret
```

Listing 1.146: GCC 4.9 ARM64 optimized for size (-Os)

```
my_memcpy:
    ; X0 = Zieladresse
    ; X1 = Quelladresse
    ; X2 = Blockgröße
    ; initialisiere Zähler (i) mit 0
    mov    x3, 0
.L2:
    ; alles kopiert? dann verlassen:
    cmp    x3, x2
    beq    .L5
    ; load byte at X1+i:
    ldrb   w4, [x1,x3]
    ; store byte at X0+i:
    strb   w4, [x0,x3]
    add    x3, x3, 1 ; i++
    b .L2
.L5:
    ret
```

Listing 1.147: German text placeholder Keil 6/2013 (German text placeholder)

\textsuperscript{97}Single Instruction, Multiple Data
ARM in ARM mode

Keil in ARM mode macht Gebrauch von konditionalen Suffixen:

my_memcpy PROC
        ; R0 = Zieladresse
        ; R1 = Quelladresse
        ; R2 = Blockgröße

        ; initialisiere Zähler (i) mit 0
        MOV r3,#0
        ; Bedingung wird am Ende der Schleife geprüft, daher springe dorthin:
        B [L0.12]

        |L0.6|
        ; lade Byte von R1+i:
        LDRB r4,[r1,r3]
        ; speichere Byte an der Stelle R0+i:
        STRB r4,[r0,r3]
        ; i++
        ADDS r3,r3,#1

        |L0.12|
        ; i<size?
        CMP r3,r2
        ; springe zum Anfang der Schleife, falls es so ist:
        BCC [L0.6]
        POP {r4,pc}
        ENDP

Deshalb gibt es hier nur einen Verzweigungsbefehl anstatt deren zwei.

MIPS

my_memcpy:

        ; springe zum Prüfen der Schleifenbedingung:
        b loc_14

        ; initialisiere Zähler (i) mit 0
        ; bleibt stets in $v0:
        move $v0, $zero ; branch delay slot
Hier tauchen zwei neue Befehle auf: LBU („Load Byte Unsigned“) und SB („Store Byte“).

Wie in ARM haben alle MIPS Register eine Breite von 32 bit, es gibt keine byte-breiten Teile wie bei x86. Wenn wir also mit einzelnen Bytes arbeiten, müssen wir stets ein komplettes 32-bit-Register hierfür verwenden. LBU lädt ein Byte und löscht alle anderen Bits („Unsigned“).

Der Befehl LB („Load Byte“) dagegen erweitert das geladene Byte zu einem vorzeichenbehafteten 32-bit-Wert.

SB schreibt ein Byte der niederwertigsten 8 Bit des Registers in den Speicher.

**Vektorisierung**

**German text placeholder** GCC kann viel mehr; siehe dieses Beispiel: 1.24.1 on page 374.

### 1.14.3 Fazit

Gerüst einer Schleife von einschließlich 2 bis einschließlich 9:

**Listing 1.150: x86**

```
    mov [counter], 2 ; Initialisierung
    jmp check

body:
    ; Schleifenkörper
    ; tue etwas
    ; verwende Zählervariable auf lokalem Stack
    add [counter], 1 ; inkrementieren

check:
    cmp [counter], 9
    jle body
```

Das Inkrementieren kann in nicht optimiertem Code durch 3 Instruktionen dargestellt werden:

**Listing 1.151: x86**

```
    MOV [counter], 2 ; Initialisierung
    JMP check

body:
    ; Schleifenkörper
    ; tue etwas
    ; verwende Zählervariable auf lokalem Stack
    MOV REG, [counter] ; inkrementieren
    INC REG
    MOV [counter], REG

check:
```
Falls der Körper einer Schleife besonders kurz ist, kann ein Register als Zähler verwendet werden:

```
Listing 1.152: x86

MOV EBX, 2 ; Initialisierung
JMP check
body:
; Schleifenkörper
; tue etwas
; verwende Zähler in EBX, aber verändere ihn nicht!
INC EBX ; inkrementieren
check:
CMP EBX, 9
JLE body
```

Einige Teile der Schleife können vom Compiler in unterschiedlichen Reihenfolgen generiert werden:

```
Listing 1.153: x86

MOV [counter], 2 ; Initialisierung
JMP label_check
label_increment:
ADD [counter], 1 ; inkrementieren
label_check:
CMP [counter], 10
JGE exit
; Schleifenkörper
; tue etwas
; verwende Zählervariable auf lokalem Stack
JMP label_increment
exit:
```

Normalerweise wird die Bedingung vor dem Körper geprüft, aber der Compiler kann den Code auch so anordnen, dass die Bedingung nach dem Körper geprüft wird.

Dies geschieht dann, wenn der Compiler sicher sein kann, dass die Bedingung im ersten Durchlauf stets wahr ist, sodass der Körper der Schleife mindestens einmal tatsächlich ausgeführt wird:

```
Listing 1.154: x86

MOV REG, 2 ; Initialisierung
body:
; Schleifenkörper
; tue etwas
; verwende Zähler in REG, aber verändere ihn nicht!
INC REG ; inkrementieren
CMP REG, 10
JL body
```

Verwendung des LOOP Befehls. Sehr selten, Compiler verwenden ihn nicht. Wenn er auftaucht, ist dies ein Zeichen dafür, dass das entsprechende Codesegment von Hand geschrieben worden ist:

```
Listing 1.155: x86

; zähle von 10 bis 1
MOV ECX, 10
body:
; Schleifenkörper
; tue etwas
; verwende Zähler in ECX, aber verändere ihn nicht!
LOOP body
```

ARM.

Das R4 Register fungiert in diesem Beispiel als Zähler:
Listing 1.156: ARM

MOV R4, 2 ; Initialisierung
B check
body:
; Schleifenkörper
; tue etwas
; verwende Zähler in R4, aber verändere ihn nicht!
ADD R4,R4, #1 ; inkrementieren
check:
CMP R4, #10
BLT body

1.14.4 Übungen

- http://challenges.re/54
- http://challenges.re/55
- http://challenges.re/56
- http://challenges.re/57

1.15 Mehr über Zeichenketten

1.15.1 strlen()

German text placeholder

```c
int my_strlen (const char * str)
{
    const char *eos = str;
    while( *eos++ ) ;
    return( eos - str - 1 );
}
```

```c
int main()
{
    // test
    return my_strlen("hello");
};
```

x86

German text placeholder MSVC

Kompilieren wir es:

```assembly
_eos$ = -4 ; size = 4
_str$ = 8 ; size = 4
_strlen PROC
    push ebp
    mov ebp, esp
    push ecx
    mov eax, DWORD PTR _str$[ebp] ; setze Pointer auf String von "str"
    mov DWORD PTR _eos$[ebp], eax ; setze ihn auf lokale Variable "eos"
$LN2@strlen_
    mov ecx, DWORD PTR _eos$[ebp] ; ECX=eos
    ; nimm ein Byte von der Adresse in ECX und speichere es als 32-bit Wert mit
    ; Vorzeichen in EDX
    movsx edx, BYTE PTR [ecx]
    mov eax, DWORD PTR _eos$[ebp] ; EAX=eos
    add eax, 1 ; erhöhe EAX
```
Wir finden hier zwei neue Befehle: MOV SX und TEST.

Der erste MOV SX nimmt ein Byte aus einer Speicheradresse und speichert den Wert in einem 32-bit-Register. MOV SX steht für MOV with Sign-Extend. MOV SX setzt die übrigen Bits vom 8. bis zum 31. auf 1, falls das Quellbyte negativ ist oder auf 0, falls es positiv ist.

Und hier ist der Grund dafür.

Standardmäßig ist der char Datentyp in MSVC und GCC vorzeichenbehaftet (signed). Wenn wir zwei Werte haben, einen char und einen int, (int ist ebenfalls vorzeichenbehaftet) und der erste Wert enthält -2 (kodiert als 0xFE) und wir kopieren dieses Byte in den int Container, erhalten wir 0x000000FE und dies entspricht als signed int 254, aber nicht -2. Der signed int-2 wird als 0xFFFFFFFFF dargestellt. Wenn wir also 0xFE vom Datentyp char nach int übertragen wollen, müssen wir das Vorzeichen identifizieren und den Wert entsprechend erweitern. Genau dies tut der Befehl MOV SX.

Weitere Informationen dazu finden sich im Abschnitt „German text placeholder“ (2.1 on page 408).

Es ist schwer zu sagen, ob der Compiler tatsächlich eine char Variable in EDX speichern muss, er könnte auch einen 8-Bit-Registerteil (z.B. DL) dafür verwenden. Offenbar arbeitet der German text placeholder des Compilers auf diese Art.

Wir finden im Weiteren den Befehl TEST EDX, EDX. Für mehr Informationen zum TEST Befehl siehe auch den Abschnitt über Bitfelder (1.19 on page 279). In unserem Fall überprüft der Befehl lediglich, ob der Wert im Register EDX gleich 0 ist.

German text placeholder GCC

Schauen wir uns GCC 4.4.1 an:

```assembler
public strlen
strlen proc near
eos = dword ptr -4
arg_0 = dword ptr 8

push ebp
mov ebp, esp
sub esp, 10h
mov eax, [ebp+arg_0]
mov [ebp+eos], eax
loc_80483F0:
    mov eax, [ebp+eos]
    movzx eax, byte ptr [eax]
    test al, al
    setnz al
    add [ebp+eos], 1
    test al, al
    jnz short loc_80483F0
    mov edx, [ebp+eos]
    mov eax, [ebp+arg_0]
    mov ecx, edx
    sub ecx, eax
    mov eax, ecx
    sub eax, 1
 גוגל
```
Das Ergebnis ist fast identisch mit dem von MSVC, aber hier finden wir MOVZX anstelle von MOVSX. MOVZX steht für MOV with Zero-Extend. Dieser Befehl kopiert einen 8-Bit- oder 16-Bit-Wert in ein 32-Bit-Register und setzt die übrigen Bits auf 0. Tatsächlich findet dieser Befehl vor allem deshalb Anwendung, weil er es uns erlaubt, folgendes Befehlspaar zu ersetzen:

`xor eax, eax / mov al, [...].`

Andererseits ist offensichtlich, dass der Compiler folgenden Code erzeugen kann:


Der nächste neue Befehl für uns ist SETNZ. In diesem Fall setzt `test al, al` das ZF flag auf 0, falls AL nicht 0 enthält, aber SETNZ setzt AL auf 1, falls ZF==0 (ITNZ steht für non zero). In natürlicher Sprache, falls AL ungleich 0, springe zu loc_80483F0. Der Compiler erzeugt leicht redundanten Code, aber bedenken wir, dass die Optimierung hier deaktiviert ist.

**German text placeholder** **MSVC**

Kompilieren wir nun alles in MSVC 2012 mit aktivierter Optimierung (/Ox):

Listing 1.157: German text placeholder MSVC 2012 /Ob0

```assembly
_str$ = 8 ; size = 4
_strlen PROC
    mov edx, DWORD PTR _str$[esp-4] ; EDX -> Pointer auf den String
    mov eax, edx ; verschiebe nach EAX
$LL2@strlen:
    mov cl, BYTE PTR [eax] ; CL = *EAX
    inc eax ; EAX++
    test cl, cl ; CL==0?
    jne SHORT $LL2@strlen ; nein, setze Schleife fort
    sub eax, edx ; berechne Differenz der Pointer
    dec eax ; dekrementiere EAX
    ret 0
_strlen ENDP
```

Jetzt ist alles einfacher. Unnötig zu erwähnen, dass der Compiler Register mit solcher Effizienz nur in kleinen Funktionen mit einigen wenigen lokalen Variablen verwenden kann.

INC/DEC—sind inkrement/dekrement Befehle; mit anderen Worten: addiere oder subtrahiere 1 zu bzw. von einer Variable.
Wir untersuchen das (optimierte) Beispiel in OllyDbg. Hier ist der erste Durchlauf:


Wenn OllyDbg ein Register mit einer gültigen Adresse, die auf einen String zeigt, findet, wird dieser String angezeigt.

Abbildung 1.59: OllyDbg: Beginn erster Durchlauf
Wir drücken einige Male F8 (German text placeholder) um zum Anfang der Schleifenkörpers zu gelangen:

Abbildung 1.60: OllyDbg: Beginn zweiter Durchlaufen

Wir sehen, dass EAX nun die Adresse des zweiten Zeichens des Strings enthält.
Durch hinreichend häufiges Drücken von F8 verlassen wir schließlich die Schleife:

Wir sehen, dass EAX jetzt die Adresse des Nullbytes direkt hinter dem String enthält. In der Zwischenzeit hat sich EDX nicht verändert, es zeigt also immer noch auf den Anfang des Strings.

Die Differenz zwischen den beiden Adressen wird jetzt berechnet.

Abbildung 1.61: OllyDbg: Pointer Differenz wird berechnet

Wir sehen, dass EAX jetzt die Adresse des Nullbytes direkt hinter dem String enthält. In der Zwischenzeit hat sich EDX nicht verändert, es zeigt also immer noch auf den Anfang des Strings.

Die Differenz zwischen den beiden Adressen wird jetzt berechnet.
Der SUB Befehl wurde gerade ausgeführt:

Abbildung 1.62: OllyDbg: EAX muss dekrementiert werden

Die Differenz der Pointer im EAX Register beträgt nun –7. Tatsächlich beträgt die Länge des „hello!” Strings 6 Zeichen, aber mit dem Nullbyte am Ende dazugezählt sind es 7. Die Funktion strlen() soll aber die Anzahl der Nicht-Null-Zeichen im String zurückliefert, also wird einmal dekrementiert und der Funktionsaufruf anschließend beendet.

**German text placeholder GCC**

Schauen wir uns GCC 4.4.1 mit aktiver Optimierung (-O3 key) an:

```assembly
public strlen
proc near
arg_0 = dword ptr 8
push ebp
mov ebp, esp
mov ecx, [ebp+arg_0]
mov eax, ecx
loc_8048418:
movzx edx, byte ptr [eax]
add eax, 1
test dl, dl
jnz short loc_8048418
not ecx
add eax, ecx
pop ebp
retn
strlen endp
```

Hier erzeugt GCC fast identischen Code zu MSVC, außer dass hier ein MOVZX auftritt. In der Tat könnte MOVZX hier durch mov dl, byte ptr [eax] ersetzt werden.

Möglicherweise ist es einfacher für den GCC Code Generator sich daran zu erinnern, dass das gesamte 32-bit-EDX Register für eine char Variable reserviert ist und so sicherzustellen, dass die oberen Bits zu
keinem Zeitpunkt Zufallsrauschen enthalten.

Danach finden wir also einen neuen Befehl NOT. Dieser Befehl kippt alle Bits in seinem Operanden. Man kann sagen, dass es sich um ein Synonym zum Befehl XOR ECX, 0xfffffffh handelt. NOT und das darauf folgende ADD berechnen die Differenz im Pointer und subtrahieren 1, nur auf eine andere Art und Weise. Zu Beginn wird ECX, in dem der Pointer auf str gespeichert ist, invertiert und vom Ergebnis wird 1 abgezogen.

Hierzu siehe auch: „German text placeholder“ (2.1 on page 408).

Mit anderen Worten, am Ende der Funktion, direkt nach dem Schleifenkörper, werden die folgenden Befehle ausgeführt:

```c
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

... und das ist äquivalent zu:

```c
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Warum GCC entschieden hat, dass das eine besser ist als das andere? Schwierig zu sagen. Möglicherweise sind aber beide Varianten gleichermaßen effizient.

**ARM**

**32-bit ARM**

**German text placeholder Xcode 4.6.3 (LLVM)**

Listing 1.158: German text placeholder Xcode 4.6.3 (LLVM)

```c
_strlen
eos = -8
str = -4

SUB SP, SP, #8 ; reserviere 8 Bytes für lokale Variablen
STR R0, [SP,#8+str]
LDR R0, [SP,#8+str]
STR R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
LDR R0, [SP,#8+eos]
ADD R1, R0, #1
STR R1, [SP,#8+eos]
LDRSB R0, [R0]
CMP R0, #0
BEQ loc_2CD4
B loc_2CB8

loc_2CD4 ; CODE XREF: strlen+24
LDR R0, [SP,#8+eos]
LDR R1, [SP,#8+str]
SUB R0, R0, R1 ; R0=eos-str
SUB R0, R0, #1 ; R0=R0-1
ADD SP, SP, #8 ; setze 8 Bytes an Speicher frei
BX LR
```
Der nicht optimierende LLVM erzeugt zu viel Code, aber wir können wir erkennen wir die Funktion mit lokalen Variablen auf dem Stack arbeitet. Es gibt nur zwei lokale Variablen in unserer Funktion: eos und str. In folgenden von IDA erzeugten Listing, sind die Variablen var_8 und var_4 in eos bzw. str umbenannt.

Der erste Befehl speichert lediglich bei Eingabewerte in str und eos.

Der Körper der Schleife startet beim Label loc_2CB8.

Die ersten drei Befehle des Schleifenkörpers (LDR, ADD, STR) laden den Wert von eos nach R0. Anschließend wird der Wert erhöht und zurück in eos auf den Stack geschrieben.

Der folgende Befehl, LDRSB R0, [R0] („Load Register Signed Byte“), lädt ein Byte aus dem Speicher von der Adresse in R0 und erweitert es mit Vorzeichen auf 32-bit. 98 Dies ist vergleichbar zum MOVVSX Befehl in x86.

Der Compiler behandelt dieses Byte als signed, das der char Typ nach dem C-Standard ebenfalls signed ist. Dies wurde in Bezug auf x86 in diesem Abschnitt bereits in (1.15.1 on page 179) beschrieben.

Man beachte, dass es in ARM unmöglich ist, einen 8- oder 16-bit-Teil eines 32-bit-Registers alleine zu verwenden, anderes als in x86.

Dies rührt daher, dass x86 eine große Bandbreite und Kompatibilität mit Vorgängerversionen besitzt, bis hin zum 16-bit 8086 oder sogar dem 8-bit 8080, ARM auf der anderen Seite jedoch von Beginn an als 32-bit RISC-Prozessor geplant wurde.

Infolgedessen müssen auch um einzelne Bytes in ARM zu verarbeiten, stets komplette 32-bit-Register verwendet werden.

Der Befehl LDRSB lädt nun die Bytes des String einzeln nach R0. Die nachfolgenden CMP und BEQ Befehle prüfen, ob das aktuelle Byte 0 ist. Wenn nicht, beginnt der Körper der Schleife erneut. Und wenn das aktuelle Byte 0 ist, dann wird die Schleife beendet.

Am Ende der Funktion wird die Differenz zwischen eos und str berechnet, 1 vom Ergebnis abgezogen und das Resultat über das Register R0 zurückgegeben.

N.B. Register wurden in dieser Funktion nicht gespeichert. Das liegt daran, dass gemäß der ARM Aufrufkonventionen die Register R0 bis R3 sogenannte „scratch register“ sind, vorgesehen für Parameterübergaben. Deshalb ist es nicht notwendig ihren Inhalt am Ende der Funktion wiederzustellen, denn die aufrufende Funktion wird diese Werte nicht weiter verwenden. Im weiteren können diese für alles Mögliche benutzt werden.

Es werden hier keine weiteren Register verwendet, sodass wir nichts auf dem Stack speichern müssen.

Damit kann der control flow über einen einfachen Sprung (BX) an die aufrufende Funktion an der Adresse im LR Register übergeben werden.

German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)

Listing 1.159: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)

<table>
<thead>
<tr>
<th>strlen</th>
<th>MOV</th>
<th>R1, R0</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc_2DF6</td>
<td>LDRB.W</td>
<td>R2, [R1],#1</td>
</tr>
<tr>
<td></td>
<td>CMP</td>
<td>R2, #0</td>
</tr>
<tr>
<td></td>
<td>BNE</td>
<td>loc_2DF6</td>
</tr>
<tr>
<td></td>
<td>MVNS</td>
<td>R0, R0</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>R0, R1</td>
</tr>
<tr>
<td></td>
<td>BX</td>
<td>LR</td>
</tr>
</tbody>
</table>

Der optimierende LLVM entscheidet also, dass eos und str keinen Platz auf dem Stack benötigen, sondern stets in Registern gespeichert werden können.

Vor dem Anfang des Schleifenkörpers befindet sich str stets in R0 und eos in R1.

Der Befehl LDRB.W R2, [R1],#1 lädt ein Byte aus dem Speicher von der Adresse aus R1 nach R2, erweitert es zum einem signed 32-bit-Wert und mehr noch: Das #1 am Ende des Befehl bewirkt „Post-indexed addressing“, was bedeutet, dass 1 zum Register R1 addiert wird, nachdem das Byte geladen wurde. Mehr zum Thema: 1.27.2 on page 401.

98Der Keil Compiler behandelt den Typ char als signed, genau wie MSVC und GCC.
Des Weiteren finden wir CMP und BNE\textsuperscript{99} im Körper der Schleife; diese Befehle werden durchlaufen, bis 0 im String gefunden wurde.

MVNS\textsuperscript{100} (invertiert alle Bits wie NOT in x86) und ADD Befehle berechnen $cos - str - 1$. Tatsächlich berechnen diese beiden Befehle $R0 = str + cos$, was äquivalent zur Formulierung im Quellcode ist und die Begründung dazu wurde bereits hier gegeben (1.15.1 on page 185).

Offenbar befindet LLVM genau wie GCC, dass diese Code kürzer (oder schneller) ist.

**German text placeholder Keil 6/2013 (German text placeholder)**

### Listing 1.160: German text placeholder Keil 6/2013 (German text placeholder)

```assembly
_strlen
    MOV   R1, R0

loc_2C8
    LDRB  R2, [R1],#1
    CMP   R2, #0
    SUBEQ R0, R1, R0
    SUBEQ R0, R0, #1
    BNE   loc_2C8
    BX    LR
```

Fast das gleiche wie zuvor, mit der Änderung, dass der $str - cos - 1$ Ausdruck nicht am Ende der Funktion, sondern mitten in der Schleife berechnet wird. Wir erinnern uns, dass der -EQ Suffix bedeutet, dass die dieser Befehl nur dann ausgeführt wird, wenn die Operanden im CMP direkt davor gleich waren. Dadurch werden beide SUBEQ Befehle ausgeführt, falls das R0 Register 0 enthält und das Ergebnis verbleibt in R0.

**ARM64**

**German text placeholder GCC (Linaro) 4.9**

```assembly
my_strlen:
    mov   x1, x0
    ; X1 ist der temporäre Pointer (eos), verhält sich wie ein Cursor

.L58:
    ldrb  w2, [x1],1
    ; Vergleich und Verzweigung, falls nicht null: vergleiche W2 mit 0, springe
    cbnz  w2, .L58
    ; berechne Differenz zwischen ursprünglichem Pointer in X0 und aktueller
    sub   x0, x1, x0
    ; dekrementiere niedere 32-bit des Ergebnisses
    sub   w0, w0, #1
    ret
```

Der Algorithmus ist der gleiche wie in 1.15.1 on page 180: finde ein Nullbyte, berechne die Differenz zwischen den Pointern und subtrahiere 1 vom Ergebnis. Einige Kommentare wurden vom Autor hinzugefügt.

Die einzig bemerkenswerte Sache ist, dass unser Beispiel in gewisser Weise fehlerhaft ist: `my_strlen()` liefert einen 32-bit int, obwohl es size_t oder einen anderen 64-bit Typ zurückliefern müsste. Der Grund dafür ist, dass `strlen()` theoretisch für einen sehr großen Speicherblock, größer als 4GB, aufgerufen werden könnte und deshalb auf einer 64-bit-Plattform in der Lage sein muss, einen 64-bit-Wert zurückzuliefern.

Aufgrund meines Fehlers, arbeitet der letzte SUB Befehl nur mit einem 32-bit-Teil des Registers, wohingegen der vorletzte SUB Befehl mit dem kompletten 64-bit-Register arbeitet (und die Differenz zwischen den Pointer berechnet).

\textsuperscript{99}(PowerPC, ARM) Branch if Not Equal
\textsuperscript{100}MoVe Not
Es handelt sich um einen Fehler von mir, und es ist besser es so zu lassen, als ein Lehrbeispiel wie Code in einem derartigen Fall aussehen kann.

**German text placeholder GCC (Linaro) 4.9**

```assembly
my_strlen:
	; Funktionsprolog
	sub  sp, sp, #32
	; erstes Argument (str) wird in [sp,8] gespeichert
	str  x0, [sp,8]
	ldr  x0, [sp,8]
	; kopiere "str" in die "eos" Variable
	str  x0, [sp,24]
	nop

.lbl62:
	; eos++
	ldr x0, [sp,24] ; lade "eos" nach X0
	add x1, x0, 1 ; erhöhe X0
	str x1, [sp,24] ; sichere X0 in "eos"
	; lade Byte aus dem Speicher von Adresse in X0 nach W0
	ldrb w0, [x0] ; null gefunden? (das 32-bit Register WZR enthält stets 0)
	cmp w0, wZR ; springe, falls nicht null (Zweig ungleich)
	bne .lbl62
	; Nullbyte gefunden, berechne jetzt Differenz
	; lade "eos" nach X1
	ldr x1, [sp,24]
	; lade "str" nach X0
	ldr x0, [sp,8]
	; berechne Differenz
	sub x0, x1, x0
	; dekrementiere Ergebnis
	sub w0, w0, #1
	; Funktionsepiilog
	add sp, sp, 32
ret
```

Es ist umfangreicher. Die Variablen werden hier viel im Speicher (lokaler Stack) herumgeschoben. Der obige Fehler findet sich auch hier: das Dekrementieren geschieht nur in einem 32-bit-Teil des Registers.

**MIPS**

Listing 1.161: **German text placeholder GCC 4.4.5 (IDA)**

```assembly
my_strlen:
	; "eos" Variable bleibt stets in $v1:
	move $v1, $a0

loc_4:
	; lade Byte von der Adresse in "eos" nach $a1:
	lb $a1, 0($v1)
	or $at, $zero ; lade delay slot, NOP
	; ist das geladene Byte ungleich 0, springe nach loc_4:
	bnez $a1, loc_4
	; erhöhe "eos" in jedem Falle:
	addiu $v1, 1 ; branch delay slot
	; Schleife beendet. Invertiere "str" Variable:
	nor $v0, $zero, $a0
	; $v0=-str-1
	jr $ra
	; return value = $v1 + $v0 = eos + (-str-1) = eos - str - 1
	addu $v0, $v1, $v0 ; branch delay slot
```

MIPS besitzt keinen NOT Befehl, dafür aber den Befehl NOR, welcher der Funktion OR + NOT entspricht.

Diese Funktion wird häufig in der Digitaltechnik verwendet\(^\text{101}\).

\(^{101}\) NOR wird „universelles Gatter“ genannt

Die NOT Funktion ist hier also durch NOR DST, $ZERO$, SRC implementiert.

Aus dem Grundlagenteil 2.1 on page 408 wissen wir, dass das bitweise invertieren einer vorzeichenbehäfteten Zahl gerade einem Wechsel des Vorzeichens mit anschließender Subtraktion von 1 entspricht.

Was NOT hier also tut, ist, den Wert von $str$ in $-str - 1$ umzuwandeln. Die folgende Addition bereitet das Ergebnis vor.

### 1.16 Ersetzen von arithmetischen Operationen

**German text placeholder**

#### 1.16.1 Multiplikation

**Multiplikation durch Addition**

Hier ist ein einfaches Beispiel:

```c
unsigned int f(unsigned int a)
{
    return a*8;
}
```

Die Multiplikation mit 8 wird durch 3 Additionsbefehle ersetzt, welche das gleiche Ergebnis erzielen. Offenbar hat der MSVC Optimierer entschieden, dass der Code so schneller sein kann.

Listing 1.162: **German text placeholder** MSVC 2010

```assembly
_TEXT SEGMENT
_a$ = 8 ; size = 4
_f PROC
  mov   eax, DWORD PTR _a$[esp-4]
  add   eax, eax
  add   eax, eax
  add   eax, eax
  ret   0
_f ENDP
_TEXT ENDS
END
```

**Multiplikation durch Verschieben**

Multiplikation mit und Divisionen durch Zahlen, die Potenzen von 2 sind, werden oft durch Schiebebefehle (oft auch Shifting genannt) ersetzt.

```c
unsigned int f(unsigned int a)
{
    return a*4;
}
```

Listing 1.163: **German text placeholder** MSVC 2010

```assembly
_a$ = 8 ; size = 4
_f PROC
  push  ebp
  mov   ebp, esp
  mov   eax, DWORD PTR _a$[ebp]
```

189

Der Befehl für Linksverschiebung funktioniert wie folgt:

```
7 6 5 4 3 2 1 0
```

CF
```
7 6 5 4 3 2 1 0
```

Die beiden rechts angefügten Bits sind stets Nullen.

**Multiplikation mit 4 in ARM:**

Listing 1.164: German text placeholder Keil 6/2013 (German text placeholder)

```
f PROC
  LSL    r0, r0, #2
  BX     lr
ENDP
```

**Multiplikation mit 4 in MIPS:**

Listing 1.165: German text placeholder GCC 4.4.5 (IDA)

```
jr    $ra
sll   $v0, $a0, 2 ; branch delay slot
```

SLL bedeutet „Shift Left Logical“.

**Multiplikation durch Verschieben, Subtrahieren und Addieren**

Es ist auch möglich die Multiplikation zu ersetzen, wenn man mit Zahlen wie 7 oder 17 multipliziert, wenn Verschiebung verwendet wird. Die zugrundeliegende Mathematik ist relativ einfach.

**32-bit**

```c
#include <stdint.h>

int f1(int a)
{
    return a*7;
};

int f2(int a)
{
    return a*28;
};

int f3(int a)
{
    return a*17;
};
```

**x86**
Listing 1.166: `German text placeholder` MSVC 2012

```assembly
; a*7
_a$ = 8
_f1 PROC
  mov ecx, DWORD PTR _a$(esp-4)
  ; ECX=a
  lea eax, DWORD PTR [ecx*8]
  ; EAX=ECX*8
  sub eax, ecx
  ; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
  ret 0
_f1 ENDP

; a*28
_a$ = 8
_f2 PROC
  mov ecx, DWORD PTR _a$(esp-4)
  ; ECX=a
  lea eax, DWORD PTR [ecx*8]
  ; EAX=ECX*8
  sub eax, ecx
  ; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
  ret 0
_f2 ENDP

; a*17
_a$ = 8
_f3 PROC
  mov eax, DWORD PTR _a$(esp-4)
  ; EAX=a
  shl eax, 4
  ; EAX=EAX<<4=EAX*16=a*16
  add eax, DWORD PTR _a$(esp-4)
  ; EAX=EAX+a=a*16+a=a*17
  ret 0
_f3 ENDP

ARM

Keil im ARM mode benutzt den Umwandler zur Verschiebung im zweiten Operanden:

Listing 1.167: `German text placeholder` Keil 6/2013 (`German text placeholder`)

```assembly
; a*7
||f1|| PROC
  RSB r0, r0, r0, LSL #3
  ; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
  BX lr
  ENDP

; a*28
||f2|| PROC
  RSB r0, r0, r0, LSL #3
  ; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
  LSL r0, r0, #2
  ; R0=R0<<2=R0*4=a*7*4=a*28
  BX lr
  ENDP

; a*17
||f3|| PROC
  ADD r0, r0, r0, LSL #4
  ; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
  BX lr
  ENDP
```

191
Da es im Thumb mode keine solchen Umwandler gibt, kann folglich f2() nicht optimiert werden:

Listing 1.168: German text placeholder Keil 6/2013 (German text placeholder)

<table>
<thead>
<tr>
<th>a*7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>LSLS r1, r0, #3</td>
</tr>
<tr>
<td>R1=R0&lt;&lt;3=a&lt;&lt;3=a*8</td>
</tr>
<tr>
<td>SUBS r0, r1, r0</td>
</tr>
<tr>
<td>R0=R1-R0=a<em>8-a=a</em>7</td>
</tr>
<tr>
<td>BX lr</td>
</tr>
<tr>
<td>ENDP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a*28</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>MOV r1, #0x1c ; 28</td>
</tr>
<tr>
<td>R1=28</td>
</tr>
<tr>
<td>MULS r0, r1, r0</td>
</tr>
<tr>
<td>R0=R1<em>R0=28</em>a</td>
</tr>
<tr>
<td>BX lr</td>
</tr>
<tr>
<td>ENDP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a*17</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>LSLS r1, r0, #4</td>
</tr>
<tr>
<td>R1=R0&lt;&lt;4=R0<em>16=a</em>16</td>
</tr>
<tr>
<td>ADDS r0, r0, r1</td>
</tr>
<tr>
<td>R0=R0+R1=a+a<em>16=a</em>17</td>
</tr>
<tr>
<td>BX lr</td>
</tr>
<tr>
<td>ENDP</td>
</tr>
</tbody>
</table>

**MIPS**

Listing 1.169: German text placeholder GCC 4.4.5 (IDA)

```mips
_f1:
    sll $v0, $a0, 3
; $v0 = $a0<<3 = $a0*8
    jr $ra
    subu $v0, $a0 ; branch delay slot
; $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7
_f2:
    sll $v0, $a0, 5
; $v0 = $a0<<5 = $a0*32
    sll $a0, 2
; $a0 = $a0<<2 = $a0*4
    jr $ra
    subu $v0, $a0 ; branch delay slot
; $v0 = $a0*32-$a0*4 = $a0*28
_f3:
    sll $v0, $a0, 4
; $v0 = $a0<<4 = $a0*16
    jr $ra
    addu $v0, $a0 ; branch delay slot
; $v0 = $a0*16+$a0 = $a0*17
```

**64-bit**

```c
#include <stdint.h>

int64_t f1(int64_t a)
{
    return a*7;
}
```
int64_t f2(int64_t a)
{
    return a*28;
};

int64_t f3(int64_t a)
{
    return a*17;
};

x64

Listing 1.170: German text placeholder MSVC 2012

```c
; a*7
f1:
    lea    rax, [0+rdi*8]
    ; RAX=RD!8=a*8
    sub    rax, rdi
    ; RAX=RAX-RDI=a*8-a=a*7
    ret

; a*28
f2:
    lea    rax, [0+rdi*4]
    ; RAX=RD!4=a*4
    sal    rdi, 5
    ; RDI=RDI<5=RD!32=a*32
    sub    rdi, rax
    ; RDI=RDI-RAX=a*32-a*4=a*28
    mov    rax, rdi
    ret

; a*17
f3:
    mov    rax, rdi
    sal    rax, 4
    ; RAX=RAX<4=a*16
    add    rax, rdi
    ; RAX=a*16+a=a*17
    ret
```

ARM64

GCC 4.9 für ARM64 fasst sich dank der Verschiebe-Umwandler ebenfalls kurz:

Listing 1.171: German text placeholder GCC (Linaro) 4.9 ARM64

```assembly
; a*7
f1:
    lsl    x1, x0, 3
    ; X1=X0<<3=X0*8=a*8
    sub    x0, x1, x0
    ; X0=X1-X0=a*8-a=a*7
    ret

; a*28
f2:
    lsl    x1, x0, 5
    ; X1=X0<<5=a*32
    sub    x0, x1, x0, lsl 2
    ; X0=X1-X0<<2=a*32-a<<2=a*32-a*4=a*28
    ret

; a*17
f3:
    add    x0, x0, x0, lsl 4
```

193
Booths Multiplikationsalgorithmus

Es gab Zeiten, in denen Computer groß und so teuer waren, dass einige von ihnen keinen Hardwaresupport für die Multiplikation in der CPU besaßen, so zum Beispiel der Data General Nova. Wenn dort eine Multiplikation benötigt wurde, musste diese softwareseitig abgebildet werden, zum Beispiel durch Booths Multiplikationsalgorithmus. Dabei handelt es sich um einen Algorithmus zur Multiplikation, welcher lediglich Additionen und Verschiebeoperationen verwendet.

Zwar gehen moderne optimierende Compiler hier anders vor, aber das Ziel (die Multiplikation) und die Ressourcenfrage (schnellere Operationen) sind gleich.

1.16.2 Division

Division durch Verschieben

Beispiel der Division durch 4:

```c
unsigned int f(unsigned int a) {
    return a/4;
}
```

Wir betrachten (MSVC 2010):

```
Listing 1.172: MSVC 2010

_a$ = 8 ; size = 4
_f PROC
    mov eax, DWORD PTR _a$[esp-4]
    shr eax, 2
    ret 0
_f ENDP
```

Der Befehl `SHR` (Shift Right) verschiebt die Zahl in diesem Beispiel um 2 Bits nach rechts. Die beiden freien Bits am linken Rand (dies sind die beiden höchstwertigsten Bits) werden auf null gesetzt. Die beiden niederwertigsten Bits werden entfernt. Diese beiden entfernten Bits entsprechen genau dem Rest der Division.

Der `SHR` Befehl funktioniert genau wie `SHL`, aber in die entgegengesetzte Richtung.

Das Vorgehen kann leicht verdeutlicht werden, wenn wir es an der Zahl 23 im Dezimalsystem veranschaulichen. Die 23 kann einfach durch 10 geteilt werden, indem die letzte Ziffer (3–Rest der Division) entfernt wird. Die 2 bleibt bei der Division als ganzzahliger Quotient übrig.

Der Rest wird also entfernt, was aber kein Problem darstellt, da wir hier ausschließlich mit ganzzahligen Werten arbeiten und nicht mit reellen Zahlen.

Division durch 4 in ARM:

```
Listing 1.173: German text placeholder Keil 6/2013 (German text placeholder)

f PROC
    LSR     r0,r0,#2
    BX      lr
ENDP
```

Division by 4 in MIPS:

Listing 1.174: German text placeholder GCC 4.4.5 (IDA)

```
jr $ra
srl $v0, $a0, 2 ; branch delay slot
```

Der Befehl SRL steht für „Shift Right Logical“.

### 1.16.3 German text placeholder
- [http://challenges.re/59](http://challenges.re/59)

### 1.17 Gleitkommaeinheit

Die FPU is ein Gerät innerhalb der CPU, welche speziell für den Umgang mit Fließkommazahlen ausgelegt ist. In der Vergangenheit wurde die FPU auch als „Koprozessor“ bezeichnet und sie befindet sich neben der CPU.

#### 1.17.1 IEEE 754

Eine Zahl besteht gemäß IEEE 754 Format aus einem Vorzeichen, einer Mantisse (auch Bruch genannt) und einem Exponenten.

#### 1.17.2 x86

Es lohnt sich einen Blick auf die Stackmaschine zu werfen oder die Grundlagen der Sprache Forth zu erlernen, bevor man die FPU in x86 genauer untersucht.

Es ist interessant zu wissen, dass sich der Koprozessor in der Vergangenheit (vor dem 80486 Chip) auf einem separaten Chip befand und nicht immer auf dem Mainboard vorinstalliert war. Es war möglich, diesen separat zu kaufen und zu installieren.

Seit der 80486 DX CPU ist die FPU in die CPU integriert.

Der Befehl FWAIT erinnert uns an diese Tatsache—er lässt die CPU in einen Wartezustand wechseln, in dem sie verbleibt, bis die FPU ihre Arbeit beendet hat.

Ein anderes Überbleibsel ist die Tatsache, dass die Opcodes der FPU Befehle mit sogenannten „escape“-Opcodes (D8..DF, d.h. Opcodes, die an einen separaten Koprozessor übergeben werden) beginnen.

Die FPU besitzt einen Stack, auf dem 8 80-bit-Register Platz finden, wobei jedes Register eine Zahl im IEEE 754 Format aufnehmen kann.

Es gibt ST(0)..ST(7). Verkürzend stellen IDA und OllyDbg ST(0), welches in manchen (Hand-)büchern als „Stack Top“ dargestellt wird, als ST dar.

#### 1.17.3 ARM, MIPS, x86/x64 SIMD

In ARM und MIPS ist die FPU kein Stack, sondern ein Registersatz.

Das gleiche Paradigma wird in den SIMD-Erweiterungen von x86/x64 CPUs verwendet.

#### 1.17.4 C/C++

Die Standardsparche C/C++ bietet zumindest zwei unterschiedliche Fließkommazahlentypen, `float` (einfache Genauigkeit, 32 Bit) und `double` (doppelte Genauigkeit, 64 Bit).

102John Carmack z.B. verwendete Fixpunktarithmetikwerte seinem Videospiel Doom, gespeichert in 32-bit GPR Registern (16 Bit für den Hauptteil und weitere 16 Bit für den gebrochenen Teil), sodass Doom auf 32-Bit-Computern ohne FPU, d.h., 80386 und 80486 SX, lauffähig war.

103Fließkommazahlen mit einfacher Genauigkeit werden auch im Abschnitt German text placeholder (1.21.6 on page 348) behandelt.

GCC unterstützt im Gegensatz zu MSVC ebenfalls den *long double* Typ (*erweiterte Genauigkeit* 80 Bit).

Der *float* Typ benötigt dieselbe Anzahl an Bits wie der *int* Typ in 32-Bit-Umgebungen, aber die Darstellung der Zahlen ist komplett verschieden voneinander.

### 1.17.5 German text placeholder

**German text placeholder**

```c
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
};

int main()
{
    printf ("%f\n", f(1.2, 3.4));
};
```

#### x86

**MSVC**

Kompilieren mit MSVC 2010 liefert:

Listing 1.175: MSVC 2010: f()

```asm
CONST SEGMENT
    real@4010666666666666r DQ 0401066666666666 ; 4.1
CONST ENDS
CONST SEGMENT
    real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14
CONST ENDS
_TEXT SEGMENT
    a$ = 8 ; size = 8
    b$ = 16 ; size = 8
_f PROC
    push ebp
    mov ebp, esp
    fld QWORD PTR _a$[ebp]
    ; aktueller Stand des Stacks: ST(0) = _a
    fddiv QWORD PTR __real@40091eb851eb851f
    ; aktueller Stand des Stacks: ST(0) = Ergebnis von _a geteilt durch 3.14
    fld QWORD PTR _b$[ebp]
    ; aktueller Stand des Stacks: ST(0) = _b;
    ; ST(1) = Ergebnis von _a geteilt durch 3.14
    fmul QWORD PTR __real@4010666666666666
    ; aktueller Stand des Stacks:
    ; ST(0) = Ergebnis von _b * 4.1;
    ; ST(1) = Ergebnis von _a geteilt durch 3.14
    faddp ST(1), ST(0)
    ; aktueller Stand des Stacks: ST(0) = Ergebnis der Addition
    pop ebp
```

196
FLD nimmt 8 Byte vom Stack und lädt die Zahl in das ST(0) Register, wobei diese automatisch in das interne 80-bit-Format (erweiterte Genauigkeit) konvertiert wird.

FDIV teilt den Wert in ST(0) durch die Zahl, die an der Adresse __real@40091eb851eb851f gespeichert ist —der Wert 3.14 ist hier kodiert. Die Syntax des Assemblers erlaubt keine Fließkommazahlen, sodass wir hier die hexadezimale Darstellung von 3.14 im 64-bit IEEE 754 Format finden.

Nach der Ausführung von FDIV enthält ST(0) den Quotienten.

Es gibt übrigens auch noch den FDIVP Befehl, welcher ST(1) durch ST(0) teilt, beide Werte vom Stack holt und das Ergebnis ebenfalls auf dem Stack ablegt. Wer mit der Sprache Forth vertraut ist, erkennt schnell, dass es sich hier um eine Stackmaschine handelt.

Der nachfolgende FLD Befehl speichert den Wert von $b$ auf dem Stack.

Anschließend wir der Quotient in ST(1) abgelegt und ST(0) enthält den Wert von $b$.

Der nächste FMUL Befehl führt folgende Multiplikation aus: $b$ aus Register ST(0) wird mit dem Wert an der Speicherstelle __real@4010666666666666 (hier befindet sich die Zahl 4.1) multipliziert und hinterlässt das Ergebnis im ST(8) Register.

Der letzte FADDP Befehl addiert die beiden Werte, die auf dem Stack zuoberst liegen, speichert das Ergebnis in ST(1) und holt dann den Wert von ST(0) vom Stack, wobei das oberste Element auf dem Stack in ST(0) gespeichert wird.

Die Funktion muss ihr Ergebnis im ST(0) Register zurückgeben, sodass außer dem Funktionsepiolog nach FADDP keine weiteren Befehle mehr folgen.
MSVC + OllyDbg

Wie sehen wie zunächst FLD einen Wert (1.2) von Stack lädt und diesen in ST(0) ablegt:

Abbildung 1.63: OllyDbg: der erste FLD wurde ausgeführt

Aufgrund der unvermeidlichen Konversionsfehler von der 64-bit IEEE 754 Fließkommazahl in ein 80-bit-Format (das intern in der FPU verwendet wird), sehen wird hier 1.1999..., was näherungsweise 1.2 entspricht.

EIP zeigt nun auf den nächsten Befehl (FDIV), der eine double-Zahl (eine Konstante) aus dem Speicher lädt. Zur besseren Übersicht zeigt OllyDbg deren Wert an: 3.14
Verfolgen wir das ganze etwas weiter. FDIV wurde ausgeführt, nun enthält ST(0) also 0.382… (Quotient):

Abbildung 1.64: OllyDbg: FDIV wurde ausgeführt
Dritter Schritt: der nächste FLD Befehl wurde ausgeführt; er lud 3.4 nach ST(0) (wir sehen hier den Näherungswert 3.39999…):

Abbildung 1.65: OllyDbg: der zweite FLD wurde ausgeführt

Gleichzeitig wird der Quotient nach ST(1) verschoben. In diesem Moment zeigt EIP auf den nächsten Befehl: FMUL. Dieser lädt die Konstante 4.1 aus dem Speicher, wie OllyDbg zeigt.
Dann: FMUL wurde ausgeführt, sodass das Produkt jetzt in ST(0) liegt.

Abbildung 1.66: OllyDbg: FMUL wurde ausgeführt
Dann: der Befehl FADDP wurde ausgeführt, sodass sich in ST(0) nunmehr das Ergebnis der Addition befindet und ST(1) wird gelöscht:

Abbildung 1.67: OllyDbg: FADDP wurde ausgeführt

Das Ergebnis bleibt in ST(0), denn die Funktion liefert ihren Rückgabewert über ST(0) zurück.

Später liest `main()` diesen Wert aus dem Register.

Wir sehen außerdem etwas Ungewöhnliches: der Wert 13.93…befindet sich nun in ST(7). Warum?

Wie bereits vorher in diesem Buch beschrieben, bilden die FPU einen Stack: 1.17.2 on page 195. Dabei handelt es sich jedoch um eine vereinfachte Darstellung.

Stellen wir uns vor, es wäre genau wie beschrieben in Hardware implementiert, dann müssten die Inhalte aller 7 Register während der push und pop-Befehle in jeweils benachbarte Register verschoben (oder kopiert) werden und das würde eine Menge Aufwand bedeuten.

In Wirklichkeit hat die FPU nur 8 Register und einen Pointer (TOP genannt), der die Registernummer enthält, die derzeit oben auf dem Stack liegt.

Wenn ein Wert auf dem Stack abgelegt wird, zeigt TOP auf das nächste verfügbare Register und dann wird der Wert in dieses Register geschrieben.

Dieser Vorgang läuft umgekehrt ab, wenn ein Wert vom Stack geholt wird, aber das freigewordene Register wird nicht gelöscht (es könnte möglicherweise gelöscht werden, aber dies würde einen Mehraufwand bedeuten und die Performance herabsetzen). Genau das sehen wir hier.

Man kann sagen, dass FADDP die Summe auf dem Stack gespeichert hat und dann ein Element vom Stack geholt hat.

Aber in Wirklichkeit hat der Befehl die Summe gespeichert und dann TOP verschoben.

Genauer gesagt bilden die Register der FPU einen Ringpuffer.
GCC

GCC 4.4.1 (mit der Option -O3) erzeugt fast den gleichen Code, nur leicht verändert.

Listing 1.176: German text placeholder GCC 4.4.1

```c
public f
f proc near
arg_0 = qword ptr 8
arg_8 = qword ptr 10h
push ebp
fld ds:dbl_8048608 ; 3.14
; Stand des Stacks: ST(0) = 3.14
mov ebp, esp
fdiavr [ebp+arg_0]
; Aktueller Stand des Stacks: ST(0) = Ergebnis der Division
fld ds:dbl_8048610 ; 4.1
; Aktueller Stand des Stacks: ST(0) = 4.1, ST(1) = Ergebnis der Division
fmul [ebp+arg_8]
; Aktueller Stand des Stacks: ST(0) = Ergebnis der Multiplikation, ST(1) = Ergebnis der Division
pop ebp
faddp st(1), st
; Aktueller Stand des Stacks: ST(0) = Ergebnis der Addition
retn
f endp
```

Der Unterschied besteht darin, dass zuerst 3.14 auf dem Stack (in ST(0)) abgelegt wird und danach der Wert in arg_0 durch den Wert in ST(0) geteilt wird.


FADDP addiert die beiden Werte und holt auch einen Wert vom Stack. Nach der Ausführung steht die Summe in ST(0).

ARM: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)

Bis die Unterstützung für Fließkommaarithmetik in ARM standardisiert wurde, fügten einige Hersteller von Prozessoren ihre eigenen Befehlserweiterungen hinzu. Schließlich wurde VFP (Vector Floating Point) standardisiert.

Ein wichtiger Unterschied zum x86 ist, dass in es in ARM keinen Stack gibt, sondern man nur mit den Registern arbeitet.

Listing 1.177: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)

```asm
f
VLDR D16, =3.14
VMOV D17, R0, R1 ; lade "a"
VMOV D18, R2, R3 ; lade "b"
VDIV.F64 D16, D17, D16 ; a/3.14
VLDR D17, =4.1
VMUL.F64 D17, D18, D17 ; b*4.1
VADD.F64 D16, D17, D16 ; +
VMOV R0, R1, D16
BX LR
```

203
Hier sehen wir, dass einige neue Register mit einem D als Präfix verwendet werden.

Bei diesen handelt es sich um 64-bit-Register; es gibt 32 von ihnen und sie können sowohl für Fließkommazahlen (doppelte Genauigkeit (double)) als auch für SIMD (heißt hier in ARM NEON) benutzt werden.

Es gibt also 32 32-bit-S-Register vorgesehen für Fließkommazahlen in einfacher Genauigkeit (float).

Es ist leicht zu merken: D-Register sind für Zahlen in doppelter Genauigkeit, während S-Register für einfache Genauigkeit (engl. single) vorgesehen sind. Mehr dazu hier: ?? on page ??

Beide Konstanten (3.14 und 4.1) werden im IEEE 754 Format im Speicher abgelegt.

Wie man leicht sieht sind VLDR und VMOV analog zu den LDR und MOV Befehlen, aber arbeiten auf D-Registern.

Es muss angemerkt werden, dass diese Befehle genau wie die D-Register nicht nur für Fließkommazahlen vorgesehen sind, sondern ebenfalls für SIMD (NEON) Operationen verwendet werden können, was wir im folgenden zeigen werden.

Die Parameter werden der Funktion auf übliche Weise über die R-Register übergeben, aber da jede Zahl in doppelter Genauigkeit eine Größe von 64 Bit hat werden jeweils zwei R-Register benötigt, um eine Zahl zu übergeben.

Der Befehl VMOV D17, R0, R1 zu Beginn, fasst zwei 32-Bit-Werte aus R0 und R1 zu einem 64-Bit-Wert zusammen und speichert diesen in D17.

VMOV R0, R1, D16 ist die umgekehrte Operation: was vorher in D16 war, wird in zwei Register, R0 und R1 aufgeteilt, denn eine Zahl in doppelter Genauigkeit, die 64 Bit Speicherplatz benötigt, wird über R0 und R1 zurückgegeben.

VDIV, VMUL und VADD sind Befehle zur Verarbeitung von Fließkommazahlen, die Quotient, Produkt bzw. Summe berechnen.

Der Code für Thumb-2 ist identisch.

**ARM: German text placeholder Keil 6/2013 (German text placeholder)**

```
f
   PUSH {R3-R7,LR}
   MOVS R7, R2
   MOVS R4, R3
   MOVS R5, R0
   MOVS R6, R1
   LDR R2, =0x66666666 ; 4.1
   LDR R3, =0x40106666
   MOVS R0, R7
   MOVS R1, R4
   BL __aeabi_dmul
   MOVS R7, R0
   MOVS R4, R1
   LDR R2, =0x51EB851F ; 3.14
   LDR R3, =0x40091EB8
   MOVS R0, R5
   MOVS R1, R6
   BL __aeabi_ddiv
   MOVS R2, R7
   MOVS R3, R4
   BL __aeabi_dadd
   POP {R3-R7,PC}
```

; 4.1 im IEEE 754 Format:

dword_364 DCD 0x66666666 ; DATA XREF: f+A
dword_368 DCD 0x40106666 ; DATA XREF: f+C

; 3.14 im IEEE 754 Format:
dword_36C DCD 0x51EB851F ; DATA XREF: f+1A
dword_370 DCD 0x40091EB8 ; DATA XREF: f+1C
Keil erzeugte Code für einen Prozessor ohne FPU oder NEON Unterstützung.

Die Fließkommazahlen in doppelter Genauigkeit werden über die üblichen R-Register übergeben und anstelle von FPU-Befehlen werden Programmbibliotheken (wie z.B. __aeabi_dmul, __aeabi_ddiv, __aeabi_dadd) aufgerufen, welche Multiplikation, Division und Addition auf Fließkommazahlen emulieren.

Diese Vorgehensweise ist natürlich langsamer als der FPU-Koprozessor, aber es ist besser als nichts. Übrigens waren ähnliche FPU-emulierende Programmbibliotheken auch in der x86-Welt sehr beliebt als Koprozessoren selten und teuer waren und nur auf wertvollen Computern installiert waren.

Die Emulation des FPU-Koprozessors wird soft float oder armel (in der ARM-Welt) genannt, wohingegen die FPU-Befehle des Koprozessors hard float oder armhf genannt werden.

**ARM64: German text placeholder GCC (Linaro) 4.9**

Sehr kompakter Code:

Listing 1.178: German text placeholder GCC (Linaro) 4.9

```assembly
f:
; D0 = a, D1 = b
    ldr   d2, .LC25 ; 3.14
    ; D2 = 3.14
    fdiv  d0, d0, d2
    ; D0 = D0/D2 = a/3.14
    ldr   d2, .LC26 ; 4.1
    ; D2 = 4.1
    fmadd d0, d1, d2, d0
    ; D0 = D1*D2+D0 = b*4.1+a/3.14
    ret

; Konstanten im IEEE 754 Format:
.LC25:
    .word   1374389535 ; 3.14
    .word   1074339512
.LC26:
    .word   1717986918 ; 4.1
    .word   1074816614

ARM64: German text placeholder GCC (Linaro) 4.9

Listing 1.179: German text placeholder GCC (Linaro) 4.9

```

Bemerkenswert ist, dass ARM64 64-Bit-Register besitzt und die D-Register ebenfalls 64 Bit breit sind. Dadurch steht es dem Compiler frei Werte von Typ *double* in GPRs anstelle auf dem lokalen Stack zu speichern. Dies ist in 32-bit-CPUs nicht möglich.

Wiederum kann man als Übung versuchen diese Funktion manuell zu optimieren ohne neue Befehl wie FMADD einzuführen.

### 1.17.6 German text placeholder

```
#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %.lf\n", pow (32.01,1.54));

    return 0;
}
```

#### x86

Schauen wir uns an, was wir in MSVC 2010 erhalten:

```
Listing 1.180: MSVC 2010

CONST SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r ; 1.54
CONST ENDS

__main PROC
    push ebp
    mov ebp, esp
    sub esp, 8 ; reserviere Speicher für erste Variable
    fld QWORD PTR __real@3ff8a3d70a3d70a4
    fstp QWORD PTR [esp]
    sub esp, 8 ; reserviere Speicher für zweite Variable
    fld QWORD PTR __real@40400147ae147ae1
    fstp QWORD PTR [esp]
    call _pow
    add esp, 8 ; gib Platz für eine Variable frei.

    ; im lokalen Stack sind hier immer noch 8 Byte für uns reserviert.
```
Ergebnis jetzt in ST(0)

```
fstp  QWORD PTR [esp] ; verschiebe Ergebnis von ST(0)
push  OFFSET $SG2651
call  _printf
add   esp, 12
xor   eax, eax
pop   ebp
ret   0
```

.FirstOrDefault nicht in ST(0)

```
verschiebe Ergebnis von ST(0)
```

Auf den lokalen Stack für printf()

```
push  OFFSET $SG2651
call  _printf
add   esp, 12
xor   eax, eax
pop   ebp
ret   0
```

_main ENDP

FLD und FSTP verschieben Variablen zwischen Datensegment und dem FPU Stack. `pow()` nimmt beide Werte vom Stack der FPU und gibt ihr Ergebnis über das ST(0) Register zurück. Die Funktion `printf()` nimmt 8 Byte vom lokalen Stack und interpretiert diese als Variable von Typ `double`.

Übrigens könnte hier auch ein Paar MOV Befehle verwendet werden, um die Werte aus dem Speicher zu holen und auf den Stack zu legen, denn die Werte sind im Speicher im IEEE 754 Format abgelegt und `pow()` arbeitet mit diesem Format, sodass keine Umwandlung notwendig ist. Genau so wird es im folgenden Beispiel für ARM auch gemacht: 1.17.6

**ARM + German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**

```
_main

var_C = -0xC

PUSH  {R7,LR}
MOV   R7, SP
SUB   SP, SP, #4
VLDR  D16, =32.01
VMOV  R0, R1, D16
VLDR  D16, =1.54
VMOV  R2, R3, D16
BLX   _pow
VMOV  D16, R0, R1
MOV   R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"
ADD   R0, PC
VMOV  R1, R2, D16
BLX   _printf
MOV   R1, 0
STR   R0, [SP,#0xC+var_C]
MOV   R0, R1
ADD   SP, SP, #4
POP   {R7,PC}
```

Wie bereits vorher erwähnt werden Pointer auf 64-Bit-Fließkommazahlen über ein Paar von R-Registern übergeben.

Dieser Code ist leicht redundant (sicherlich aufgrund der deaktivierten Optimierung), da es möglich ist Werte direkt in die R-Register zu laden, ohne die D-Register zu verwenden.

Wie wir also sehen erhält die _pow Funktion ihr erster Argument in R0 und R1 und das zweite in R2 und R3. Die Funktion speichert ihr Ergebnis in R0 und R1. Das Ergebnis von _pow wird zunächst nach D16 und anschließend in das Paar R1 und R2 verschoben, von wo aus printf() das Ergebnis übernimmt.

**ARM + German text placeholder Keil 6/2013 (German text placeholder)**

```
_main

STMFD  SP!, {R4-R6,LR}
LDR   R2, =0xA3D70A4 ; y
LDR   R3, =0xFF8A3D7
LDR   R0, =0xAE147AE1 ; x
```

104 eine Standard-C-Funktion, die eine Zahl potenziert
Die D-Register werden hier nicht verwendet, sondern nur Paare von R-Registern.

ARM64 + German text placeholder GCC (Linaro) 4.9

Listing 1.181: German text placeholder GCC (Linaro) 4.9

Die Konstanten werden nach D0 und D1 geladen: pow() übernimmt sie von dort. Das Ergebnis befindet sich nach der Ausführung von pow() in D0. Es wird ohne weitere Änderung oder Verschiebung an die Funktion printf() übergeben, da printf() ganzzahlige Werte und Pointer aus X-Registern, Fließkommaparameter jedoch aus D-Registern übernimmt.

1.17.7 German text placeholder

German text placeholder

```
#include <stdio.h>

double d_max (double a, double b)
{
    if (a>b)
        return a;
    return b;
};

int main()
{
```

208
```c
printf ("%f\n", d_max (1.2, 3.4));
printf ("%f\n", d_max (5.6, -4));
```

### x86

**German text placeholder MSVC**

MSVC 2010 erzeugt den folgenden Code:

#### Listing 1.182: German text placeholder MSVC 2010

```assembly
PUBLIC  _d_max
_TEXT   SEGMENT
_a$   = 8 ; size = 8
_b$   = 16 ; size = 8
_d_max   PROC
    push ebp
    mov ebp, esp
    fld QWORD PTR _b$[ebp]
    ; Zustand des Stacks: ST(0) = _b
    ; vergleiche _b (ST(0)) und _a, und hole Register vom Stack
    fcomp QWORD PTR _a$[ebp]
    ; Stack ist jetzt leer
    fnstsw ax
    test ah, 5
    jp SHORT $LN1@d_max
    ; hierher gelangen wir nur, falls a>b
    fld QWORD PTR _a$[ebp]
    jmp SHORT $LN2@d_max
$LN1@d_max:
    fld QWORD PTR _b$[ebp]
$LN2@d_max:
    pop ebp
    ret 0
_d_max   ENDP
```

Der Befehl FLD lädt _b nach ST(0).

FCOMP vergleicht den Wert in ST(0) mit dem Wert, der sich in _a befindet und setzt die C3/C2/C0 im FPU Status Register entsprechend. Das Statusregister ist ein 16-Bit-Register, das den aktuelle Zustand der FPU abbildet.

Nachdem die Bits gesetzt worden sind, nimmt der FCOMP Befehl auch eine Variable vom Stack. Dieses Verhalten unterscheidet ihn von FCOM, der einfach zwei Werte vergleicht und den Stack unangetastet lässt.

Leider verfügen CPUs vor Intel P6\(^\text{105}\) über keinerlei bedingte Sprungbefehle, die die C3/C2/C0 prüfen.

After the bits are set, the FCOMP instruction also pops one variable from the stack. This is what distinguishes it from FCOM, which is just compares values, leaving the stack in the same state. Vielleicht ist diese Tatsache historisch begründet (man erinnere sich: die FPU war früher ein eigener Chip).

Moderne CPUs, beginnend mit Intel P6 haben FCOMI/FCOMIP/FUCOMI/FUCOMIP Befehle –welche im Prinzip das gleiche tun, aber die ZF/PF/CF Flags der CPU verändern können.

Der FNSTSW Befehl kopiert das FPU Statusregister nach AX. C3/C2/C0 werden an den Stellen 14/10/8 abgelegt, sie befinden sich im AX Register an den gleichen Stellen und sie werden alle in höherwertigen Teil von AX — AH abgelegt.

- Falls in unserem Beispiel \(b > a\), dann werden die C3/C2/C0 Bits wie folgt gesetzt: 0, 0, 0.
- Falls \(a > b\), dann ist das Bitmuster: 0, 0, 1.

\(^\text{105}\) Intel P6 ist Pentium Pro, Pentium II, etc.
• Falls \( a = b \), dann ist das Bitmuster: 1, 0, 0.
• Wenn das Ergebnis (z.B. im Fehlerfall) ungeordnet ist, dann werden die Bits wie folgt gesetzt: 1,1,1.

So werden die C3/C2/C0 Bits im AX Register angeordnet:

<table>
<thead>
<tr>
<th>14</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>C2</td>
<td>C1</td>
<td>C0</td>
</tr>
</tbody>
</table>

So werden die C3/C2/C0 Bits im AH Register angeordnet:

<table>
<thead>
<tr>
<th>6</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>C2</td>
<td>C1</td>
<td>C0</td>
</tr>
</tbody>
</table>

Nach der Ausführung von test ah, 5106 werden nur die C0 und C2 Bits (an den Stellen 0 und 2) betrachtet, alle übrigen Bits werden einfach überlesen.

Werfen wir nun einen Blick auf ein anderes bemerkenswertes historisches Überbleibsel: das parity flag.

Dieses Flag wird auf 1 gesetzt, falls die Anzahl der Einsen im Ergebnis der letzten Berechnung gerade ist und auf 1, falls dies nicht der Fall ist.

Schlagen wir in der Wikipedia nach107:

Ein guter Grund das Parity Flag abzufragen, hat tatsächlich gar nichts mit Parität zu tun. Die FPU hat vier Bedingungsflags (C0 bis C3), aber diese können nicht direkt abgefragt werden, sondern müssen zunächst in das Flags Register kopiert werden. Wenn dies geschieht, wird C0 im Carry Flag abgelegt, C2 im Parity Flag und C3 im Zero Flag. Das C2 Flag ist gesetzt, wenn z.B. unvergleichbare Fließkommawerte (NaN oder nicht unterstütztes Format) über der FUCOM Befehl miteinander verglichen werden. (Übersetzung aus der englischen Wikipedia.)

Wie in der Wikipedia dargestellt wird das Parity Flag manchmal im FPU Code verwendet; schauen wir uns genauer an wie das funktioniert.

Das PF Flag wird auf 1 gesetzt, wenn sowohl C0 als auch C2 beide 0 oder beide 1 sind. In diesem Fall wird der nachfolgende Sprung JP (jump if PF==1) ausgeführt. Wenn wir die Werte der C3/C2/C0 in den unterschiedlichen Fällen betrachten, dann sehen wir, dass der bedingte Sprung JP in zwei Fällen ausgeführt wird: wenn \( b > a \) oder wenn \( a = b \) (das C3 Bit wird hier nicht betrachtet, da es durch den Befehl test ah, 5) gelöscht wurde).

Der Rest ist leicht nachvollziehbar. Denn der bedingte Sprung ausgeführt wurde, lädt FLD den Wert von \( _b \) nach ST(0) und wenn nicht, wird der Wert von \( _a \) dorthin geladen.

**Was ist mit der Abfrage von C2?**

Das C2 Flag wird im Fehlerfall (German text placeholder, etc.) gesetzt, aber unser Code prüft dies nicht. Wenn sich der Programmierer für FPU Fehler interessiert, muss er zusätzliche Abfragen hinzufügen.

---

106\(^{5}=101b
107https://en.wikipedia.org/wiki/Parity_flag
Erstes OllyDbg Beispiel: $a=1.2$ und $b=3.4$

Laden wir das Beispiel in OllyDbg:

Abbildung 1.68: OllyDbg: erstes FLD wurde ausgeführt

Die aktuellen Parameter der Funktion sind: $a = 1.2$ und $b = 3.4$ (Wir finden sie auf dem Stack zwei 32-Bit-Werte). $b$ (3.4) wurde bereits nach ST(0) geladen. Jetzt wird FCOMP ausgeführt. OllyDbg zeigt das zweite Argument von FCOMP, welches sich jetzt auf dem Stack befindet.
FCOMP wurde ausgeführt:

Abbildung 1.69: OllyDbg: FCOMP wurde ausgeführt

Wir sehen, dass das AX Register Nullen enthält: das passt, da alle Flag auf Null gesetzt sind. (OllyDbg disassembliert hier den Befehl `FNSTSW` als `FSTSW`—die beiden sind synonym).

Abbildung 1.70: OllyDbg: FNSTSW wurde ausgeführt
Test wurde ausgeführt:

Das PF Flag ist auf 1 gesetzt.


\textsuperscript{108}Jump Parity Even (German text placeholder)
Die Funktion beendet ihre Arbeit.
Zweites OllyDbg Beispiel: \( a=5.6 \) und \( b=-4 \)

Laden wir unser Beispiel in OllyDbg:

Abbildung 1.73: OllyDbg: erstes FLD ausgeführt

Die aktuellen Funktionsparameter sind: \( a = 5.6 \) und \( b = -4 \). \( b \) (-4) wurde bereits nach \( ST(0) \) geladen. FCOMP wird jetzt ausgeführt. OllyDbg zeigt das zweite Argument von FCOMP, welches sich jetzt auf dem Stack befindet.
Wir betrachten den Status der FPU Flas: alle null, außer C0.
Abbildung 1.75: OllyDbg: FNSTSW wird ausgeführt

Wir sehen, dass das AX Register den Wert 0x100 enthält: das C0 Flag sitzt auf dem achten Bit.
TEST wird ausgeführt:

![CPU - main thread, module d_max](image.png)

Abbildung 1.76: OllyDbg: TEST wird ausgeführt

Das PF Flag wird gelöscht. Begründung: die Anzahl der gesetzten Bits in 0x100 ist 1 und 1 ist eine ungerade Zahl. JPE wird jetzt übersprungen.
JPE wurde nicht ausgelöst, sodass FLD den Wert von \(a\) (5.6) nach ST(0) lädt:

\[
\text{Abbildung 1.77: OllyDbg: zweites FLD wurde ausgeführt}
\]

Die Funktion beendet ihre Arbeit.

**German text placeholder MSVC 2010**

---

`Listing 1.183: German text placeholder MSVC 2010`

```
a$ = 8 ; size = 8
b$ = 16 ; size = 8
d_max PROC
  fld QWORD PTR b$[esp-4]
  fld QWORD PTR a$[esp-4]

; Zustand des Stacks: ST(0) = _a, ST(1) = _b
  fcom ST(1) ; vergleiche _a und ST(1) = (_b)
  fnstsw ax
  fncle ST(0) ; lasse (_a) oben auf dem Stack
  fnstsw ax
  fstp ST(1)

; Zustand des Stacks: ST(0) = _a
  ret 0
```

---

`SLN5@d_max`

```
kopiere ST(0) nach ST(0) und hole Register vom Stack,
; lasse (_b) oben auf dem Stack
```

---

220
FCOM unterscheidet sich von FCOMP in dem Sinne, dass es nur die Werte vergleicht ohne den FPU Stack zu verändern. Anders als im vorangehenden Beispiel liegen die Operanden hier in umgekehrter Reihenfolge um vor. Dies ist der Grund warum das Ergebnis dieses Vergleichs bezüglich der C3/C2/C0 unterschiedlich ist:

- Falls \( a > b \) in unserem Beispiel, dann werden die C3/C2/C0 Bits wie folgt gesetzt: 0, 0, 0.
- Falls \( b > a \), dann ist das Bitmuster: 0, 0, 1.
- Falls \( a = b \), dann ist das Bitmuster: 1, 0, 0.

Der Befehl `test ah, 65` setzt zwei Bits — C3 und C0. Beide werden auf 0 gesetzt, falls \( a > b \): in diesem Fall wird der JNE Sprungbefehl nicht ausgeführt. Dann folgt `FSTP ST(1)` — dieser Befehl kopiert den Wert von ST(0) in den Operanden und holt einen Wert vom FPU Stack. Mit anderen Worten, der Befehl kopiert ST(0) (in dem sich gerade der Wert von \( _a \) befindet) nach ST(1). Anschließend befinden sich zwei Kopien von \( _a \) oben auf dem Stack. Nun wird ein Wert wieder vom Stack geholt. Schließlich enthält ST(0) den Wert \( _a \) und die Funktion beendet sich.

Der bedingte Sprung JNE wird in zwei Fällen ausgeführt: wenn \( b > a \) oder wenn \( a = b \). ST(0) wird nach ST(0) kopiert; dabei handelt es sich um eine Operation ohne Wirkung (`NOP`), dann wird ein Wert vom Stack geholt und in ST(0) steht dann was sich vorher in ST(1) befunden hat, nämlich \( _b \). Danach beendet sich die Funktion. Der Grund dafür, dass dieser Befehl hier erzeugt wird ist wahrscheinlich, dass die FPU über keinen anderen Befehl verfügt um einen Wert vom Stack zu holen und anschließend zu entsorgen.
Erstes OllyDbg Beispiel: $a=1.2$ und $b=3.4$

Beide FLD werden ausgeführt:

Abbildung 1.78: OllyDbg: beide FLD werden ausgeführt

FCOM wird ausgeführt: OllyDbg zeigt die Inhalte ST(0) und ST(1) übersichtlich an.
Abbildung 1.79: OllyDbg: FCOM wurde ausgeführt

C0 ist gesetzt, alle anderen Flags sind gelöscht.
FNSTSW wurde ausgeführt, AX=0x3100:

Abbildung 1.80: OllyDbg: FNSTSW wird ausgeführt
TEST wird ausgeführt:

Abbildung 1.81: OllyDbg: TEST wird ausgeführt

ZF=0, conditional bedingter Sprung wird jetzt ausgeführt.
FSTP ST (oder FSTP ST(0)) wurde ausgeführt — 1.2 wurde vom Stack geholt und 3.4 bleibt oben auf liegen:

Abbildung 1.82: OllyDbg: FSTP wird ausgeführt

Wir sehen, dass das Ergebnis des Befehls FSTP ST dem Holen eines Wertes vom FPU Stack entspricht.
Second OllyDbg example: \( a = 5.6 \) and \( b = -4 \)

Beide FLD werden ausgeführt:

Abbildung 1.83: OllyDbg: beide FLD werden ausgeführt

FCOM wird gleich ausgeführt.
Abbildung 1.84: OllyDbg: FCOM ist beendet

Alle Bedingungs-Flags sind gelöscht.
FNSTSW ist abgearbeitet, AX=0x3000:

Abbildung 1.85: OllyDbg: FNSTSW wurde ausgeführt
TEST wurde ausgeführt:

Abbildung 1.86: OllyDbg: TEST wurde ausgeführt

ZF=1, der Sprung wird jetzt nicht ausgeführt.
FSTP ST(1) wurde ausgeführt: der Wert 5.6 liegt jetzt oben auf dem FPU Stack.

Abbildung 1.87: OllyDbg: FSTP wurde ausgeführt

Wir erkennen, dass der Befehl FSTP ST(1) wie folgt funktioniert: er lässt das oberste Element des Stacks an seinem Platz, löscht aber das Register ST(1).

GCC 4.4.1

Listing 1.184: GCC 4.4.1

d_max proc near

b = qword ptr -10h
a = qword ptr -8
a_first_half = dword ptr 8
a_second_half = dword ptr 0Ch
b_first_half = dword ptr 10h
b_second_half = dword ptr 14h

push ebp
mov ebp, esp
sub esp, 10h

; lege a und b auf den lokalen Stack:

mov eax, [ebp+a_first_half]
mov dword ptr [ebp+a], eax
mov eax, [ebp+a_second_half]
mov dword ptr [ebp+a+4], eax
mov eax, [ebp+b_first_half]
mov dword ptr [ebp+b], eax

231
mov    eax, [ebp+b_second_half]
mov    dword ptr [ebp+b+4], eax

; lade a und b auf den FPU Stack:
    fld    [ebp+a]
fld    [ebp+b]

; aktueller Stand des Stacks: ST(0) - b; ST(1) - a

fxch    st(1); dieser Befehl vertauscht ST(1) und ST(0)

; aktueller Stand des Stacks: ST(0) - a; ST(1) - b

fucompp ; vergleiche a und b und nimm zwei Werte (d.h. a und b) vom
; Stack
fnstsw  ax ; speichere FPU Status in AX
sahf    ; lade SF, ZF, AF, PF, und CF Flags aus AH
setnbe  al ; speichere 1 in AL, falls CF=0 und ZF=0
test    al, al ; AL==0 ?
jz      short loc_8048453 ; ja
fld    [ebp+a]
jmp      short locret_8048456

loc_8048453:
    fld    [ebp+b]
locret_8048456:
    leave
    retn
d_max  endp

FUCOMPP ist fast wie like FCOM, nimmt aber beide Werte vom Stand und behandelt „undefinierte Zahlenwerte“ anders.

Ein wenig über undefinierte Zahlenwerte.

Die FPU ist in der Lage mit speziellen undefinierten Werten, den sogenannten not-a-number (kurz German text placeholder) umzugehen. Beispiele sind etwa der Wert unendlich, das Ergebnis einer Division durch 0, etc. Undefinierte Werte können entweder „quiet“ oder „signaling“ sein. Es ist möglich mit „quiet“ NaNs zu arbeiten, aber beim Versuch einen Befehl auf „signaling“ NaNs auszuführen, wird eine Exception geworfen.

FCOM erzeugt eine Exception, falls irgendein Operand ein German text placeholder ist. FUCOM erzeugt eine Exception nur dann, wenn ein Operand eine „signaling“ German text placeholder (SNaN) ist.

Der nächste Befehl ist SAHF (Store AH into Flags) — es handelt sich hierbei um einen seltenen Befehl, der nicht mit der FPU zusammenhängt. 8 Bits aus AH werden in die niedersten 8 Bit der CPU Flags in der folgenden Reihenfolge verschoben:

Erinnern wir uns, dass FNSTSW die für uns interessanten Bits (C3/C2/C0) auf den Stellen 6,2,0 im AH Register setzt:

Mit anderen Worten: der Befehl fnstsw ax / sahf verschiebt C3/C2/C0 nach ZF, PF und CF.

Überlegen wir uns auch die Werte der C3/C2/C0 in unterschiedlichen Szenarien:

- Falls in unserem Beispiel a größer als b ist, dann werden die C3/C2/C0 auf 0,0,0 gesetzt.
- Falls a kleiner als b ist, werden die Bits auf 0,0,1 gesetzt.
- Falls a = b, dann werden die Bits auf 1,0,0 gesetzt.

Mit anderen Worten, die folgenden Zustände der CPU Flags sind nach drei FUCOMPP/FNSTSW/SAHF Befehlen möglich:

- Falls a > b, werden die CPU Flags wie folgt gesetzt ZF=0, PF=0, CF=0.
Falls \( a < b \), werden die CPU Flags wie folgt gesetzt: \( ZF=0 \), \( PF=0 \), \( CF=1 \).

Und falls \( a = b \), dann gilt: \( ZF=1 \), \( PF=0 \), \( CF=0 \).

Abhängig von den CPU Flags und Bedingungen, speichert SETNBE entweder 1 oder 0 in AL. Es ist also quasi das Gegenstück von JNBE mit dem Unterschied, dass SETcc

Depending on the CPU flags and conditions, SETNBE stores 1 or 0 to AL. It is almost the counterpart of JNBE, with the exception that SETcc\(^{109}\) eine 1 oder 0 in AL speichert, aber Jcc tatsächlich auch springt. SETNBE speicher 1 nur, falls \( CF=0 \) und \( ZF=0 \). Wenn dies nicht der Fall ist, dann wird 0 in AL gespeichert.

Nur in einem Fall sind CF und ZF beide 0: falls \( a > b \).

In diesem Fall wird 1 in AL gespeichert, der nachfolgende JZ Sprung wird nicht ausgeführt und die Funktion liefert _a zurück. In allen anderen Fällen wird _b zurückgegeben.

German text placeholder GCC 4.4.1

<table>
<thead>
<tr>
<th>Listing 1.185: German text placeholder GCC 4.4.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>public d_max</td>
</tr>
<tr>
<td>d_max proc near</td>
</tr>
<tr>
<td>arg_0 = qword ptr 8</td>
</tr>
<tr>
<td>arg_8 = qword ptr 10h</td>
</tr>
<tr>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>fld [ebp+arg_0] ; _a</td>
</tr>
<tr>
<td>fld [ebp+arg_8] ; _b</td>
</tr>
<tr>
<td>; Zustand des Stacks: ST(0) = _b, ST(1) = _a</td>
</tr>
<tr>
<td>fxch st(1)</td>
</tr>
<tr>
<td>; Zustand des Stacks: ST(0) = _a, ST(1) = _b</td>
</tr>
<tr>
<td>fucom st(1) ; vergleiche _a und _b</td>
</tr>
<tr>
<td>fnstsw ax</td>
</tr>
<tr>
<td>sahf</td>
</tr>
<tr>
<td>ja short loc_8048448</td>
</tr>
<tr>
<td>; speichere ST(0) in ST(0) (Befehl ohne Auswirkung),</td>
</tr>
<tr>
<td>; nimm obersten Wert vom Stack,</td>
</tr>
<tr>
<td>; lasse _b oben auf</td>
</tr>
<tr>
<td>fstp st</td>
</tr>
<tr>
<td>jmp short loc_804844A</td>
</tr>
<tr>
<td>loc_8048448:</td>
</tr>
<tr>
<td>; speichere _a in ST(1), nimm obersten Wert vom Stand, lasse _a oben auf</td>
</tr>
<tr>
<td>fstp st(1)</td>
</tr>
<tr>
<td>loc_804844A:</td>
</tr>
<tr>
<td>pop ebp</td>
</tr>
<tr>
<td>retn</td>
</tr>
<tr>
<td>d_max endp</td>
</tr>
</tbody>
</table>

Dies ist fast das gleiche, außer dass ach SAHF verwendet wird. Tatsächlich prüfen die bedingte Sprungbefehle, die vorzeichenlose Zahlen auf „größer“, „kleiner“ oder „gleich“ prüfen (das sind JA, JAE, JB, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) lediglich die Flags CF und ZF.

Erinnern wir uns, an welchen Stellen die C3/C2/C0 sich im AH Register befinden, nachdem der Befehl FSTSW/FNSTSW ausgeführt wurde:

```
   6 2 1 0
C3 C2 C1 C0
```

Halten wir uns auch vor Augen wie die Bits aus AH in den CPU Flags nach der Ausführung von SAHF abgelegt werden:

\(^{109}\)cc is condition code
Nach dem Vergleich werden die C3 und C0 Bits nach ZF und CF verschoben, sodass der bedingte Sprung danach funktionieren kann. ird ausgeführt, falls sowohl CF als auch ZF gleich 0 sind.
Hierbei können alle hier aufgelisteten Sprungbefehle nach einem FNSTSW/SAHF Befehlspaar verwendet werden.

Offenbar wurden die C3/C2/C0 Status Bits der CPU dort bewusst platziert, sodass diese leicht auf die CPU Flags übertragen werden können, ohne dass zusätzliche Vertauschungen notwendig sind.

**GCC 4.8.1 mit aktivierter -O3 Optimierung**
Mit der P6 Intel Familie wurden einige neue FPU Befehle hinzugefügt. Diese sind FUCOMI (vergleiche Operanden und setze Flags der CPU) und FCMOVcc (arbeitet wie CMOVcc, aber auf FPU Registern). Offenbar haben sich die Verwalter von GCC dazu entschieden, den Support von vor-P6 Intel CPUs (frühe Pentiums, 80486, etc.) einzustellen.
Außerdem ist die FPU nicht länger eine separate Einheit in der P6 Intel Familie, sodass es nun auch möglich ist, die Flags der CPU von der FPU aus zu prüfen oder zu verändern.

Wir erhalten also das Folgende:

**Listing 1.186:** German text placeholder GCC 4.8.1

```assembly
fld QWORD PTR [esp+4] ; lade "a"
fld QWORD PTR [esp+12] ; lade "b"
fxch st(1)
; ST0=a, ST1=b
fucomi st, st(1)
; vergleiche "a" und "b"
fcmovbe st, st(1)
; kopiere ST1 (hier: "b") nach ST0, falls a<=b
; lasse "a" sonst in ST0
fungve st, st(1)
; verwirf den Wert in ST1
fstp st(1)
ret
```

Schwer zu sagen, warum FXCH (vertausche Operanden) hier verwendet wird.
Es ist möglich, diesen Befehl loszuwerden, indem man die ersten beiden FLD Befehle vertauscht oder FCMOVBEC (below or equal) durch FCMOVA (above) ersetzt. Wahrscheinlich handelt es sich hierbei um eine Ungenauigkeit im Compiler.
FUCOMI vergleicht also ST(0) (a) und ST(1) (b) und setzt einige Flags in der CPU. FCMOVBE prüft die Flags und kopiert ST(1) (in diesem Moment also b) nach ST(0) (hier: a), falls ST0(a) <= ST1(b). Andernfalls (a > b) wird a in ST(0) belassen.
Der letzte FSTP Befehl belässt ST(0) oben auf dem Stack und verwirft den Inhalt von ST(1).
Verfolgen wir den Funktionsverlauf in GDB:

**Listing 1.187:** German text placeholder GCC 4.8.1 and GDB

```bash
./d_max
$(gdb) b d_max
Breakpoint 1 at 0x80484a0
$(gdb) run
Starting program: /home/dennis/polygon/d_max
$(gdb) f u d_max
$(gdb) ni
Breakpoint 1, 0x80484a0 in d_max ()
$(gdb) f u
$(gdb) disas $eip
```

110Beginnend mit Pentium Pro, Pentium-II, etc.
0x080484a0 <+0>:     fldl  0x4(%esp)
0x080484a4 <+4>:     fldl  0xc(%esp)
0x080484a8 <+8>:     fxch   %st(1)
0x080484aa <+10>:   fucomi  %st(1),%st
0x080484ac <+12>:   fcmovbe  %st(1),%st
0x080484ae <+14>:   fstp   %st(1)
0x080484b0 <+16>:   ret

End of assembler dump.

(gdb) ni
0x080484a8 in d_max ()
(gdb) info float
R7: Valid 0x3fff9999999999999800 +1.199999999999999956
R6: Valid 0x40000d999999999999800 +3.399999999999999911
R5: Empty 0x00000000000000000000
R4: Empty 0x00000000000000000000
R3: Empty 0x00000000000000000000
R2: Empty 0x00000000000000000000
R1: Empty 0x00000000000000000000
R0: Empty 0x00000000000000000000

Status Word:  0x3000
TOP:  6
Control Word: 0x037f  IM DM ZM OM UM PM
PC: Extended Precision (64-bits)
RC: Round to nearest
Tag Word: 0x0fff
Instruction Pointer: 0x73:0x080484a4
Operand Pointer: 0x7b:0xbfffff118
Opcode: 0x0000

(gdb) ni
0x080484ac in d_max ()
(gdb) info registers
eax  0x1  1
ecx  0xbfffff1c4  -1073745468
ddx  0x8048340  134513472
ebx  0xb7fbf000  -1208225792
esp  0xbfffff10c  0xbfffff10c
ebp  0xbfffff128  0xbfffff128
esi  0x0  0

End of assembler dump.
Unter Verwendung von „ni“ führen wir die ersten beiden FLD Befehle aus.
Sehen wir uns die FPU Register (Zeile 33) an.

Wie bereits erwähnt, bildet der FPU Registersatz einen Ringpuffer und keinen Stack (1.17.5 on page 202). Außerdem zeigt GDB nicht die STx Register, sondern die internen FPU Register (Rx). Der Pfeil (in Zeile 35) zeigt auf das aktuell obere Ende des Stacks.

Wir sehen auch den Inhalt des T0P Registers in Status Word (Zeile 36-37)–hier ist dieser 6, sodass das oberste Element im Stack also aktuell auf das interne Register 6 zeigt.

Die Werte von $a$ und $b$ werden nach Ausführung von FXCH (Zeile 54) vertauscht.

FCOMI wird ausgeführt (Zeile 83). Betrachten wir die Flags: CF ist gesetzt (Zeile 95).

FCMOVBE hat den Wert von $b$ kopiert (siehe Zeile 104).

FSTP lässt einen Wert oben auf dem Stack (Zeile 139). Der Wert von T0P beträgt jetzt 7, was bedeutet, dass das obere Ende des FPU Stacks jetzt auf das interne Register 7 zeigt.

**ARM**

**German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**

### Listing 1.188: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)

<table>
<thead>
<tr>
<th>Befehl</th>
<th>Register 1</th>
<th>Register 2</th>
<th>Register 3</th>
<th>Anmerkung</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV</td>
<td>D16, R2, R3</td>
<td>$b$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VMOV</td>
<td>D17, R0, R1</td>
<td>$a$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCMPE.F64</td>
<td>D17, D16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VMRS</td>
<td>AP SR_nzcv, FPSCR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VM0 VGT.F64</td>
<td>D16, D17</td>
<td></td>
<td></td>
<td>kopiere &quot;$a&quot; nach D16</td>
</tr>
<tr>
<td>VMOV</td>
<td>R0, R1, D16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>LR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ein recht einfacher Fall. Die Eingabewerte werden in die Register D17 und D16 geladen und dann mit dem Befehl VCMPE verglichen.

Genau wie der x86-Koprozessor besitzt auch der ARM-Koprozessor seine eigenen Status und Flag Register (FPSCR\(^{111}\)), denn es gibt auch hier die Notwendigkeit die spezifischen Flags des Koprozessors zu speichern.

Und genau wie beim x86 gibt es auch in ARM keine Befehle für bedingte Sprünge, die die Bits im Statusregister des Koprozessors abfragen können. So gibt es den Befehl VMRS, um 4 Bits (N, Z, C, V) vom Statusregister des Koprozessors in das allgemeine Statusregister (APSR\(^{112}\)) zu kopieren.

VMOVGT ist das Analogon zum MOVGT Befehl für D-Register: er wird ausgeführt, wenn ein Operand bezüglich eines GT—Greater Than (dt. größer als) Vergleichs größer ist als der andere.

Wenn er ausgeführt wird, wird der Wert von $a$ nach D16 geschrieben (dieser wird aktuell in D17 gespeichert). Andernfalls bleibt der Wert von $b$ im D16 Register.

Der vorletzte Befehl VMOV bereitet den Wert im D16 Register für die Rückgabe über das Registerpaar R0 und R1 vor.

**German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**

### Listing 1.189: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)

<table>
<thead>
<tr>
<th>Befehl</th>
<th>Register 1</th>
<th>Register 2</th>
<th>Register 3</th>
<th>Anmerkung</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV</td>
<td>D16, R2, R3</td>
<td>$b$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VMOV</td>
<td>D17, R0, R1</td>
<td>$a$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCMPE.F64</td>
<td>D17, D16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VMRS</td>
<td>AP SR_nzcv, FPSCR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT GT</td>
<td>D16, D17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VM0 VGT.F64</td>
<td>R0, R1, D16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>LR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^{111}\)(ARM) Floating-Point Status and Control Register

\(^{112}\)(ARM) Application Program Status Register

Thumb-2 wurde erweitert, um zu ermöglichen alten Thumb-Befehlen nachträglich Prädikate zuzuweisen. Hier, im von IDA erzeugten Listing finden wir den VM0VGT Befehl aus dem vorherigen Beispiel wieder.

Tatsächlich ist hier das gewöhnliche VM0 kodiert, aber IDA ergänzt den Suffix -GT, da sich direkt davor eine IT GT Befehl befindet.

Der IT Befehl definiert einen sogenannten If-Then-Block.

Nach dem Befehl können bis zu 4 weitere Befehle, jeder mit einem beschreibenden Suffix, verwendet werden. In unserem Beispiel impliziert IT GT, dass der Folgebefehl genau dann ausgeführt werden soll, wenn die ITGT (Greater Than) Bedingung wahr ist.

Hier ist ein komplexeres Codefragment, welches aus Angry Birds (für iOS) stammt:

<table>
<thead>
<tr>
<th>Listing 1.190: Angry Birds Classic</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>ITE NE</td>
</tr>
<tr>
<td>VMOVNE</td>
</tr>
<tr>
<td>VMOVEQ</td>
</tr>
<tr>
<td>BLX</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

ITE steht für if-then-else und kodiert Suffixe für die beiden folgenden Befehle.

Der erste Befehl wird ausgeführt, wenn die durch ITE (NE, not ewual, dt. ungleich) kodierte Bedingung wahr ist und der zweite wenn die Bedingung falsch ist (die inverse Bedingung zu NE ist EQ (equal, dt. gleich)).

Der dem zweiten Befehl folgende VM0V (oder VM0VEQ) ist ein gewöhnlicher Befehl ohne Suffix (BLX).

Ein weiteres etwas schwieriger verständliches Codefragment, ebenfalls aus Angry Birds:

<table>
<thead>
<tr>
<th>Listing 1.191: Angry Birds Classic</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>ITTTT EQ</td>
</tr>
<tr>
<td>MOVEQ</td>
</tr>
<tr>
<td>ADDEQ</td>
</tr>
<tr>
<td>POPEQ.W</td>
</tr>
<tr>
<td>POPEQ</td>
</tr>
<tr>
<td>BLX</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Vier „T“ Symbole in der Mnemonik des Befehls bedeuten, dass die vier folgenden Befehle alle ausgeführt werden, wenn die Bedingung wahr ist.

Aus diesem Grund fügt IDA den -EQ Suffix an jeden der vier Befehle an.

Gäbe es beispielsweise ITEEE EQ (if-then-else-else-else), dann würden wie folgt Suffixe angehängt werden:

-EQ
-NE
-NE
-NE

Ein weiteres Fragment aus Angry Birds:

<table>
<thead>
<tr>
<th>Listing 1.192: Angry Birds Classic</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>CMP.W</td>
</tr>
</tbody>
</table>

238
ITTE LE
SUBLE.W R10, R0, #1
NEGLE R0, R0
MOVG R10, R0
MOVS R6, #0 ; ohne Suffix
CBZ R0, loc_1E7E32 ; ohne Suffix
...

ITTE (if-then-then-else) impliziert, dass der erste und zweite Befehl ausgeführt werden, wenn die LE (Less or Equal, dt. mindestens) Bedingung wahr ist und der dritte, wenn die inverse Bedingung (GT—Greater Than, dt. mehr als) wahr ist.

Für gewöhnlich erzeugen Compiler nicht alle denkbaren Kombinationen. Im betrachteten Spiel Angry Birds beispielsweise (classic Version für iOS) werden nur die folgenden Variationen des IT Befehls verwendet: IT, ITE, ITT, ITTE, ITTT, ITTTT. Bleibt die Frage, wie man dies lernen kann. In IDA ist es mögliche Listing-Dateien zu erzeugen mit der Option 4 Bytes für jeden Opcode anzuzeigen. Dadurch können wir bei Kenntnis des höherwertigen Teils des 16-Bit-Opcodes (IT entspricht 0xBF) unter Zuhilfenahme von grep wie folgt vorgehen:

```bash
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

Übrigens, wenn man von Hand Assemblerprogramme für ARM in Thumb-2 mode schreibt und man die Suffixe für die Bedingungen selbst anhängt, wird der Assemblierer die entsprechenden IT Befehle inklusive der benötigten Flags automatisch an den benötigten Stellen hinzufügen.

**German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**

```
Listing 1.193: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)
```

b = -0x20
da = -0x18
val_to_return = -0x10
saved_R7 = -4

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR</td>
<td>R7, [SP,#saved_R7]!</td>
</tr>
<tr>
<td>MOV</td>
<td>R7, SP</td>
</tr>
<tr>
<td>SUB</td>
<td>SP, SP, #0x1C</td>
</tr>
<tr>
<td>BIC</td>
<td>SP, SP, #7</td>
</tr>
<tr>
<td>VM0V</td>
<td>D16, R2, R3</td>
</tr>
<tr>
<td>VM0V</td>
<td>D17, R0, R1</td>
</tr>
<tr>
<td>VSTR</td>
<td>D17, [SP,#0x20+a]</td>
</tr>
<tr>
<td>VSTR</td>
<td>D16, [SP,#0x20+b]</td>
</tr>
<tr>
<td>VLDR</td>
<td>D16, [SP,#0x20+a]</td>
</tr>
<tr>
<td>VLDR</td>
<td>D17, [SP,#0x20+b]</td>
</tr>
<tr>
<td>VCMPE.F64</td>
<td>D16, D17</td>
</tr>
<tr>
<td>VMRS</td>
<td>APSR_nzcv, FPSCR</td>
</tr>
<tr>
<td>BLE</td>
<td>loc_2E08</td>
</tr>
<tr>
<td>VLDK</td>
<td>D16, [SP,#0x20+a]</td>
</tr>
<tr>
<td>VSTR</td>
<td>D16, [SP,#0x20+val_to_return]</td>
</tr>
<tr>
<td>B</td>
<td>loc_2E10</td>
</tr>
</tbody>
</table>

loc_2E08 | VLDK    | D16, [SP,#0x20+a] |
| VSTR    | D16, [SP,#0x20+val_to_return] |
loc_2E10 | VLDK    | D16, [SP,#0x20+val_to_return] |
| VM0V    | R0, R1, D16 |
| MOV     | SP, R7     |
| LDR     | R7, [SP+0x20+b],#4 |
| BX      | LR         |

Fast identisch mit dem, was wir schon gesehen haben, aber hier gibt es zu viel redundanten Code, weil die Variablen a und b im lokalen Stack und zusätzlich als Rückgabewerte gespeichert werden.
German text placeholder Keil 6/2013 (German text placeholder)

Listing 1.194: German text placeholder Keil 6/2013 (German text placeholder)

```
PUSH {R3-R7,LR)
MOV S R4, R2
MOV S R5, R3
MOVE R6, R0
MOV S R7, R1
BL __aeabi_cdrcmple
BCS loc_1C0
MOV S R0, R6
MOV S R1, R7
PO P {R3-R7,PC}

loc_1C0
MOVS R0, R4
MOVS R1, R5
PO P {R3-R7,PC}
```

Keil erzeugt keine FPU-Befehle, da er sich das darauf verlassen kann, dass diese von der Ziel-CPU unterstützt werden und dies nicht durch einfache bitweisen Vergleich erledigt werden kann.

Keil ruft also eine Funktion einer externen Programmbibliothek (__aeabi_cdrcmple) auf, um den Vergleich durchzuführen. Das Ergebnis des Vergleichs wird von der Funktion in den Flags belassen, sodass der folgende BCS (Carry set—Greater than or equal) Befehl ohne zusätzlichen Code funktioniert.

**ARM64**

German text placeholder GCC (Linaro) 4.9

```
d_max:
; D0 - a, D1 - b
   fcmpe d0, d1
   fcsel d0, d0, d1, gt
; Ergebnis jetzt in D0
   ret
```

Der ARM64 ISA verfügt über FPU-Befehle, die der Einfachheit halber die Flags der CPU APSR anstelle von FPSCR setzen. Die FPU ist hier kein separates Gerät mehr (zumindest logisch).

Wir finden hier FCMPE. Er vergleicht die beiden über D0 und D1 übergebenen Werte (dabei handelt es sich um das erste und zweite Argument der Funktion) und setzt APSR die Flags (N, Z, C, V).

FCSEL (Floating Conditional Select) kopiert den Wert von D0 oder D1 nach D0, abhängig von der Bedingung (GT—Greater Than), und verwendet wiederum Flags im APSR Register anstatt derer von FPSCR.

Dies ist im Vergleich zum Befehlssatz alter CPUs deutlich praktischer.

 Falls die Bedingung wahr ist (GT), dann wird der Wert von D0 nach D0 kopiert (d.h. es geschieht nichts). Falls die Bedingung falsch ist, wird der Wert von D1 nach D0 kopiert.

**German text placeholder GCC (Linaro) 4.9**

```
d_max:
; speichere Eingabeparameter in der "Register Save Area"
   sub sp, sp, #16
   str d0, [sp,8]
   str d1, [sp]
; lade Werte erneut
   ldr x1, [sp,8]
   ldr x0, [sp]
   fmov d0, x1
   fmov d1, x0
; D0 - a, D1 - b
   fcmpe d0, d1
```
Der nicht optimierende GCC ist weniger kompakt.
Zunächst speichert die Funktion ihre Eingabewerte auf dem lokalen Stack (*Register Save Area*), danach lädt der Code die Werte erneut in die Register X0/X1 und kopiert sie schließlich nach D0/D1, um sie mittels FCMPE zu vergleichen. Eine Menge redundanten Code, aber so arbeitet ein nicht optimierender Compiler nun einmal. FCMPE vergleich die Werte und setzt die APSR Flags. Zu diesem Zeitpunkt entscheidet sich der Compiler noch nicht für den praktischeren FCSEL Befehl und arbeitet stattdessen mit herkömmlichen Methoden: er verwendet den BLE Befehl (*Branch if Less than or Equal*). Im ersten Fall (a > b) wird der Wert von a nach X0 geladen. Im anderen Fall (a <= b) wird der Wert von b nach X0 geladen. Schließlich wird der Wert aus X0 nach D0 kopiert, denn der Rückgabewert muss sich in diesem Register befinden.

**German text placeholder**
Dem Leser bleibt als Übung, den vorstehenden Code zu optimieren, indem manuell die redundanten Instruktionen entfernt werden ohne dabei neue einzuführen (wie etwa FCSEL).

**German text placeholder GCC (Linaro) 4.9—float**
Wir wollen nun dieses Beispiel umschreiben, indem wir float anstelle von double verwenden.

```plaintext
float f_max (float a, float b) {
   if (a>b)
      return a;
   return b;
};
```

Es ist der gleiche Code, aber hier werden die S-Register anstelle der D-Register verwendet. Das ist darauf zurückzuführen, dass der float Typ in 32-Bit-S-Registern übergeben wird (welche in Wirklichkeit nichts anderes als die niederen Teile der 64-Bit-D-Register sind).

**MIPS**
Der Koprozessor des MIPS Prozessors hat ein Condition Bit, welches in der FPU gesetzt und in der CPU geprüft werden kann.
Frühere MIPS haben nur ein Condition Bit (genannt FCC0), spätere haben deren 8 (genannt FCC7-FCC0). Diese(s) Bit(s) befinden sich im Register FCCR.

Listing 1.195: German text placeholder GCC 4.4.5 (IDA)
; springe zu locret_14 , falls das Condition Bit gesetzt ist
    bclt locret_14
; dieser Befehl wird stets ausgeführt (setze Rückgabewert auf "a"):
    mov.d $f0, $f12 ; branch delay slot
; dieser Befehl wird nur ausgeführt, falls der Zweig nicht betreten wurde (d.h.,
; falls b>=a); setze Rückgabewert auf "b":
    mov.d $f0, $f14
locret_14:
    jr $ra
    or $at, $zero ; branch delay slot, NOP

C.LT.D vergleicht zwei Werte. LT ist die Bedingung „Less Than“ (weniger als). D impliziert einen Wert vom Typ double. Abhängig vom Ergebnis des Vergleichs wird das FCC0 Condition Bit entweder gesetzt oder gelöscht.

BC1T prüft das FCC0 Bit und springt, falls das Bit gesetzt ist. T bedeutet, dass der Sprung ausgeführt wird, wenn das Bit gesetzt („True“) ist. Daneben gibt es auch den Befehl BC1F, der springt, wenn das Bit gelöscht („FALSE“) ist.

Abhängig vom Sprung wird einer der Funktionsargument in $F0 abgelegt.

### 1.17.8 Einige Konstanten

Es ist leicht, in Wikipedia die Darstellungen einiger Konstanten nach dem IEEE 754 Standard nachzulesen. Es ist interessant zu wissen, dass 0.0 nach IEEE 754 als 32 Nullbits (in einfacher Genauigkeit) oder 64 Nullbits (in doppelter Genauigkeit) dargestellt wird. Wenn also eine Fließkommavariable im Register oder Speicher auf 0.0 gesetzt werden soll, kann der Befehl MOV oder XOR reg, reg verwendet werden. Dies ist geeignet für Strukturen, in denen viele Variable unterschiedlichster Datentypen vorhanden sind. Mit der gewöhnlichen memset() Funktion kann man alle Integervariablen auf 0, alle Booleschen Variablen auf false, alle Pointer auf NULL und alle Fließkommavariablen (beliebiger Genauigkeit) auf 0.0 setzen.

### 1.17.9 Kopieren

Man könnte fälschlicherweise annehmen, dass die Befehle FLD/FST verwendet werden müssen, um IEEE 754 Werte zu laden oder zu speichern (also auch zu kopieren). Nichtsdestotrotz kann dies einfacher durch den Befehl MOV erreicht werden, welcher, logischerweise, Werte bitweise kopiert.

### 1.17.10 Stack, Taschenrechner und umgekehrte polnische Notation

Jetzt können wir einsehen, warum manche alten Taschenrechner die umgekehrte polnische Notation verwenden. Für die Addition von 12 und 34 muss beispielsweise zuerst 12, dann 34 und dann das „plus“-Zeichen eingegeben werden.

Dies liegt daran, dass alte Taschenrechner als einfache Stackmaschinen implementiert waren und es auf diese Weise wesentlich einfacher war, mit komplexen geklammerten Ausdrücken umzugehen.

### 1.17.11 x64

Mehr dazu wie Fließkommazahlen in x86-64 verarbeitet werden unter: 1.26 on page 388

### 1.17.12 Übungen

- [http://challenges.re/60](http://challenges.re/60)
- [http://challenges.re/61](http://challenges.re/61)

### 1.18 Arrays

German text placeholder
1.18.1

```c
#include <stdio.h>

int main()
{
    int a[20];
    int i;
    for (i=0; i<20; i++)
        a[i]=i*2;
    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);
    return 0;
}
```

x86

MSVC

Kompilieren wir das Beispiel:

Listing 1.196: MSVC 2008

```assembly
_TEXT SEGMENT
_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 84 ; 00000054H
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN6@main
$LN5@main:
    mov eax, DWORD PTR _i$[ebp]
    add eax, 1
    mov DWORD PTR _a$[ebp], eax
$LN6@main:
    cmp DWORD PTR _i$[ebp], 20 ; 00000014H
    jge SHORT $LN4@main
    mov ecx, DWORD PTR _i$[ebp]
    shl ecx, 1
    mov edx, DWORD PTR _i$[ebp]
    mov DWORD PTR _a$[ebp+edx*4], ecx
    jmp SHORT $LN5@main
$LN4@main:
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN3@main
$LN2@main:
    mov eax, DWORD PTR _i$[ebp]
    add eax, 1
    mov DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp DWORD PTR _i$[ebp], 20 ; 00000014H
    jge SHORT $LN1@main
    mov ecx, DWORD PTR _i$[ebp]
    mov edx, DWORD PTR _a$[ebp+ecx*4]
    push edx
    mov eax, DWORD PTR _a$[ebp]
    push eax
    push OFFSET $SG2463
    call _printf
    add esp, 12 ; 0000000cH
    jmp SHORT $LN2@main
$LN1@main:
    xor eax, eax
    mov esp, ebp
    pop ebp
    mov eax, 1
    ret
```
Soweit nichts Außergewöhnliches, nur zwei Schleifen: die erste füllt mit Werten auf und die zweite gibt Werte aus. Der Befehl `shl ecx, 1` wird für die Multiplikation mit 2 in ECX verwendet; mehr dazu unten 1.16.2 on page 194.

Auf dem Stack werden 80 Bytes für das Array reserviert: 20 Elemente von je 4 Byte.
Untersuchen wir dieses Beispiel in OllyDbg.
Wir erkennen wie das Array befüllt wird:

jedes Element ist ein 32-Bit-Wort vom Typ `int` und der Wert ist der Index multipliziert mit 2:

Abbildung 1.88: OllyDbg: nach dem Füllen des Arrays

Da sich dieses Array auf dem Stack befindet, finden wir dort alle seine 20 Elemente.

GCC

Hier ist was GCC 4.4.1 erzeugt:

Listing 1.197: GCC 4.4.1
Die Variable `a` ist übrigens vom Typ `int*` (Pointer auf `int`)–man kann einen Pointer auf ein Array an eine andere Funktion übergeben, aber es ist richtiger zu sagen, dass der Pointer auf das erste Element des Arrays übergeben wird. (Die Adressen der übrigen Elemente werden in bekannter Weise berechnet.)

Wenn man diesen Pointer mittels `a[idx]` indiziert, wird `idx` zum Pointer addiert und das dort abgelegte Element (auf das der berechnete Pointer zeigt) wird zurückgegeben.


Das ist der Grund warum es es möglich ist, Dinge wie „`string`[i]“ zu schreiben–es handelt sich dabei um einen korrekten C/C++ Ausdruck!

**ARM**

**German text placeholder Keil 6/2013 (German text placeholder)**

```asm
EXPORT _main

_main

STMFD SP!, {R4,LR} ; Platz für 20 int Variablen reservieren
SUB SP, SP, #0x50

; erste Schleife

MOV R4, #0 ; i
B loc_4A0

loc_494

MOV R0, R4,LSL#1 ; R0=R4*2
STR R0, [SP,R4,LSL#2] ; sichere R0 in SP+R4<<2 (entspricht SP+R4*4)
ADD R4, R4, #1 ; i=i+1

loc_4A0
```
CMP R4, #20 ; i<20?
BLT loc_494 ; falls ja, Rumpf der Schleife erneut ausführen

; zweite Schleife
MOV R4, #0 ; i
B loc_4C4
loc_4B0
LDR R2, [SP, R4, LSL#2] ; (zweites printf Argument) R2=*(SP+R4<<4) (entspricht *(SP+R4*4))
MOV R1, R4 ; (erstes printf Argument) R1=i
ADR R0, aADD ; "a[%d]=%d\n"
BL __2printf
ADD R4, R4, #1 ; i=i+1
loc_4C4
CMP R4, #20 ; i<20?
BLT loc_4B0 ; falls ja, Rumpf der Schleife erneut ausführen
MOV R0, #0 ; Rückgabewert
ADD SP, SP, #0x50 ; Block freigeben, der für die 20 int Variablen reserviert wurde
LDMFD SP!, {R4, PC}

Der Typ int benötigt 32 Bit (oder 4 Byte) zum Speichern, weshalb zum Speichern von 20 int Variablen 80 (0x50) Bytes benötigt werden. Deshalb reserviert der Befehl SUB SP, SP, #0x50 im Funktionsprolog genau diese Menge an Speicherplatz auf dem Stack.

In sowohl der ersten als auch der zweiten Scheife befindet sich der Scheifenzähler i im R4 Register.

Die Zahl, die in das Array geschrieben wird, wird über den Ausdruck i·2 berechnet, was äquivalent zur Linksverschiebung um 1 Bit ist, weshalb der Befehl MOV R0, R4, LSL#1 verwendet wird.

STR R0, [R5, R2] ; sichere R1 in *(R5+R2) (entspricht R5+i*4) bei <20, führe Schleifenrumpf erneut aus

Ein Pointer auf ein Arrayelement wird wie folgt berechnet: SP zeigt auf den Beginn des Arrays, Reg4 ist i. Eine Linksverschiebung von i um 2 Bits entspricht effektiv einer Multiplikation mit 4 (da jedes Arrayelement eine Größe von 4 Byte hat) und wird dann zur Adresse am Beginn des Arrays addiert.

Die zweite Schleife enthält den inversen Befehl LDR R2, [SP, R4, LSL#2]. Er lädt den benötigten Wert aus dem Array und der Pointer hierauf wird analog berechnet.

German text placeholder Keil 6/2013 (German text placeholder)

_main
PUSH {R4, R5, LR}
; Platz für 20 int Variablen und eine weitere Variable reservieren
SUB SP, SP, #0x54

; erste Schleife
MOVS R0, #0 ; i
MOV R5, SP ; Pointer auf das erste Arrayelement
loc_1CE
LSLS R1, R0, #1 ; R1=i<<1 (entspricht i*2)
LSLS R2, R0, #2 ; R2=i<<2 (entspricht i*4)
ADDS R0, R0, #1 ; i=i+1
CMP R0, #20 ; i<20?
STR R1, [R5, R2] ; sichere R1 in *(R5+R2) (entspricht R5+i*4)
BLT loc_1CE ; falls i<20, führe Schleifenrumpf erneut aus

; zweite Schleife
MOVS R4, #0 ; i=0
loc_1DC
LSLS R0, R4, #2 ; R0=i<<2 (entspricht i*4)
LDR R2, [R5, R0] ; lade von *(R5+R0) (entspricht R5+i*4)
MOVS R1, R4
ADR R0, aADD ; "a[%d]=%d\n"
BL __2printf
Der Thumb Code ist sehr ähnlich. Der Thumb mode kennt spezielle Befehl für das bitweise Verschieben (wie LSLS), der den in das Array zu schreibenden Wert und die Adresse jedes Elements im Array berechnet.

Der Compiler reserviert ein wenig mehr Platz auf dem lokalen Stack, aber die letzten 4 Byte davon werden nicht verwendet.

**German text placeholder GCC 4.9.1 (ARM64)**

```
.LC0:
    .string "a[%d]=%d\n"

main:
    ; speichere FP und LR im Stackframe:
    stp    x29, x30, [sp, -112]!
    ; setze Stackframe (FP=SP)
    add    x29, sp, 0
    ; initialisiere Zählervariable mit 0 (WZR ist das Register, das immer 0 enthält):
    str    wzr, [x29,108]
    ; springe zum Code, der die Schleifenbedingung prüft:
    b      .L2

.L2:
    ; prüfen, ob wir fertig sind:
    ldr    w0, [x29,108]
    cmp    w0, 19
    ; springe zu L3 (Beginn des Schleifenrumpfes) falls nicht:
    bne    .L3
    ; der zweite Teil der Funktion fängt hier an.
    ; initialisiere Zählervariable mit 0
    ; es wurde übrigens derselbe Platz im lokalen Stack für den Zähler verwendet,
    ; da dieselbe lokale Variable (i) als Zähler verwendet wird.
    str    wzr, [x29,108]
    b      .L4

.L3:
    ; berechne Arrayadresse:
    add    x0, x29, 24
    ; berechne "i" Wert:
    ldrsw  x1, [x29,108]
    ; lade Wert aus dem Array von der Adresse (X0+X1<<2 = Adresse des Arrays + i*4)
    ldr    w2, [x0,x1,lsl 2]
    ; berechne Adresse des "a[%d]=%d\n" Strings:
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
    ; lade Variable "i" nach W1 und übergebe sie an printf() als zweites Argument:
    ldr    w1, [x29,108]
    ; W2 enthält immer noch den Wert des gerade geladenen Arrayelement.
```

ADD  R4, R4, #1 ; i=i+1
CMP  R4, #20 ; < 20?
BLT   loc_1DC ; falls < 20, führe Schleifenrumpf erneut aus
MOVS R0, #0 ; Rückgabewert

; Block freigeben, der für die 20 int Variablen und die weitere Variable reserviert wurde
ADD  SP, SP, #0x54
POP  {R4,R5,PC}
Die Funktion verwendet eine Menge S-Register, die gesichert werden müssen. Das ist der Grund dafür, dass die Werte im Funktionsprolog gespeichert und im Funktionsepilog wiederhergestellt werden.

Listing 1.199: German text placeholder GCC 4.4.5 (IDA)

```c
main:

var_70 = -0x70
var_68 = -0x68
var_14 = -0x14
var_10 = -0x10
var_C = -0xC
var_8 = -8
var_4 = -4

; Funktionsprolog:
lui $gp, (__gnu_local_gp >> 16)
addiu $sp, -$0x80
la $gp, (__gnu_local_gp & 0xFFFF)
sw $ra, 0x80+var_4($sp)
sw $s3, 0x80+var_8($sp)
sw $s2, 0x80+var_C($sp)
sw $s1, 0x80+var_10($sp)
sw $s0, 0x80+var_14($sp)
sw $gp, 0x80+var_70($sp)
addiu $s1, $sp, 0x80+var_68
move $v1, $s1
move $v0, $zero

; dieser Wert wird als terminierendes Zeichen für die Schleife verwendet.
; er wurde vom GCC Compiler zur Compilerzeit vorausberechnet
li $a0, 0x28 # '('

loc_34: # CODE XREF: main+3C

; speichere Wert:
sw $v0, 0($v1)
; erhöhe zu speichernden Wert bei jeder Iteration um 2
addiu $v0, 2
; terminierendes Zeichen erreicht?
bne $v0, $a0, loc_34
; immer 4 zur Adresse addieren:
addiu $v1, 4
; Schleife zum Befüllen des Arrays ist beendet
; Beginn der zweiten Schleife
la $s3, $.LC0 # "a[%d]=%d\n"
; Variable "i" bleibt in $s0:
move $s0, $zero
li $s2, 0x14

loc_54: # CODE XREF: main+70

; Aufruf von printf():

bne $v0, $a0, loc_34
```
lw   $t9, (printf & 0xFFFF)($gp)
lw   $a2, 0($s1)
move  $a1, $s0
move  $a0, $s3
jalr  $t9
; erhöhe "i":
addiu  $s0, 1
lw    $gp, 0x80+var_70($sp)
; springe zum Rumpf der Schleife, falls das Ende noch nicht erreicht ist
bne   $s0, $s2, loc_54
; setze Memory Pointer auf das nächste 32-Bit-Wort:
addiu  $s1, 4
; Funktionsepilog
lw    $ra, 0x80+var_4($sp)
move  $v0, $zero
lw    $s3, 0x80+var_8($sp)
lw    $s2, 0x80+var_C($sp)
lw    $s1, 0x80+var_10($sp)
lw    $s0, 0x80+var_14($sp)
jr    $ra
addiu  $sp, 0x80

$LC0: .ascii "a[%d]=%d\n"<0>  # DATA XREF: main+44

Interessant: es gibt zwei Schleifen und die erste benötigt $i$ nicht; sie benötigt nur $i \cdot 2$ (erhöht um 2 bei jedem Iterationsschritt) und die Adresse im Speicher (erhöht um 4 bei jedem Iterationsschritt).
Wir sehen hier also zwei Variablen: eine (in $V0), die jedes Mal um 2 erhöht wird, und eine andere (in $V1), die um 4 erhöht wird.
Die zweite Schleife ist der Ort, an dem printf() aufgerufen wird und dem Benutzer den Wert von $i$ zurückliefert, es gibt also eine Variable die in $S0$ inkrementiert wird und eine Speicheradresse in $S1$, die jedes Mal um 4 erhöht wird.
Das erinnert uns an die Optimierung von Schleifen, die wir früher betrachtet haben: ?? on page ??.
Das Ziel der Optimierung ist es, die Multiplikationen loszuwerden.

1.18.2 Puffer-Überlauf

Lesezugriff außerhalb von Arraygrenzen

Der indizierte Zugriff auf ein Array wird durch array[index] realisiert. Wenn man sich den erzeugten Code genau ansieht, bemerkt man, dass eine Prüfung der Indexgrenzen fehlt, welche die Bedingung kleiner als 20 validiert. Was also passiert, wenn der Index 20 oder größer ist? Hier haben wir es mit einem unschönen Feature von C/C++ zu tun.
Hier ein Beispielcode der erfolgreich kompiliert wurde und funktioniert:

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;
    for (i=0; i<20; i++)
        a[i]=i*2;
    printf("a[20]=%d\n", a[20]);
    return 0;
}
```

Ergebnis des Kompiliervorgangs (MSVC 2008):

```
Listing 1.200: German text placeholder MSVC 2008
$SG2474  DB    a[20]=%d', 0AH, 00H
i$ = -84; size = 4
```
Der Code produziert dieses Ergebnis:

Listing 1.201: OllyDbg: console output

a[20]=1638280

Es handelt sich um irgendetwas, das auf dem Stack in der Nähe des Arrays gelegen hat, 80 Byte von dessen erstem Element entfernt.
Versuchen wir mit OllyDbg herauszufinden, woher dieser Wert kommt.
Laden und finden wir also den Wert, der sich direkt hinter dem letzten Arrayelement befindet:

Abbildung 1.89: OllyDbg: das 20. Element lesen und printf() ausführen

Worum handelt es sich? Dem Stacklayout nach zu urteilen ist dies der gespeicherte Wert des EBP Registers.
Verfolgen wir das ganze weiter und schauen uns an, wie dieser wiederhergestellt wird:

Wie könnte es anders gelöst werden? Der Compiler könnte zusätzlichen Code erzeugen, der sicherstellt, dass der Index sich stets innerhalb der Arraygrenzen befindet (wie in höheren Programmiersprachen\(^{113}\)), aber das würde den Code langsamer machen.

**Schreibzugriff außerhalb von Arraygrenzen**

Nehmen wir an, wir hätten ein paar Werte illegalerweise vom Stack gelesen, wie könnten wir etwas hineinschreiben?

Hier ist, was wir haben:

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<30; i++)
    {
        a[i]=i;
    }
    return 0;
}
```

\(^{113}\)Java, Python, etc.
Wir erhalten das Folgende:

Listing 1.202: German text placeholder MSVC 2008

Das kompilierte Programm stürzt nach der Ausführung ab. Das verwundert nicht. Schauen wir, was genau den Absturz verursacht.
Laden wir das Programm in OllyDbg und verfolgen den Ablauf, bis alle 30 Elemente geschrieben worden sind:

Abbildung 1.91: OllyDbg: nach Wiederherstellung des Wertes von EBP

255
Nachverfolgen bis zum Ende der Funktion:

Abbildung 1.92: OllyDbg: EIP wurde wiederhergestellt, aber OllyDbg kann an 0x15 nicht disassemblieren

Richten wir unser Augenmerk auf die Register.

EIP ist jetzt gerade 0x15. Das ist keine gültige Adresse für Code—zumindest nicht für win32 Code! Interessant ist auch, dass das EBP Register 0x14 enthält und ECX sowie EDX jeweils 0x1D

Schauen wir uns das Stacklayout etwas genauer an.

Nachdem der Control Flow an main() übergeben worde ist, wurde der Wert in EBP auf dem Stack abgelegt. Danach wurden 84 Byte für das Array und die Variable i reserviert. Das entspricht $(20+1) \times \text{sizeof(int)}$. ESP zeigt jetzt auf die Variable _i im lokalen Stack und nach der Ausführung von PUSH something scheint sich something neben _i zu befinden.

Hier ist das Stacklayout während der Control Flow in der main() ist:

```
| ESP     | 4 Byte reserviert für Variable i |
| ESP+4   | 80 Byte reserviert für Array a[20] |
| ESP+84  | sichere Wert von EBP |
| ESP+88  | Rücksprungsadresse |
```

Der Befehl a[19]=something schreibt den letzten int innerhalb der Grenzen des Arrays (bis hierhin ist alles in Ordnung!). Der Befehl a[20]=something schreibt something an die Stelle, an der der EBP gespeichert ist.

Sehen wir uns den Zustand der Register im Moment des Absturzes an. In unserem Fall wurde 20 in das zwanzigste Element geschrieben. Am Ende der Funktion stellt der Funktionsepilog den originalen Wert von
EBP wieder her. (20 dezimal entspricht 0x14 hexadezimal). Danach wird RET ausgeführt, was äquivalent zum Befehl POP EIP ist.

Der Befehl RET nimmt die Rücksprungadresse vom Stack (das ist die Adresse in CRT, die main() aufgerufen hat) und speichert hier den Wert 21 (0x15 hexadezimal). Die CPU springt an die Adresse 0x15, aber hier befindet sich kein ausführbarer Code, sodass eine Exception geworfen wird.

Dies nennt man einen Buffer Overflow\(^\text{114}\).

Ersetzt man das int Array durch einen String (char Array) und erzeugt absichtlich einen langen String und übergibt ihn im Programm an eine Funktion, die die Länge des Strings nicht prüft und ihn in einen kurzen Buffer kopiert, kann man das Programm zwingen an eine bestimmte Adresse zu springen. In der Realität ist dieses Verhalten nicht so einfach zu erzeugen, funktioniert aber von Prinzip her genau wie hier. Ein klassischer Artikel dazu:[Aleph One, *Smashing The Stack For Fun And Profit*, (1996)]\(^\text{115}\).

**GCC**

Kompilieren wir denselben Code mit GCC 4.4.1, erhalten wir:

```assembly
public main
main proc near
a = dword ptr -54h
i = dword ptr -4
    push    ebp
    mov     ebp, esp
    sub     esp, 60h
    mov     [ebp+i], 0
    jmp     short loc_80483D1
loc_80483C3:
    mov     eax, [ebp+i]
    mov     edx, [ebp+i]
    mov     [ebp+eax*4+a], edx
    add     [ebp+i], 1
loc_80483D1:
    cmp     [ebp+i], 1Dh
    jle     short loc_80483C3
    mov     eax, 0
    leave
    ret
main endp
```

Lässt man das Programm unter Linux laufen, lautet das Ergebnis: Segmentation fault.

Wenn wir es mit dem GDB Debugger laufen lassen, erhalten wir das Folgende:

```
(gdb) r
Starting program: /home/dennis/RE/1
Program received signal SIGSEGV, Segmentation fault.
0x000000016 in ?? ()
(gdb) info registers
eax  0x0    0
ecx  0x2df96388 -755407992
edx  0x1d  29
ebx  0x26eff4 2551796
esp  0xbfffffff0 0xbfffffff0
ebp  0x15  0x15
esi  0x0    0
edi  0x0    0
eip  0x16  0x16
eflags 0x10202 [ IF RF ]
cs  0x73  115
ss  0x7b  123
```

\(^{114}\)\[wikipedia\]

\(^{115}\)\[German text placeholder http://go.yurichev.com/17266\]
Die Registerwerte unterscheiden sich geringfügig vom win32 Beispiel, da auch das Stacklayout ein wenig anders ist.

### 1.18.3 Schutz vor Buffer Overflows

Es gibt verschiedene Möglichkeiten um sich vor solchen Problemen zu schützen, unabhängig von der Unachtsamkeit des C/C++ Programmierers. MSVC kennt Optionen wie\[^{116}\]:

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

Eine Methode ist eine Zufallszahl zwischen die lokalen Variablen auf dem Stack am Funktionsprolog zu schreiben und diesen im Funktionsepilog vor dem Beenden der Funktion zu überprüfen. Wenn der Wert nicht identisch ist, sollte der letzte RET Befehl nicht ausgeführt werden, sondern das Programm angehalten werden. Der Prozess wird anhalten, aber das ist deutlich besser als eine Fernattacke auf Ihren Rechner.


Wenn wir unser einfaches Arraybeispiel in MSVC\[^{118}\] mit Optionen RTC1 und RTCs kompilieren (1.18.1 on page 243) finden wir einen Aufruf von `@_RTC_CheckStackVars@8`, eine Funktion am Ende der Funktion, die prüft, ob der „canary“ korrekt ist.

Schauen wir uns an, wie GCC die Sache handhabt. Betrachten wir ein Beispiel mit `alloca()` (1.6.3 on page 34):

```c
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
    #ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
    #else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
    #endif
    puts (buf);
}
```

Ohne zusätzliche Optionen fügt GCC 4.7.3 standardmäßig dem Code einen „canary“ zum Überprüfen hinzu:

```assembly
Listing 1.203: GCC 4.7.3
```

\[^{116}\]Compilerseitiger Schutz vor Buffer Overflows: wikipedia.org/wiki/Buffer_overflow_protection

\[^{117}\]wikipedia.org/wiki/Domestic_canary#Miner.27s_canary

\[^{118}\]Microsoft Visual C++
Der Zufallswert befindet sich in gs:20. Er wird auf den Stack geschrieben und am Ende der Funktion wird der Wert auf dem Stack mit dem korrekten „canary“ in gs:20 verglichen. Wenn die Werte ungleich sind, wird die Funktion __stack_chk_fail aufgerufen und wir erkennen in der Konsole in etwa das Folgende (Ubuntu 13.04 x86):

```
*** buffer overflow detected ***: ./2_1 terminated
====== Backtrace: ========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935]
====== Memory map: ========
08048000-08049000 r-xp 00000000 08:01 2097586 /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586 /home/dennis/2_1
0804a000-0804b000 rw-p 00010000 08:01 2097586 /home/dennis/2_1
094d1000-094f2000 rw-p 00000000 00:00 0 [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781 /lib/i386-linux-gnu/libc.so.2.17.so
b7740000-b7742000 r--p 001a0000 08:01 1050781 /lib/i386-linux-gnu/libc.so.2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781 /lib/i386-linux-gnu/libc.so.2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b7749c000-b775d000 rw-p 00000000 00:00 0 [vdso]
b775a000-b775d000 rw-p 00000000 00:00 0 [stack]
Aborted (core dumped)
```

gs ist das sogenannte Segmentregister. Diese Register wurden zu Zeiten von MS-DOS und DOS-Erweiterungen häufig verwendet. Heute ist sein Zweck ein anderer: Kurz gesagt, zeigt das gs Register in Linux stets auf den TLS\(^{119}\) (\(6.2\) on page \(456\))–hier werden threadspezifische Informationen gespeichert. In win32 spielt

\(^{119}\)Thread Local Storage
das fs Register übrigens die gleiche Rolle und zeigt stets auf TIB\textsuperscript{120,121}.

Mehr Informationen finden sich im Quellcode des Linux Kernels (zumindest in der Version 3.11), in
\texttt{arch/x86/include/asm/stackprotector.h} wird diese Variable in den Kommentaren beschrieben.

\textbf{German text placeholder} \textit{Xcode 4.6.3 (LLVM)} (German text placeholder)

Betrachten wir nochmals unser einfaches Arraybeispiel(\textit{1.18.1 on page 243}), erkennen wir nun wie LLVM die Korrektheit des „canary“ überprüft:

\begin{verbatim}
\texttt{main}
\begin{verbatim}
    var_64 = -0x64
    var_60 = -0x60
    var_5C = -0x5C
    var_58 = -0x58
    var_54 = -0x54
    var_50 = -0x50
    var_4C = -0x4C
    var_48 = -0x48
    var_44 = -0x44
    var_40 = -0x40
    var_3C = -0x3C
    var_38 = -0x38
    var_34 = -0x34
    var_30 = -0x30
    var_2C = -0x2C
    var_28 = -0x28
    var_24 = -0x24
    var_20 = -0x20
    var_1C = -0x1C
    var_18 = -0x18
    canary = -0x14
    var_10 = -0x10

    PUSH {R4-R7,LR}
    ADD R7, SP, #0x8
    STR.W R8, [SP,#0x8+var_10]!
    SUB SP, SP, #0x54
    MOVS R0, #aobjc_methtype ; "objc_methtype"
    MOVS R2, #0
    MOVT.W R0, #0
    MOVS R5, #0
    ADD R0, PC
    LDR.W R8, [R0]
    LDR.W R0, [R8]
    STR R0, [SP,#0x64+canary]
    MOVS R0, #2
    STR R2, [SP,#0x64+var_64]
    STR R0, [SP,#0x64+var_60]
    MOVS R0, #4
    STR R0, [SP,#0x64+var_5C]
    MOVS R0, #6
    STR R0, [SP,#0x64+var_58]
    MOVS R0, #8
    STR R0, [SP,#0x64+var_54]
    MOVS R0, #0xA
    STR R0, [SP,#0x64+var_50]
    MOVS R0, #0xC
    STR R0, [SP,#0x64+var_4C]
    MOVS R0, #0xE
    STR R0, [SP,#0x64+var_48]
    MOVS R0, #0x10
    STR R0, [SP,#0x64+var_44]
    MOVS R0, #0x12
    STR R0, [SP,#0x64+var_40]
    MOVS R0, #0x14
\end{verbatim}
\end{verbatim}

\textsuperscript{120}\texttt{Thread Information Block}
\textsuperscript{121}\url{wikipedia.org/wiki/Win32_Thread_INFORMATION_Block}

\end{verbatim}
Zunächst hat LLVM die Schleife entwickelt und alle Werte werden nacheinander vorberechnet in ein Array geschrieben, da LLVM dies für schneller hält. Befehle im ARM mode können helfen, das noch schneller auszuführen und dies herauszufinden könnte Ihre Aufgabe sein.

Am Ende der Funktion sehen wir den Vergleich der beiden „canaries“–dem im lokalen Stack und dem richtigen, auf den R8 zeigt. Wenn sie gleich sind wird durch ITTTT EQ ein Block aus vier Befehlen ausgeführt, der 0 nach R0 schreibt, den Funktionsepiolog durchführt und dann beendet. Wenn die „canaries“ ungleich sind, wird der Block übersprungen und es wird zu ___stack_chk_fail gesprungen und die Ausführung wird angehalten.

### 1.18.4 NocheinWort zu Arrays

Wir verstehen nun warum es nicht möglich ist etwas wie das Folgende in C/C++ Code zu schreiben:

```c
void f(int size) {
    int a[size];
    ...
}
```
Das liegt daran, dass der Compiler die exakte Größe des Arrays zur Compilerzeit kennen muss, um Platz auf dem lokalen Stack zu reservieren.

Wenn man ein Array beliebiger Größe benötigt, muss es über malloc() angelegt werden und dann über den reservieren Speicherblock als Arrays von Variablen des benötigten Typs angesprochen werden.

Oder man verwendet das C99 Standardfeature [ISO/IEC 9899:TC3 (C99 standard), (2007)6.7.5/2], dass intern wie alloca() (1.6.3 on page 34) arbeitet.

Es ist auch möglich, C-Bibliotheken zu verwenden, die als Garbagecollector fungieren. Des Weiteren gibt es auch Bibliotheken für C++, die intelligente Pointer unterstützen.

### 1.18.5 Array von Stringpointern

Hier ist ein Beispiel für ein Array aus Pointern.

Listing 1.204: Get month name

```c
#include <stdio.h>

const char* month1[] = {
    "January", "February", "March", "April",
    "May", "June", "July", "August",
    "September", "October", "November", "December"
};

// in 0..11 range
const char* get_month1 (int month)
{
    return month1[month];
}
```

#### x64

Listing 1.205: *German text placeholder* MSVC 2013 x64

```assembly
_DATA SEGMENT
month1 DQ FLAT:$SG3122
    DQ FLAT:$SG3123
    DQ FLAT:$SG3124
    DQ FLAT:$SG3125
    DQ FLAT:$SG3126
    DQ FLAT:$SG3127
    DQ FLAT:$SG3128
    DQ FLAT:$SG3129
    DQ FLAT:$SG3130
    DQ FLAT:$SG3131
    DQ FLAT:$SG3132
    DQ FLAT:$SG3133
$SG3122 DB 'January', 00H
$SG3123 DB 'February', 00H
$SG3124 DB 'March', 00H
$SG3125 DB 'April', 00H
$SG3126 DB 'May', 00H
$SG3127 DB 'June', 00H
$SG3128 DB 'July', 00H
$SG3129 DB 'August', 00H
$SG3130 DB 'September', 00H
$SG3135 DB '%s', 0AH, 00H
$SG3131 DB 'October', 00H
$SG3132 DB 'November', 00H
$SG3133 DB 'December', 00H
_DATA ENDS

months = 8
get_month1 PROC
    movsx rax, ecx
    lea rcx, OFFSET FLAT:month1
    mov rax, QWORD PTR [rcx+rax*8]
get_month1 ENDP
```
Der Code ist sehr einfach:

- Der erste MOVSD Befehl kopiert einen 32-Bit-Wert aus ECX (wohin das month Argument übergeben wird) nach RAX und erweitert ihn um ein Vorzeichen (da month vom Typ int ist).
  Der Grund für die Erweiterung ist, dass dieser 32-Bit-Wert in Berechnungen mit anderen 64-Bit-Werten zusammen verwendet wird.\(^\text{122}\)
- Danach wird die Adresse der Pointertabelle nach RCX geladen.
- Schließlich wird der Eingabewert (month) mit 9 multipliziert und zur Adresse addiert. Es gilt: wir befinden uns in einer 64-Bit-Umgebung und alle Adressen (oder Pointer) benötigen genau 64 Bit (oder 8 Byte) zum Speichern. Deshalb ist jedes Element der Tabelle 8 Byte breit. Ebenfalls deshalb müssen, um ein spezifisches Element auszuwählen month\(^8\) Bytes vom Start weg übersprüngen werden. Genau das tut MOV. Zusätzlich lädt dieser Befehl auch ein Element in diese Adresse. Für 1 würde das Element ein Pointer auf den String „Februar“ sein, etc.

German text placeholder GCC 4.9 ist noch effizienter:\(^\text{123}\):

<table>
<thead>
<tr>
<th>Listing 1.206: German text placeholder GCC 4.9 x64</th>
</tr>
</thead>
<tbody>
<tr>
<td>movsx rdi, edi</td>
</tr>
<tr>
<td>mov rax, QWORD PTR month1[0+rdi*8]</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>

32-bit MSVC
Kompilieren wir den Code mit dem 32-Bit-MSVC-Compiler:

<table>
<thead>
<tr>
<th>Listing 1.207: German text placeholder MSVC 2013 x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>_month$ = 8</td>
</tr>
<tr>
<td>_get_month1 PROC</td>
</tr>
<tr>
<td>mov eax, DWORD PTR _month$[esp-4]</td>
</tr>
<tr>
<td>mov eax, DWORD PTR _month1[eax*4]</td>
</tr>
<tr>
<td>ret</td>
</tr>
<tr>
<td>_get_month1 ENDP</td>
</tr>
</tbody>
</table>

Der Eingabewert darf nicht zu einem 64-Bit-Wert erweitert werden und wird so wie er ist verwendet. Er wird mit 4 multipliziert, da die Tabellenelemente 32 Bit (oder 4 Byte) breit sind.

32-bit ARM

ARM im ARM mode

<table>
<thead>
<tr>
<th>Listing 1.208: German text placeholder Keil 6/2013 (German text placeholder)</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_month1 PROC</td>
</tr>
<tr>
<td>LDR r1,</td>
</tr>
<tr>
<td>LDR r0, [r1, r0, LSL #2]</td>
</tr>
<tr>
<td>BX lr</td>
</tr>
<tr>
<td>ENDP</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>DCD</td>
</tr>
<tr>
<td>DCB &quot;January&quot;, 0</td>
</tr>
<tr>
<td>DCB &quot;February&quot;, 0</td>
</tr>
<tr>
<td>DCB &quot;March&quot;, 0</td>
</tr>
<tr>
<td>DCB &quot;April&quot;, 0</td>
</tr>
<tr>
<td>DCB &quot;May&quot;, 0</td>
</tr>
</tbody>
</table>

\(^\text{122}\)Es ist seltsam, aber negative Arrayindizes für month können hier verwendet werden (negative Arrayindizes werden später erklärt: ?? on page ??). Wenn dies passiert, wird der negative Eingabewert vom Typ int korrekt um ein Vorzeichen erweitert und das zugehörige Element vor der Tabelle wird ausgewählt. Ohne Vorzeichenerweiterung würde es nicht korrekt funktionieren.
\(^\text{123}\)"0+" blieb im Listing, da der GCC Assembler-Output nicht sauber genug ist, um es zu eliminieren.
Die Adresse der Tabelle wird nach R1 geladen.
Der ganze Rest wird mit lediglich einem LDR Befehl erledigt.
Danach wird der Eingabewert `month` um zwei Bit nach links verschoben (dies entspricht einer Multiplikation mit 4), schließlich zu R1 addiert (wo sich die Adresse der Tabelle befindet) und schließlich wird ein Tabellenelement aus dieser Adresse geladen.
Das 32-Bit-Tabellenelement wird aus der Tabelle nach R0 geladen.

**ARM im Thumb mode**
Der Code ist fast identisch, aber weniger dicht, denn der LSL Suffix kann hier nicht an den LDR Befehl angehängt werden:

```assembly
get_month1 PROC
  LSLS  r0, r0, #2
  LDR  r1, |L0.64|
  LDR  r0, [r1, r0]
  BX  lr
ENDP
```

**ARM64**

Listing 1.209: German text placeholder GCC 4.9 ARM64

```assembly
get_month1:
  adrp  x1, .LANCHOR0
  add  x1, x1, :lo12:.LANCHOR0
  ldr  x0, [x1, w0, sxtw 3]
  ret

.LANCHOR0 = . + 0
.type  month1, %object
.size  month1, 96

month1:
  .xword  .LC2
  .xword  .LC3
  .xword  .LC4
  .xword  .LC5
  .xword  .LC6
  .xword  .LC7
  .xword  .LC8
  .xword  .LC9
  .xword  .LC10
```

264
Die Adresse der Tabelle wird mit ADRP/ADD nach X1 geladen.

Dann wird das zugehörige Element mit einem LDR ausgewählt, das W0 nimmt (das Register, in dem sich der Eingabewert month befindet), es um 3 Bit nach links verschiebt (was einer Multiplikation mit 8 entspricht), um ein Vorzeichen erweitert (das bedeutet der Suffix „sxtw“) und dann zu X0 addiert. Schließlich wird der 64-Bit-Wert aus der Tabelle nach X0 geladen.

**MIPS**

Listing 1.210: German text placeholder GCC 4.4.5 (IDA)

```
.get_month1:
    ; lade Adresse der Tabelle nach $v0:
    la $v0, month1
    ; nimm den Eingabewert und multipliziere mit 4:
    sll $a0, 2
    ; addiere Adresse der Tabelle und berechneten Wert:
    addu $a0, $v0
    ; lade Tabellenelement an dieser Adresse nach $v0:
    lw $v0, 0($a0)
    ; Rückgabe
    jr $ra
    or $at, $zero ; branch delay slot, NOP
.data # .data.rel.local
.globl month1
month1: .word aJanuary # "January"
 .word aFebruary # "February"
 .word aMarch # "March"
 .word aApril # "April"
 .word aMay # "May"
 .word aJune # "June"
 .word aJuly # "July"
 .word aAugust # "August"
 .word aSeptember # "September"
 .word aOctober # "October"
 .word aNovember # "November"
 .word aDecember # "December"
.data # .rodata.str1.4
aJanuary: .ascii "January"<0>
aFebruary: .ascii "February"<0>
```
Array Overflow

Unsere Funktion akzeptiert Werte im Bereich von 0 bis 11, aber was, wenn 12 übergeben wird? Es gibt an
dieser Stelle in der Tabelle kein Element.

Die Funktion wird also irgendeinen dort befindlichen Wert laden und ihn zurückgeben.

Kurz danach kann eine andere Funktion versuchen, einen Textstring von dieser Adresse zu laden und
könnte abstürzen.

Kompilieren wir das Beispiel mit MSVC für win64 und öffnen es in IDA, um zu sehen was der Linker hinter
ter Tabelle angelegt hat:

Listing 1.211: Executable file in IDA

```assembly
off_140011000 dq offset aJanuary_1 ; DATA XREF: .text:0000000140001003
  dq offset aFebruary_1 ; "February"
  dq offset aMarch_1 ; "March"
  dq offset aApril_1 ; "April"
  dq offset aMay_1 ; "May"
  dq offset aJune_1 ; "June"
  dq offset aJuly_1 ; "July"
  dq offset aAugust_1 ; "August"
  dq offset aSeptember_1 ; "September"
  dq offset aOctober_1 ; "October"
  dq offset aNovember_1 ; "November"
  dq offset aDecember_1 ; "December"
aJanuary_1 db 'January',0 ; DATA XREF: sub_140001020+4
  .data:off_140011000
daFebruary_1 db 'February',0 ; DATA XREF: .data:0000000140001008
  align 4
aMarch_1 db 'March',0 ; DATA XREF: .data:0000000140001010
  align 4
aApril_1 db 'April',0 ; DATA XREF: .data:0000000140001018
```

Die Monatsnamen befinden sich direkt dahinter.

Unser Programm ist winzig, sodass hier nicht viele Daten im Datensegment abgelegt werden müssen, nur
die Monatsnamen. Wir stellen aber fest, dass sich hier irgendetwas befinden könnte, was der Linker hier
zufällig platziert hat.

Was also, falls 12 an die Funktion übergeben wird? Das 13. Element wird zurückgegeben.

Schauen wir uns an, wie die CPU die Bytes dort wie einen 64-Bit-Wert behandelt:

Listing 1.212: Executable file in IDA

```assembly
off_140011000 dq offset qword_140011060 ; DATA XREF: .text:000000140001003
  dq offset aFebruary_1 ; "February"
  dq offset aMarch_1 ; "March"
  dq offset aApril_1 ; "April"
  dq offset aMay_1 ; "May"
  dq offset aJune_1 ; "June"
  dq offset aJuly_1 ; "July"
  dq offset aAugust_1 ; "August"
  dq offset aSeptember_1 ; "September"
  dq offset aOctober_1 ; "October"
  dq offset aNovember_1 ; "November"
```

266
Dieser Wert ist 0x797261756E614A.

Kurz danach könnte eine andere Funktion (möglicherweise eine, die Strings verarbeitet) versuchen, Bytes von dieser Adresse zu lesen, weil sie hier einen C-String erwartet.

Höchstwahrscheinlich wird dies zu einem Absturz führen, da der Wert keine gültige Adresse sein wird.

**Schutz vor Array Overflows**

Alles, was schiefgehen kann, wird auch schiefgehen

---

Murphy’s Law

Es ist ein wenig naiv zu erwarten, dass jeder Programmierer, der diese Funktion oder Bibliotek verwendet, nie ein größeres Argument als 11 übergeben wird.

Es gibt einen Ansatz, der besagt „scheitere früh und laut“ oder „scheitere schnell“, und dessen Aussage es ist, dass Probleme so früh wie möglich gemeldet werden und das Programm angehalten werden sollte.

Eine solche Methode in C/C++ sind Assertions.

Wir modifizieren unser Programm, sodass es einen Fehler liefert, wenn ein falscher Wert übergeben wird:

**Listing 1.213:** assert() added

```c
const char* get_month1_checked (int month)
{
    assert (month<12);
    return month1[month];
}
```

Das Assertionmakro prüft zu Beginn der Funktion auf valide Werte und liefert einen Fehler, wenn der Ausdruck falsch ist.

**Listing 1.214:** German text placeholder MSVC 2013 x64

```c
$SG3143 DB 'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '.', 00H
$SG3144 DB 'c', 00H, 00H, 00H
$SG3144 DB 'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '<', 00H
$SG3144 DB '1', 00H, '2', 00H, 00H, 00H

month$ = 48
get_month1_checked PROC
$LN5:
    push    rbx
    sub     rsp, 32
    movsxd   rbx, ecx
    cmp      ebx, 12
    jl      SHORT $LN3@get_month1
    lea     rdx, OFFSET FLAT:$SG3143
    lea     rcx, OFFSET FLAT:$SG3144
    mov      r8d, 29
    call     wassert
    lea     rcx, OFFSET FLAT:month1
    mov      rax, QWORD PTR [rcx+rbx*8]
    add      rsp, 32
    pop      rbx
    retn    0
get_month1_checked ENDP
```
Tatsächlich ist assert() keine Funktion, sondern ein Makro. Es prüft auf eine Bedingung und übergibt dann auch die Zeilennummer und den Dateinamen an eine andere Funktion, die diese Informationen an den Benutzer weiterleitet.

Wir erkennen hier, dass sowohl Dateiname als auch Bedingung in UTF-16 kodiert sind. Die Zeilennummer wird ebenfalls übergeben (hier: 29).

Dieser Mechanismus ist möglicherweise in allen Compilern der gleiche. GCC erzeugt den folgenden Code:

```
Listing 1.215: German text placeholder
GCC 4.9 x64

.LC1:
 .string "month.c"
.LC2:
 .string "month<12"

get_month1_checked:
   cmp    edi, 11
   jg    .L6
   movsx  rdi, edi
   mov    rax, QWORD PTR month1[0+rdi*8]
   ret

.L6:
   push   rax
   mov    ecx, OFFSET FLAT:__PRETTY_FUNCTION___.2423
   mov    edx, 29
   mov    esi, OFFSET FLAT:.LC1
   mov    edi, OFFSET FLAT:.LC2
   call   __assert_fail

__PRETTY_FUNCTION___.2423:
 .string "get_month1_checked"
```

Das Makro übergibt praktischerweise in GCC auch den Funktionsnamen. Es gibt aber nichts umsonst und das gilt auch für solche Überprüfungen.

Sie machen das Programm langsamer, vor allem, wenn das assert() Makro in kleinen zeittkritischen Funktionen verwendet wird.


Natürlich arbeitet der „geprüfte“ Kernel wegen der vielen Überprüfungen langsamer, sodass es normalerweise nur zum Debuggen verwendet wird.

### 1.18.6 Multidimensionale Arrays

Intern ist ein multidimensionales Array im Prinzip das gleiche wie ein lineares Array.

Da der Speicher eines Rechners linear ist, ist es ein eindimensionales Array. Zur Vereinfachung kann dieses multidimensionales Array leicht als eindimensionales dargestellt werden.

Beispielsweise werden die Elemente eines 3x4 Arrays folgendermaßen in einem eindimensionalen Array aus 12 Zellen gespeichert:

Tabelle 1.3: Zweidimensionales Array in eindimensionaler Speicherdarstellung

Auf diese Weise wird jede Zellen des 3*4 Arrays im Speicher abgelegt:

<table>
<thead>
<tr>
<th>Offset im Speicher</th>
<th>Arrayelement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0][0]</td>
</tr>
<tr>
<td>1</td>
<td>[0][1]</td>
</tr>
<tr>
<td>2</td>
<td>[0][2]</td>
</tr>
<tr>
<td>3</td>
<td>[0][3]</td>
</tr>
<tr>
<td>4</td>
<td>[1][0]</td>
</tr>
<tr>
<td>5</td>
<td>[1][1]</td>
</tr>
<tr>
<td>6</td>
<td>[1][2]</td>
</tr>
<tr>
<td>7</td>
<td>[1][3]</td>
</tr>
<tr>
<td>8</td>
<td>[2][0]</td>
</tr>
<tr>
<td>9</td>
<td>[2][1]</td>
</tr>
<tr>
<td>10</td>
<td>[2][2]</td>
</tr>
<tr>
<td>11</td>
<td>[2][3]</td>
</tr>
</tbody>
</table>

Tabelle 1.4: Speicheradressen jeder Zelle des zweidimensionalen Arrays

Um also die Adresse des benötigten Elements zu berechnen, multiplizieren wir zunächst den ersten Index mit 4 (der Arraybreite) und addieren dann den zweiten Index. Dies nennt man Zeilenordnung (engl. row-major order) und diese Methode zur Darstellung von Arrays und Matrizen wird mindestens von C/C++ und Python verwendet. Der Ausdruck row-major order bedeutet: „schreibe zuerst die Elemente der ersten Zeilen, dann die zweite Zeile...und schließlich die Elemente der letzten Zeile“.

Eine andere Methode zur Darstellung heißt Spaltenordnung (engl. column-major order) (die Indizes des Arrays werden in umgekehrter Reihenfolge verwendet) und wird zumindest in Fortran, MATLAB und R verwendet. Der Ausdruck column-major oder bedeutet: „schreibe zuerst die Elemente der ersten Spalte, dann die zweite Spalte...und schließlich die Elemente der letzten Spalte“.

Welche Method ist besser?

Generell ist hinsichtlich Performance und Cachespeicher die beste Methode der Datenorganisation diejenige, in der auf die Elemente sequentiell zugegriffen wird.

Wenn eine Funktion auf Daten zeilenweise zugreift, ist Zeilenordnung besser und umgekehrt.

**Beispiel für zweidimensionales Array**

Wir werden mit einem Array vom Typ char arbeiten, was bedeutet, dass jedes Element nur ein Byte Speicherplatz benötigt.

**Beispiel: Zeile füllen**

Füllen wir die zweite Zeilen mit den Werten 0..3:

Listing 1.216: Row filling example

```c
#include <stdio.h>

char a[3][4];

int main()
{
    int x, y;

    // Array leeren
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;

    // zweite Spalte mit 0..3 füllen:
    ```

269
for (y=0; y<4; y++)
    a[1][y]=y;
}

Alle drei Zeilen sind rot markiert. Wir erkennen, dass die zweite Zeilen nun die Werte 0,1,2 und 3 enthält:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000004</td>
<td>00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000008</td>
<td>00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0000000C</td>
<td>00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

Abbildung 1.93: OllyDbg: Array ist befüllt

**Beispiel: Spalte füllen**

Füllen wir die dritte Spalte mit den Werten 0..2:

Listing 1.217: Column filling example

```c
#include <stdio.h>
char a[3][4];
int main()
{
    int x, y;
    // leere Array
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;
    // fülle dritte Spalte mit 0..2:
    for (x=0; x<3; x++)
        a[x][2]=x;
}
```

Die drei Spalten sind hier ebenfalls rot markiert.

Wir erkennen, dass sich in jeder Zeile an der dritten Stelle die Werte 0,1 und 2 befinden.

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000004</td>
<td>00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000008</td>
<td>00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0000000C</td>
<td>00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

Abbildung 1.94: OllyDbg: Array ist befüllt

**Eindimensionaler Zugriff auf zweidimensionales Array**

Wir können uns leicht davon überzeugen, dass es auf mindestens zwei Arten möglich ist, auf ein zweidimensionales Array eindimensional zuzugreifen:

```c
#include <stdio.h>
char a[3][4];

c char get_by_coordinates1 (char array[3][4], int a, int b)
{
    return array[a][b];
};

c char get_by_coordinates2 (char *array, int a, int b)
{
    // behandle Eingabearray eindimensional
    // 4 entspricht der Arraybreite
    return array[a*4+b];
};
```
char get_by_coordinates3 (char *array, int a, int b)
{
    // handle Eingabarray als Pointer,
    // berechne Adresse, lade Wert an dieser Stelle
    // 4 entspricht der Arraybreite
    return *(array+a*4+b);
}

int main()
{
    a[2][3]=123;
    printf("%d\n", get_by_coordinates1(a, 2, 3));
    printf("%d\n", get_by_coordinates2(a, 2, 3));
    printf("%d\n", get_by_coordinates3(a, 2, 3));
}

Kompilieren und ausführen: es zeigt korrekte Werte an Was MSVC 2013 getan hat ist faszinierend: alle
drei Routinen sind identisch!

Listing 1.218: German text placeholder MSVC 2013 x64

arrays$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=address of array
; RDX=a
; R8=b
    movsxd rax, r8d
    ; EAX=b
    movsxd r9, edx
    ; R9=a
    add rax, rcx
    ; RAX=b+address of array
    movzx eax, BYTE PTR [rax+r9*4]
    ; AL=load byte at address RAX+R9*4=b+address of array+a*4=address of array+a*4+b
    ret 0
get_by_coordinates3 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates2 PROC
    movsxd rax, r8d
    movsxd r9, edx
    add rax, rcx
    movzx eax, BYTE PTR [rax+r9*4]
    ret 0
get_by_coordinates2 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates1 PROC
    movsxd rax, r8d
    movsxd r9, edx
    add rax, rcx
    movzx eax, BYTE PTR [rax+r9*4]
    ret 0
get_by_coordinates1 ENDP

GCC erzeugt ebenfalls äquivalente Routinen, aber ein wenig anders:

Listing 1.219: German text placeholder GCC 4.9 x64

; RDI=address of array
; RSI=a
; RDX=b
get_by_coordinates1:
Beispiel: dreidimensionales Array
Mit multidimensionalen Arrays ist es das gleiche.
Wir werden nun mit einem Array vom Typ int arbeiten: jedes Element benötigt 4 Byte Speicherplatz.
Sehen wir es uns an:

Listing 1.220: simple example

```c
#include <stdio.h>
int a[10][20][30];
void insert(int x, int y, int z, int value)
{
    a[x][y][z]=value;
}
```

x86
Wir erhalten das Folgende (MSVC 2010):

Listing 1.221: MSVC 2010

```
_DATA SEGMENT
COMM    _a:DWORD:01770H
_DATA    ENDS
PUBLIC _insert
_TEXT    SEGMENT
_x$ = 8    ; size = 4
_y$ = 12   ; size = 4
_z$ = 16   ; size = 4
_value$ = 20    ; size = 4
_insert   PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _x$[ebp]
    imul eax, 2400    ; eax=600*4*x
    mov ecx, DWORD PTR _y$[ebp]
```
Nichts Außergewöhnliches. Zur Berechnung des Index’ werden in der Formel 
\[ \text{address} = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z \]
drei Eingabewerte verwendet, um das multidimensionale Array zu repräsentieren. Vergessen wir nicht, dass der \textit{int} Typ 32 Bit (4 Byte) breit ist, sodass alle Koeffizienten mit 4 multipliziert werden müssen.

\begin{verbatim}
Listing 1.222: GCC 4.4.1

define public insert
insert proc near
x = dword ptr 8
y = dword ptr 0Ch
z = dword ptr 10h
value = dword ptr 14h
push ebp
mov ebp, esp
push ebx
mov ebx, [ebp+x]
mov eax, [ebp+y]
mov ecx, [ebp+z]
lea edx, [eax+eax] ; edx=y+2
mov eax, edx ; eax=y+2
shl eax, 4 ; eax=(y+2)<<4 = y+2*16 = y+32
sub eax, edx ; eax=y+32 - y+2=y+30
imul edx, ebx, 600 ; edx=x*600
add eax, edx ; eax+edx=y+30 + x*600
lea edx, [eax+ecx] ; edx=y+30 + x*600 + z
mov eax, [ebp+value]
mov dword ptr ds:a[edx*4], eax ; *(a+edx*4)=value
pop ebx
pop ebp
retn
insert endp
end define

Der GCC Compiler arbeitet anders.
Für eine der Operationen in der Berechnung (30y) produziert GCC Code ohne Multiplikationsbefehle. Das funktioniert wie folgt: 
\((y + y) \ll (y + y) = (2y) \ll 2 = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y\).
So werden für die 30y Berechnung nur ein Addierbefehl, eine bitweiser Verschiebebefehl und ein Subtraktionsbefehl verwendet. So geht es schneller.

\textbf{ARM \textit{+} German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)}

\begin{verbatim}
Listing 1.223: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)

_insert

value = -0x10
z = -0xC
y = -.8
x = -.4

; reserviere auf dem lokalen Stack Platz für 4 Werte vom Typ int
SUB SP, SP, #0x10
MOV R9, 0xFC2 ; a
ADD R9, PC
LDR.W R9, [R9] ; lade Pointer auf Array
STR R0, [SP,#0x10+x]

\end{verbatim}

273
Die Tricks für das Ersetzen der Multiplikation durch Verschieben, Addieren und Subtrahieren, die wir bereits kennengelernt haben, kommen hier auch vor.

Hier finden wir auch einen für uns neuen Befehl: RSB (Reverse Subtract).

Er arbeitet genau wie SUB, aber vertauscht die Operanden vor der Ausführung. Warum?

SUB und RSB sind Befehle, bei denen auf den zweiten Operanden eine bitweise Verschiebung angewendet werden kann: (LSL#4). Dieser Koeffizient kann aber nur auf den zweiten Operanden angewendet werden.

Das ist günstig für kommutative Operationen wie Addition und Multiplikation (die Operanden können vertauscht werden, ohne das Ergebnis zu verändern).

Subtraktion dagegen ist nicht kommutativ, weshalb für diese Fälle RSB existiert.

**MIPS**

Das Beispiel ist sehr klein, sodass der GCC Compiler entschieden hat das Array a im 64KiB Platz abzulegen, um es durch den globalen Pointer zugreifbar zu machen.

**ARM**

Die Adresse des Arrayelements wird über die eben gezeigte Formel berechnet.

**Listing 1.224:**

```assembly
MOV R9, #0x10FC
MOV.W R12, #2400
MOVT.W R9, #0
RSB.W R1, R1, R1, LSL#4 ; R1 - y. R1=y<<4 - y = y*16 - y = y*15
ADD R9, PC
LDR.W R9, [R9] ; R9 = Pointer auf ein Array
MLA.W R0, R0, R12, R9 ; R0 - x, R12 - 2400, R9 - Pointer auf a. R0=x*2400 + Pointer auf a
ADD.W R0, R0, R1, LSL#3 ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + Pointer auf a + y*15*8 =
                        ; Pointer auf a + y*30*4 + x*600*4
STR.W R3, [R0,R2,LSL#2] ; R2 - z, R3 - Werte. Adresse=R0+z*4 =
                        ; Pointer auf a + y*30*4 + x*600*4 + z*4
BX LR
```

Die Tricks für das Ersetzen der Multiplikation durch Verschieben, Addieren und Subtrahieren, die wir bereits kennengelernt haben, kommen hier auch vor.
Wie man sieht wird mindestens eine Befehl benötigt, der einen Wert aus dem Speicher lädt, um den Pointer
zu aktualisieren.

Die Antwort ist: ja, wenn man die Liste aus String als zweidimensionales Array darstellt:

Ist es möglich diesen Speicherzugriff loszuwerden?

Betrachten wir erneut die Funktion, die den Namen eines Monats zurückgibt: Listing 1.18.7 Strings als zweidimensionales Array

Ein anderes Beispiel in diesem Buch ist das Spiel Minesweeper: das Feld ist auch ein zweidimensionales Array: ?? on page ??.

**Weitere Beispiele**

Der Bildschirm wird als 2D-Array dargestellt, aber der Videopuffer ist ein lineares 1D-Array. Wir betrachten hier näher: ?? on page ??.

Ein anderes Beispiel in diesem Buch ist das Spiel Minesweeper: das Feld ist auch ein zweidimensionales Array: ?? on page ??.

1.18.7 Strings als zweidimensionales Array

Betrachten wir erneut die Funktion, die den Namen eines Monats zurückgibt: Listing 1.204.

Wie man sieht wird mindestens eine Befehl benötigt, der einen Wert aus dem Speicher lädt, um den Pointer auf den String, der den Monatsnamen enthält, vorzubereiten.

Ist es möglich diesen Speicherzugriff loszuwerden?

Die Antwort ist: ja, wenn man die Liste aus String als zweidimensionales Array darstellt:

```c
#include <stdio.h>
#include <assert.h>

const char month2[12][10] =
{
    { 'J', 'a', 'n', 'u', 'a', 'r', 'y', 0, 0, 0 },
    { 'F', 'e', 'b', 'r', 'u', 'a', 'r', 'y', 0, 0, 0 },
    { 'M', 'a', 'r', 'c', 'h', 0, 0, 0, 0, 0 },
    { 'A', 'p', 'r', 'i', 'l', 'l', 0, 0, 0, 0 },
    { 'M', 'a', 'y', 0, 0, 0, 0, 0, 0, 0 },
    { 'J', 'u', 'n', 'e', 0, 0, 0, 0, 0, 0 },
    { 'J', 'u', 'l', 'y', 0, 0, 0, 0, 0, 0 },
    { 'A', 'u', 'g', 'u', 's', 't', 0, 0, 0, 0 },
    { 'S', 'e', 'p', 't', 'e', 'm', 'b', 'e', 'r', 0 },
    { 'O', 'c', 't', 'o', 'b', 'e', 'r', 'e', 'r', 0, 0 },
    { 'N', 'o', 'v', 'e', 'm', 'b', 'e', 'r', 0, 0 },
    { 'D', 'e', 'c', 'e', 'm', 'b', 'e', 'r', 'r', 0, 0 }
};
```
Hier ist was wir erhalten:

### Listing 1.226: German text placeholder MSVC 2013 x64

```c
const char* get_month2 (int month) {
    return month2[month][0];
}
```

Es gibt überhaupt keine Speicherzugriffe. Alles, was diese Funktion tut, ist einen Pointer zu berechnen, der auf den ersten Buchstaben des Monats zeigt: `pointer_to_the_table + month ⋅ 10`.

Es gibt auch zwei LEA Befehle, die wie mehrere MUL und MOV Befehle funktionieren.

Die Breite des Arrays beträgt 10 Byte.

Der längste String im Beispiel—„September“—hat eine Länge von 9 Byte zuzüglich einer terminierenden Null, also insgesamt 10 Byte.

Die übrigen Monatsnamen werden mit Zerobytes aufgefüllt, sodass alle denselben Speicherplatz (10 Byte) benötigen.

Dadurch arbeitet unsere Funktion noch schneller, denn die Startadresse jedes Strings kann so einfach berechnet werden.

German text placeholder GCC 4.9 kann sogar noch kürzeren Code erzeugen:

### Listing 1.227: German text placeholder GCC 4.9 x64

```assembly
get_month2 PROC
; erweiterte Eingabewert um Vorzeichen und wandle um in 64-Bit-Wert
    movsx rax, ecx
    lea rcx, QWORD PTR [rax+rax*4]
    lea rax, OFFSET FLAT:month2
    RAX=Pointer auf die Tabelle
    lea rax, QWORD PTR [rax+rcx*2]
    ; RAX=Pointer auf die Tabelle + RCX*2=Pointer auf die Tabelle + Monat*5*2=Pointer auf die Tabelle + Monat*10
    ret
get_month2 ENDP
```

LEA wird hier auch für die Multiplikation mit 10 verwendet.

Nicht optimierende Compiler führen die Multiplikation anders durch.

### Listing 1.228: German text placeholder GCC 4.9 x64

```assembly
get_month2:
    push rbp
    mov rbp, rsp
    mov DWORD PTR [rbp-4], edi
```
mov  eax, DWORD PTR [rbp-4]
movsx  rdx, eax
; RDX = erweiterte Eingabewert um Vorzeichen
mov  rax, rdx
; RAX = Monat
sal  rax, 2
; RAX = Monat<<2 = Monat*4
add  rax, rdx
; RAX = RAX+RDX = Monat*4+Monat = Monat*5
add  rax, rax
; RAX = RAX*RAX = Monat*5*2 = Monat*10
add  rax, OFFSET FLAT:month2
; RAX = Monat*10 + Pointer auf die Tabelle
pop  rbp
ret

German text placeholder MSVC verwendet nur den IMUL Befehl:

Listing 1.229: German text placeholder MSVC 2013 x64

month$ = 8
get_month2 PROC
    mov  DWORD PTR [rsp+8], ecx
movsx  eax, DWORD PTR month$[rsp]
; RAX = Eingabewert um Vorzeichen und auf 64 Bit erweitern
    imul  eax, eax, 10
; RAX = RAX*10
    lea  rcx, OFFSET FLAT:month2
; RCX = Pointer auf die Tabelle
    add  rcx, rax
; RCX = RCX+RAX = Pointer auf die Tabelle+Monat*10
    mov  rax, rcx
; RAX = Pointer auf die Tabelle+Monat*10
mov  ecx, 1
; RCX = 1
    imul  rcx, rcx, 0
; RCX = 1*0 = 0
    add  rax, rcx
; RAX = Pointer auf die Tabelle+Monat*10 + 0 = Pointer auf die Tabelle+Monat*10
    add  rax, rcx
; RAX = Pointer auf die Tabelle+Monat*10 + 0 = Pointer auf die Tabelle+Monat*10
    pop  rbp
ret
get_month2 ENDP

Eine Sache hier ist seltsam: warum wird die Multiplikation mit null und die Addition von null zum Endergebnis hinzugefügt?

Dies sieht wie ein Fehler im Codegenerator des Compilers aus, der nicht durch die Tests des Compilers abgefangen wurde. (Trotzdem funktioniert der erzeugte Code korrekt.)

Wir betrachten solche Codes ganz bewußt, damit der Leser sich klarmacht, dass man sich über solche Merkwürdigkeiten und Artefakte des Compilers nicht allzu sehr wundern soll.

32-bit ARM

German text placeholder Keil im Thumb mode verwendet zur Multiplikation den Befehl MULS:

Listing 1.230: German text placeholder Keil 6/2013 (German text placeholder)

; R0 = Monat
    MOVS  r1,#0xa
; R1 = 10
    MULS  r0,r1,r0
; R0 = R1*R0 = 10*Monat
    LDR  r1,[r0,68]
; R1 = Pointer auf die Tabelle
    ADDS  r0,r0,r1
; R0 = R0+R1 = 10*Monat + Pointer auf die Tabelle
    BX  lr

German text placeholder Keil für ARM mode verwendet Additions- und Schiebebefehle:
ARM64

Listing 1.232: German text placeholder GCC 4.9 ARM64

; W0 = Monat
  sxtw  x0, w0
; X0 = vorzeichenenerweiterter Eingabewert
  adrp  x1, .LANCHOR1
  add   x1, x1, :lo12:.LANCHOR1
; X1 = Pointer auf die Tabelle
  add   x0, x0, x0, lsl 2
; X0 = X0+X0<<2 = X0+X0*4 = X0*5
  add   x0, x1, x0, lsl 1
; X0 = X1+X0<<1 = X1+X0*2 = Pointer auf die Tabelle + X0*10
ret

SXTW wird für Vorzeichenenerweiterung und Übertragung von 32-Bit-Werten in 64-Bit-Werte und das Speichern in X0 verwendet.

Das ADRP/ADD Paar wird für das Laden der Adresse der Tabelle verwendet.

Der ADD Befehl trägt auch den LSL Suffix, der bei der Multiplikation hilft.

MIPS

Listing 1.233: German text placeholder GCC 4.4.5 (IDA)

.get_month2:
.globl get_month2
; $a0=month
  sll $v0, $a0, 3
; $v0 = $a0<<3 = month*8
  sll $a0, 1
; $a0 = $a0<<1 = month*2
  addu $a0, $v0
; $a0 = month*2+month*8 = month*10
; load address of the table:
  la $v0, month2
; sum up table address and index we calculated and return:
  jr $ra
  addu $v0, $a0

month2: .ascii "January"<0>
    .byte 0, 0, 0
aFebruary: .ascii "February"<0>
    .byte 0, 0, 0
aMarch: .ascii "March"<0>
    .byte 0, 0, 0, 0
aApril: .ascii "April"<0>
    .byte 0, 0, 0, 0
aMay: .ascii "May"<0>
    .byte 0, 0, 0, 0, 0, 0
aJune: .ascii "June"<0>
    .byte 0, 0, 0, 0, 0
aJuly: .ascii "July"<0>
    .byte 0, 0, 0, 0, 0
aAugust: .ascii "August"<0>
    .byte 0, 0, 0
aSeptember: .ascii "September"<0>
Fazit


1.18.8 Fazit

Ein Array ist eine Ansammlung von Werten, die im Speicher nebeneinander angeordnet sind.
Dies gilt für alle Elementtypen und sogar für Structs.
Der Zugriff auf ein spezielles Element des Arrays entspricht lediglich eine Berechnung von dessen Adresse.

1.18.9 Übungen

- http://challenges.re/62
- http://challenges.re/63
- http://challenges.re/64
- http://challenges.re/65
- http://challenges.re/66

1.19 Manipulieren einzelner Bits

Eine Menge Funktionen definiert ihre Eingabeargumente als Flags in Bitfields.
Natürlich können diese auch durch Variablen von Typ bool ersetzt werden; das wäre jedoch umständlicher als nötig.

1.19.1 German text placeholder

x86

Win32 API Beispiel:

HANDLE fh;

fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

We get (MSVC 2010):

Listing 1.234: MSVC 2010

push 0
push 128 ; 00000080H
push 4
push 0
push 1
push -1073741824 ; c0000000H
push OFFSET $SG78813
call DWORD PTR __imp__CreateFileA@28
mov DWORD PTR _fh$[ebp], eax
Schauen wir uns WinNT.h genauer an:

### Listing 1.235: WinNT.h

```c
#define GENERIC_READ (0x80000000L)
#define GENERIC_WRITE (0x40000000L)
#define GENERIC_EXECUTE (0x20000000L)
#define GENERIC_ALL (0x10000000L)
```

Alles eindeutig beschrieben: GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000. Dieser Wert wird als zweites Argument für die Funktion `CreateFile()` verwendet.

Wie würde `CreateFile()` diese Flags überprüfen? Wenn wir und die `KERNEL32.DLL` in Windows XP SP3 x86 anschauen, finden wir dieses Codefragment in `CreateFileW`:

### Listing 1.236: KERNEL32.DLL (Windows XP SP3 x86)

```c
.text:7C83D429 test byte ptr [ebp+dwDesiredAccess+3], 40h
.text:7C83D42D mov [ebp+var_8], 1
.text:7C83D434 jz short loc_7C83D417
.text:7C83D436 jmp loc_7C810817
```

Wir finden hier den `TEST` Befehl, aber dieser nimmt nicht das ganze zweite Argument, sondern nur das MSB (`ebp+dwDesiredAccess+3`) und prüft es auf das Flag `0x40` (welches hier dem GENERIC_WRITE Flag entspricht).

TEST ist prinzipiell der gleiche Befehl wie `AND`, aber das Ergebnis wird nicht gespeichert. (Erinnern wir uns, dass `CMP` das gleiche macht wie `SUB`, aber auch ohne das Ergebnis zu speichern (1.8.4 on page 71)).

Die Logik dieses Codefragments ist die folgende:

```c
if ((dwDesiredAccess & 0x40000000) == 0) goto loc_7C83D417
```

Wenn der AND Befehl dieses Bit hinterlässt, wird das ZF Flag gelöscht und der bedingte Sprung `JZ` wird nicht ausgeführt. Der bedingte Sprung wird nur dann ausgeführt, wenn das Bit `0x40000000` in der Variable `dwDesiredAccess` fehlt — dann ist das Ergebnis von `AND 0`, ZF wird gesetzt und der bedingte Sprung wird ausgeführt.

Schauen wir es uns mit GCC 4.4.1 unter Linux an:

```c
#include <stdio.h>
#include <fcntl.h>

void main()
{
    int handle;
    handle=open ("file", 0_RDWR | 0_CREAT);
}
```

Wir erhalten folgenden Code:

### Listing 1.237: GCC 4.4.1

```asm
public main
main proc near
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_4 = dword ptr -4
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
```

---

Wenn wir uns die Funktion open() in der Bibliothek libc.so.6 anschauen, gibt es nur einen syscall:

```c
Listing 1.238: open() (libc.so.6)
```

Die Bitfields für open() werden also offenbar irgendwo im Linux Kernel geprüft.

Natürlich ist es ein Leichtes sowohl GLibc als auch den Quellcode des Linux Kernels herunterzuladen, aber wir wollen wir Sache ohne den Quellcode verstehen.

Wenn also in Linux 2.6 der syscall sys_open verwendet wird, wird die Kontrolle an do_sys_open übergeben und anschließend von dort aus— an die Funktion do_filp_open() (diese befindet sich im Verzeichnisbaum des Kernel-Quellcodes in fs/namei.c).

Neben der Übergabe von Argumenten über den Stack gibt es auch die Möglichkeit einige von ihnen direkt über Register zu übergeben. Dies wird auch fastcall (6.1.3 on page 449) genannt. Es ist schneller, da die CPU nicht auf den Stack im Speicher zugreifen muss, um die Funktionsargumente einzulesen. GCC kennt dafür die Option regparm126, mithilfe derer es möglich ist, die Anzahl der Argumente anzugeben, die über Register übergeben werden sollen.

Der Linux 2.6 Kernel wird mit der Option -mregparm=3 kompiliert127,128. Dies bedeutet, dass die erste 3 Argumente über die Register EAX, EDX und ECX übergeben werden und der Rest über den Stack. Ist die Anzahl der Argumente kleiner als 3 wird natürlich nur ein Teil der genannten Register verwendet.

Laden wir also den Linux Kernel 2.6.31 herunter, kompilieren ihn in Ubuntu (make vmlinux) und öffnen ihn in IDA, so finden wir die Funktion do_filp_open(). Am Beginn derselben finden wir den folgenden Code (die Kommentare stammen vom Autor):

```c
Listing 1.239: do_filp_open() (Linux Kernel 2.6.31)
```

GCC speichert die Werte der ersten drei Argumente auf dem lokalen Stack. Der Compiler würde diese Register ansonsten nicht verwenden und das wäre für den German text placeholder des Compilers nicht umsetzbar.

Finden wir dieses Codefragment:

126ohse.de/uwe/articles/gcc-attributes.html#func-regparm
127kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f
128Siehe auch die arch/x86/include/asm/calling.h Datei im Kernel Verzeichnisbaum
0x40—enspricht dem O_CREAT Makro. open_flag wird auf Anwesenheit des 0x40 Bits hin überprüft und, wenn das Bit 1 ist, wird der folgende JNZ Befehl ausgelöst.

ARM
Das O_CREAT Bit wird im Linux Kernel 3.8.0 anders überprüft:

```c
static int do_last(struct nameidata *nd, struct path *path, struct file *file, const struct open_flags *op, int *opened, struct filename *name)
{
    if (!(open_flag & O_CREAT)) {
        error = lookup_fast(nd, path, &inode);
    } else {
        error = complete_walk(nd);
    }
}
```

So sieht der für den ARM mode kompilierte Kernel in IDA aus:

```assembly
Listing 1.242: do_last() aus vmlinux (IDA)

.text:C0169EA8 MOV R9, R3 ; R3 - (4th argument) open_flag
.text:C0169ED4 LDR R6, [R9] ; R6 - open_flag
.text:C0169F68 TST R6, #0x40 ; jumptable C0169F00 default case
.text:C0169F6C BNE loc_C016A128
.text:C0169F70 LDR R2, [R4,#0x10]
.text:C0169F74 ADD R12, R4, #8
.text:C0169F78 LDR R3, [R4,#0xC]
```

282
TST ist analog zum Befehl TEST in x86. Wir können dies in diesem Codefragment daran erkennen, dass entweder lookup_fast() oder complete_walk() ausgeführt wird. Dies entspricht dem Quellcode der Funktion do_last(). Das Makro O_CREAT entspricht hier 0x40.

### 1.19.2 German text placeholder

German text placeholder:

```c
#include <stdio.h>

#define IS_SET(flag, bit) ((flag) & (bit))
#define SET_BIT(var, bit) ((var) |= (bit))
#define REMOVE_BIT(var, bit) ((var) &= ~(bit))

int f(int a)
{
    int rt=a;
    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);
    return rt;
}

int main()
{
    f(0x12340678);
}
```

### x86

**German text placeholder MSVC**

Wir erhalten folgenden Code: (MSVC 2010):

```
Listing 1.243: MSVC 2010

    _rt$ = -4 ; size = 4
    _a$ = 8 ; size = 4
    _f PROC
    push ebp
    mov ebp, esp
    push ecx
    mov eax, DWORD PTR _a$[ebp]
    mov DWORD PTR _rt$[ebp], eax
    mov ecx, DWORD PTR _rt$[ebp]
    or ecx, 16384 ; 00004000H
```

283
Der OR Befehl setzt ein Bit auf einen Wert, während die übrigen ignoriert werden.

AND ressetet ein Bit. Man kann sagen, dass AND einfach alle Bits bis auf eines kopiert. Tatsächlich werden im zweiten Operanden von AND nur die Bits gesetzt, die auch gespeichert werden müssen, lediglich das eine, das nicht kopiert werden soll (die 0 in der Bitmaske), wird nicht gesetzt. Auf diese Weise kann man sich die Logik des Befehls leichter merken.
Untersuchen wir dieses Beispiel in OllyDbg. Schauen wir zunächst die binäre Form der zu verwendenden Konstanten an:

- $0x200$ (0b0000000000000000000100000000) (d.h. das 10. Bit (vom ersten aus gezählt)).
- $0x200$ ist invertiert: $0xFFFFFDFF$ (0b111111111111111111101111111).
- $0x400$ (0b000000000000000000000000000) (d.h. das 15. Bit).

Der Eingabewert ist: $0x12340678$ (0b10010001101000000011001111000). Wir beobachten, wie der Wert geladen wird:

Abbildung 1.95: OllyDbg: der Wert wird nach ECX geladen
Das 15. Bit ist gesetzt: 0x1234678 (0b10010001101000100011000111000).

Abbildung 1.96: OllyDbg: OR wurde ausgeführt
Der Wert wird erneut geladen (da der Compiler nicht optimiert hat):

Abbildung 1.97: OllyDbg: der Wert wird erneut nach EDX geladen
Das 10. Bit wurde gelöscht (oder, mit anderen Worten: alle Bite außer dem 10. wurden stehengelassen) und das Endergebnis ist 0x12344 (0b10010001101000100010001).

**German text placeholder** **MSVC**

Wenn wir das Beispiel mit MSVC mit Optimierung (/Ox) kompilieren, ist der Code noch kürzer:

<table>
<thead>
<tr>
<th>Listing 1.244: German text placeholder MSVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>public f</td>
</tr>
<tr>
<td>proc near</td>
</tr>
<tr>
<td>var_4 = dword ptr -4</td>
</tr>
<tr>
<td>arg_0 = dword ptr 8</td>
</tr>
<tr>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>sub esp, 10h</td>
</tr>
<tr>
<td>mov eax, [ebp+arg_0]</td>
</tr>
<tr>
<td>mov [ebp+var_4], eax</td>
</tr>
<tr>
<td>mov [ebp+var_4], 0xffffddfH</td>
</tr>
<tr>
<td>mov eax, [ebp+var_4]</td>
</tr>
<tr>
<td>leave</td>
</tr>
<tr>
<td>ret</td>
</tr>
<tr>
<td>f endp</td>
</tr>
</tbody>
</table>

**German text placeholder** **GCC**

Untersuchen wir GCC 4.4.1 ohne Optimierung:

<table>
<thead>
<tr>
<th>Listing 1.245: German text placeholder GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>public f</td>
</tr>
<tr>
<td>proc near</td>
</tr>
<tr>
<td>var_4 = dword ptr -4</td>
</tr>
<tr>
<td>arg_0 = dword ptr 8</td>
</tr>
<tr>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>sub esp, 10h</td>
</tr>
<tr>
<td>mov eax, [ebp+arg_0]</td>
</tr>
<tr>
<td>mov [ebp+var_4], eax</td>
</tr>
<tr>
<td>mov [ebp+var_4], 0xffffddfH</td>
</tr>
<tr>
<td>mov eax, [ebp+var_4]</td>
</tr>
<tr>
<td>leave</td>
</tr>
<tr>
<td>ret</td>
</tr>
<tr>
<td>f endp</td>
</tr>
</tbody>
</table>
Obwohl es hier redundanten Code gibt, ist das Ergebnis kürzer als die MSVC Version ohne Optimierung. Jetzt aktivieren wir die Optimierung von GCC mit -03:

**German text placeholder GCC**

![Listing 1.246: German text placeholder GCC](image)


![Byte-Nummer:](image)

Der 16-Bit CPU 8086 Akkumulator wurde AX getauft und bestand aus zwei 8-Bit-Hälften—AL (niederes Byte) und AH (höheres Byte). In 80386 wurden fast alle Register auf 32 Bit erweitert und der Akkumulator wurde fortan EAX genannt, aber aus Kompatibilitätsgründen ist es immer noch möglich gezielt AX/AH/AL anzusprechen.

Da alle x86 CPUs Nachfolger der 16-Bit 8086 CPU sind, sind die älteren 16-Bit Opcodes kürzer als die neueren 32-Bit Opcodes. Aus diesem Grund benötigt der Befehl `or ah, 40h` nur 3 Bytes. Logischer wäre es zwar, hier `or eax, 04000h` zu verwenden, aber dieser Befehl würde 5 oder sogar 6 Byte (falls das Register im ersten Operanden nicht EAX ist) verbrauchen.

**German text placeholder GCC und regparm**

Es wäre noch kürzer, wenn man die Optimierung mit -03 anschaltet und regparm=3 setzt.

![Listing 1.247: German text placeholder GCC](image)

Das erste Argument ist schon nach EAX geladen worden, sodass es möglich ist, damit direkt weiterzuarbeiten. Bemerkenswert ist, dass sowohl der Funktionsprolog (`push ebp / mov ebp,esp`) als auch der Funktionsepiolog (`pop ebp`) hier wegfallen können. GCC ist aber möglicherweise nicht gut genug um eine solche Code Optimierung hier durchzuführen. Auf jeden Fall sind solche kurzen Funktion am besten als inline functions (?? on page ??) zu kennzeichnen.

**ARM + German text placeholder Keil 6/2013 (German text placeholder)**
Der Befehl **BIC** (**Bitwise bit Clear**) dient zum Löschen spezifischer Bits. Er arbeitet wie ein AND Befehl mit invertierten Operanden. Er entspricht also einem NOT +AND Befehlspaar.

**ORR** bedeutet „logical or“ und ist analog zu **OR** in x86.

So weit, so gut.

**ARM + German text placeholder Keil 6/2013 (German text placeholder)**

Listing 1.249: German text placeholder Keil 6/2013 (German text placeholder)

| 01 21 89 03 | MOVS R1, 0x4000 |
| 08 43      | ORRS R0, R1     |
| 49 11      | ASRS R1, R1, #5 ; erzeuge 0x200 und speichere in R1 |
| 88 43      | BICS R0, R1     |
| 70 47      | BX LR           |

Es scheint, dass Keil entschieden hat, dass der Code im Thumb mode, der 0x200 statt 0x4000 verwendet, kompakter ist, als einer, der 0x200 in ein beliebiges Register schreibt.

Das ist der Grund dafür, dass dieser Wert mithilfe von **ASRS** (**German text placeholder**) als 0x4000 ▷ 5 berechnet wird.

**ARM + German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**

Listing 1.250: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)

| 42 0C C0 E3 | BIC R0, R0, #0x4200 |
| 01 09 80 E3 | ORR R0, R0, #0x4000 |
| 1E FF 2F E1 | BX LR               |

Der Code, der von LLVM erzeugt wurde, könnte als Quellcode etwa wie folgt ausgesehen haben:

```c
REMOVE_BIT (rt, 0x4200);
SET_BIT (rt, 0x4000);
```

Er macht genau das, was wir erwarten. Aber woher das 0x4200? Möglicherweise handelt es sich um ein Artefakt des Optimierers von LLVM.

### Möglicherweise also ein Fehler des Optimierers im Compiler; aber der erzeugte Code funktioniert trotzdem korrekt.

Mehr zu Compiler-Anomalien hier: (10.4 on page 512).

**German text placeholder** Xcode 4.6.3 (LLVM) im Thumb mode erzeugt identischen Code.

**ARM: Mehr zum Befehl BIC**

Überarbeiten wir unser Beispiel ein wenig:

```c
int f(int a) {
    int rt=a;
    REMOVE_BIT (rt, 0x1234);
    return rt;
}
```

Hier wurde der LLVM Build 2410.2.00 mit Apple Xcode 4.6.3 gebündelt.
Jetzt erzeugt der optimierende Keil 5.03 im ARM mode folgenden Code:

```
f PROC
    BIC    r0, r0, #0x1000
    BIC    r0, r0, #0x234
    BX     lr
ENDP
```

Es gibt zwei BIC Befehle, d.h. die Bits 0x1234 werden in zwei Durchgängen gelöscht.
Das liegt daran, dass es nicht möglich ist, 0x1234 in einem einzigen BIC Befehl zu kodieren; deshalb müssen 0x1000 und 0x234 getrennt werden.

**ARM64: German text placeholder GCC (Linaro) 4.9**

German text placeholder GCC compiling für ARM64 kann den Befehl AND anstelle von BIC verwenden:

```
Listing 1.251: German text placeholder GCC (Linaro) 4.9

f:
    and  w0, w0, -513 ; 0xFFFFFFFFFFFFFDFF
    orr  w0, w0, 16384 ; 0x4000
    ret
```

**ARM64: German text placeholder GCC (Linaro) 4.9**

German text placeholder GCC erzeugt mehr redundanten Code; dieser funktiniert aber wie die optimierte Variante:

```
Listing 1.252: German text placeholder GCC (Linaro) 4.9

f:
    sub  sp, sp, #32
    str  w0, [sp,12]
    ldr  w0, [sp,28]
    str  w0, [sp,28]
    ldr  w0, [sp,28]
    orr  w0, w0, 16384 ; 0x4000
    str  w0, [sp,28]
    ldr  w0, [sp,28]
    and  w0, w0, -513 ; 0xFFFFFFFFFFFFFDFF
    str  w0, [sp,28]
    ldr  w0, [sp,28]
    add  sp, sp, 32
    ret
```

**MIPS**

```
Listing 1.253: German text placeholder GCC 4.4.5 (IDA)

f:
    ; $a0=a
    ori  $a0, 0x4000
    ; $a0=a|0x4000
    li   $v0, 0xFFFFFDFF
    jr   $ra
    and  $v0, $a0, $v0
    ; am Ende gilt: $v0 = $a0 & $v0 = a|0x4000 & 0xFFFFFDFF
```

ORI ist natürlich ein OR Befehl. Das „I“ im Befehlsnamen bedeutet, dass der Wert in den Maschinencode eingebettet wird.

Danach finden wir AND. Hier kann nicht ANDI verwendet werden, da es nicht möglich ist, die Zahl 0xFFFFFDFF in einen einzigen Befehl einzubetten, sodass der Compiler zunächst 0xFFFFFDFF in das Register $V0 lädt und dann ein AND erzeugt, das alle seine Eingabewerte aus den Registern entnimmt.
1.19.3 German text placeholder

Bitverschiebungen sind in C/C++ mit den Befehlen « und » implementiert. Die x86 ISA verwendet die Befehle SHL (SHift Left) und SHR (SHift Right) zu diesem Zweck. Schiebebefehle werden oft bei der Division und Multiplikation mit Potenzen von 2 2 (d.h. 1,2,4,8, etc.) verwendet: 1.16.1 on page 189, 1.16.2 on page 194.

Schiebebefehle sind auch wichtig, da sie oft für die Isolation einzelnes Bits oder für die Konstruktion von Werten aus mehreren einzelnen Bits verwendet werden.

1.19.4 Setzen und Löschen einzelner Bits: FPU Beispiele

In der folgenden Form werden die Bits in einem float gemäß IEEE 754 Format abgelegt:

\[
\begin{array}{c|c|c}
\text{S} & \text{31-30} & \text{23-22} \\
\hline
\end{array}
\]

Das Vorzeichen der Zahl ist im MSB kodiert. Wir fragen uns, ob es möglich ist, das Vorzeichen einer Fließkommazahl ohne FPU Befehle zu ändern.

```c
#include <stdio.h>

float my_abs (float i)
{
    unsigned int tmp=(*(unsigned int*)&i) & 0x7FFFFFFF;
    return *(float*)&tmp;
};

float set_sign (float i)
{
    unsigned int tmp=(*(unsigned int*)&i) | 0x80000000;
    return *(float*)&tmp;
};

float negate (float i)
{
    unsigned int tmp=(*(unsigned int*)&i) ^ 0x80000000;
    return *(float*)&tmp;
};

int main()
{
    printf ("my_abs():\n");
    printf ("%f\n", my_abs (123.456));
    printf ("%f\n", my_abs (-456.123));
    printf ("set_sign():\n");
    printf ("%f\n", set_sign (123.456));
    printf ("%f\n", set_sign (-456.123));
    printf ("negate():\n");
    printf ("%f\n", negate (123.456));
    printf ("%f\n", negate (-456.123));
};
```

Wir brauchen dieser Trickserei in C/C++ um von oder in float Werte ohne tatsächliche Konvertierung zu kopieren. Hier gibt es also drei Funktionen: my_abs() setzt das MSB; set_sign() setzt das MSB! und negate() kippt es.

XOR kann verwendet werden, um ein Bit zu kippen: ?? on page ??.

x86

Der Code ist einfach:

Listing 1.254: German text placeholder MSVC 2012

\[\text{MSB!}\]

130
Ein Eingabewert von Typ float wird vom Stack gelesen, aber wie ein Integerwert behandelt. AND und OR resetten und setzen das gewünschte Bit und XOR kippt es. Schließlich wird der modifizierte Wert nach ST0 geladen, das Fließkommazahlen über dieses Register zurückgegeben werden.

Betrachten wir den optimierenden MSVC 2012 für x64:

Listing 1.255: German text placeholder MSVC 2012 x64

Der Eingabewert wird nach XMM0 übergeben und dann auf den lokalen Stack kopiert. Wir finden hier einige für uns neue Befehle: BTR, BTS und BTC.
Diese Befehle werden zum Resetten (BTR), Setzen (BTS) und Invertieren (oder Komplementieren: BTC) einzelner Bits verwendet. Das 31. Bit von 0 gezählt ist das MSB!

Schließlich wird das Ergebnis nach XMM0 kopiert, da Fließkommazahlen in einer Win64 Umgebung über das Register XMM0 zurückgegeben werden.

**MIPS**

GCC 4.4.5 für MIPS erzeugt im Großen und Ganzen den gleichen Code:

```
Listing 1.256: German text placeholder GCC 4.4.5 (IDA)

my_abs:
    ; hole 1 vom Koprozessor:
    mfc1 $v1, $f12
    li $v0, 0x7FFFFFFF
    $v0=0x7FFFFFFF
    ; führe AND aus:
    and $v0, $v1
    ; kopiere 1 zum Koprozessor 1:
    mfc1 $v0, $f0
    ; return
    jr $ra
    or $at, $zero ; branch delay slot

set_sign:
    ; hole 1 vom Koprozessor:
    mfc1 $v0, $f12
    lui $v1, 0x80000000
    $v1=0x80000000
    ; führe OR aus:
    or $v0, $v1, $v0
    ; kopiere 1 zum Koprozessor:
    mfc1 $v0, $f0
    ; return
    jr $ra
    or $at, $zero ; branch delay slot

negate:
    ; hole 1 vom Koprozessor:
    mfc1 $v0, $f12
    lui $v1, 0x80000000
    $v1=0x80000000
    ; do XOR:
    xor $v0, $v1, $v0
    ; kopiere 1 zum Koprozessor:
    mfc1 $v0, $f0
    ; return
    jr $ra
    or $at, $zero ; branch delay slot
```

Ein einzelner LUI Befehl wird verwendet, um 0x80000000 in ein Register zu laden, den LUI löscht die niedersten 16 Bits und da diese ohnehin Nullen in der Konstanten entsprechen genügt hier ein LUI ohne nachfolgendes ORI.

**ARM**

```
Listing 1.257: German text placeholder Keil 6/2013 (German text placeholder)

my_abs PROC
    ; lösche Bit:
    BIC r0, r0, #0x80000000
    BX lr
ENDP

set_sign PROC
    ; führe OR aus:
```

Germantextplaceholder Keil 6/2013 (Germantextplaceholder)
So weit, so gut. ARM verfügt über den `BIC` Befehl, der ausdrücklich spezifizierte Bits löscht. `EOR` ist in ARM der Name des Befehls für `XOR` („exklusives OR“).

**German text placeholder Keil 6/2013 (German text placeholder)**

Listing 1.258: German text placeholder Keil 6/2013 (German text placeholder)

```assembly
my_abs PROC
    LSLS r0,r0,#1
    ; r0=i<<1
    LSRS r0,r0,#1
    ; r0=(i<<1)>>1
    BX lr
ENDP

set_sign PROC
    MOVS r1,#1
    ; r1=1
    LSLS r1,r1,#31
    ; r1=1<<31=0x80000000
    ORRS r0,r0,r1
    ; r0=r0 | 0x80000000
    BX lr
ENDP

negate PROC
    MOVS r1,#1
    ; r1=1
    LSLS r1,r1,#31
    ; r1=1<<31=0x80000000
    EORS r0,r0,r1
    ; r0=r0 ^ 0x80000000
    BX lr
ENDP
```

Thumb mode im ARM verwendet 16-Bit-Befehle und da in diesen nicht viele Daten kodiert werden können, wird hier ein `MOVS/LSLS` Befehlspaar benötigt, um die Konstante `0x80000000` zu konstruieren. Das Befehlspaar funktioniert wie folgt: `i << 31 = 0x80000000`.

Der Code von `my_abs` ist seltsam und entspricht tatsächlich dem folgenden Ausdruck: `(i << 1) >> 1`. Dieser Ausdruck scheint zunächst bedeutungslos. Wenn aber `input << 1` ausgeführt wird, befinden sich alle Bits an ihren korrekten Plätzen, nur dass das `MSB` null ist, da alle neuen Bits aus, die durch den Schiebebefehl eingefügt werden, stets Nullen sind. Auf diese Weise löscht das Befehlspaar `LSLS/LSRS` das `MSB`.

**German text placeholder GCC 4.6.3 (Raspberry Pi, German text placeholder)**

Listing 1.259: German text placeholder GCC 4.6.3 for Raspberry Pi (German text placeholder)

```assembly
my_abs
    ; Kopiere von S0 nach R2:
    FMRS R2, S0
    ; lösche Bit:
    BIC R3, R2, #0x80000000
    ; kopiere von R3 nach S0:
    FMSR S0, R3
    BX LR
```
Lassen wir den Raspberry Pi Linux in QEMU laufen und emulieren eine ARM FPU, dann werden hier S-Register anstelle der R-Register für den Umgang mit Fließkommazahlen verwendet.

Der Befehl FMRS kopiert Daten von GPR zur FPU und zurück my_abs() und set_sign() sehen wie erwartet aus, aber was ist mit negate()? Warum wird hier ADD anstelle von XOR verwendet?

Es ist auf den ersten Blick schwer zu glauben, aber der Befehl ADD register, 0x80000000 entspricht XOR register, 0x80000000. Erinnern wir uns an das Ziel des Befehls: Das Ziel ist es, das MSB! zu invertieren, also kümmern wir uns zunächst nicht um den XOR Befehl. Aus der Schulmathematik wissen wir, dass die Addition von Werten wie z.B. 1000 die letzten drei Stellen einer Zahl nicht verändert. Zum Beispiel gilt: 1234567 + 10000 = 1244567 (die letzten vier Stellen können sich nicht verändern).

Hier arbeiten wir mit Binärzahlen und 0x80000000 ist 0b10000000000000000000000000000000, d.h. hier sind nur das höchste Bit gesetzt.

Eine Addition von 0x80000000 zu einem anderen Werte kann also nie die niedersten 31 Bit verändern, sondern nur das MSB!. Addieren wir 1 zu 1, erhalten wir 1.

Addieren wir 1 zu 1, erhalten wir 0b10 in binär, aber das 32. Bit (von 0 gezählt) wird fallengelassen, da unsere Register eine Breite von 32 Bit haben, sodass das Ergebnis 0 ist. Deshalb kann in diesem Fall XOR durch ADD ersetzt werden.

Es ist schwer nachzuvollziehen, warum GCC diese Ersetzung vorgenommen hat, aber sie funktioniert tadellos.

### 1.19.5 Gesetzte Bits zählen

Hier ist ein einfaches Beispiel einer Funktion, die die Anzahl der gesetzten Bits in einem Eingabewert zählt. Diese Operation wird auch „population count“ genannt.

```c
#include <stdio.h>

#define IS_SET(flag, bit) ((flag) & (bit))

int f(unsigned int a) {
    int i;
    int rt=0;
    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;
    return rt;
}

int main() {
    f(0x12345678); // test
}
```

-moderner x86 CPUs (die SSE4 unterstützen) haben zu diesem Zweck sogar einen eigenen POPCNT Befehl
In dieser Schleife wird der Wert von $i$ schrittweise von 0 bis 31 erhöht, sodass der Ausdruck $1 \ll i$ von 1 bis $0x80000000$ zählt. In natürlicher Sprache würden wir diese Operation als *verschiebe 1 um n Bits nach links* beschreiben. Mit anderen Worten: Der Ausdruck $1 \ll i$ erzeugt alle möglichen Bitpositionen in einer 32-Bit-Zahl. Das freie Bit auf der rechten Seite wird jeweils gelöscht.

Hier ist eine Tabelle mit allen Werten von $1 \ll i$ für $i = 0 \ldots 31$:

<table>
<thead>
<tr>
<th>C/C++ Ausdruck</th>
<th>Zweierpotenz</th>
<th>Dezimalzahl</th>
<th>Hexadezimalzahl</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \ll 0$</td>
<td>$2^0$</td>
<td>1</td>
<td>0x1</td>
</tr>
<tr>
<td>$1 \ll 1$</td>
<td>$2^1$</td>
<td>2</td>
<td>0x2</td>
</tr>
<tr>
<td>$1 \ll 2$</td>
<td>$2^2$</td>
<td>4</td>
<td>0x4</td>
</tr>
<tr>
<td>$1 \ll 3$</td>
<td>$2^3$</td>
<td>8</td>
<td>0x8</td>
</tr>
<tr>
<td>$1 \ll 4$</td>
<td>$2^4$</td>
<td>16</td>
<td>0x10</td>
</tr>
<tr>
<td>$1 \ll 5$</td>
<td>$2^5$</td>
<td>32</td>
<td>0x20</td>
</tr>
<tr>
<td>$1 \ll 6$</td>
<td>$2^6$</td>
<td>64</td>
<td>0x40</td>
</tr>
<tr>
<td>$1 \ll 7$</td>
<td>$2^7$</td>
<td>128</td>
<td>0x80</td>
</tr>
<tr>
<td>$1 \ll 8$</td>
<td>$2^8$</td>
<td>256</td>
<td>0x100</td>
</tr>
<tr>
<td>$1 \ll 9$</td>
<td>$2^9$</td>
<td>512</td>
<td>0x200</td>
</tr>
<tr>
<td>$1 \ll 10$</td>
<td>$2^{10}$</td>
<td>1024</td>
<td>0x400</td>
</tr>
<tr>
<td>$1 \ll 11$</td>
<td>$2^{11}$</td>
<td>2048</td>
<td>0x800</td>
</tr>
<tr>
<td>$1 \ll 12$</td>
<td>$2^{12}$</td>
<td>4096</td>
<td>0x1000</td>
</tr>
<tr>
<td>$1 \ll 13$</td>
<td>$2^{13}$</td>
<td>8192</td>
<td>0x2000</td>
</tr>
<tr>
<td>$1 \ll 14$</td>
<td>$2^{14}$</td>
<td>16384</td>
<td>0x4000</td>
</tr>
<tr>
<td>$1 \ll 15$</td>
<td>$2^{15}$</td>
<td>32768</td>
<td>0x8000</td>
</tr>
<tr>
<td>$1 \ll 16$</td>
<td>$2^{16}$</td>
<td>65536</td>
<td>0x10000</td>
</tr>
<tr>
<td>$1 \ll 17$</td>
<td>$2^{17}$</td>
<td>131072</td>
<td>0x20000</td>
</tr>
<tr>
<td>$1 \ll 18$</td>
<td>$2^{18}$</td>
<td>262144</td>
<td>0x40000</td>
</tr>
<tr>
<td>$1 \ll 19$</td>
<td>$2^{19}$</td>
<td>524288</td>
<td>0x80000</td>
</tr>
<tr>
<td>$1 \ll 20$</td>
<td>$2^{20}$</td>
<td>1048576</td>
<td>0x100000</td>
</tr>
<tr>
<td>$1 \ll 21$</td>
<td>$2^{21}$</td>
<td>2097152</td>
<td>0x200000</td>
</tr>
<tr>
<td>$1 \ll 22$</td>
<td>$2^{22}$</td>
<td>4194304</td>
<td>0x400000</td>
</tr>
<tr>
<td>$1 \ll 23$</td>
<td>$2^{23}$</td>
<td>8388608</td>
<td>0x800000</td>
</tr>
<tr>
<td>$1 \ll 24$</td>
<td>$2^{24}$</td>
<td>16777216</td>
<td>0x10000000</td>
</tr>
<tr>
<td>$1 \ll 25$</td>
<td>$2^{25}$</td>
<td>33554432</td>
<td>0x20000000</td>
</tr>
<tr>
<td>$1 \ll 26$</td>
<td>$2^{26}$</td>
<td>67108864</td>
<td>0x40000000</td>
</tr>
<tr>
<td>$1 \ll 27$</td>
<td>$2^{27}$</td>
<td>134217728</td>
<td>0x80000000</td>
</tr>
<tr>
<td>$1 \ll 28$</td>
<td>$2^{28}$</td>
<td>268435456</td>
<td>0x100000000</td>
</tr>
<tr>
<td>$1 \ll 29$</td>
<td>$2^{29}$</td>
<td>536870912</td>
<td>0x200000000</td>
</tr>
<tr>
<td>$1 \ll 30$</td>
<td>$2^{30}$</td>
<td>1073741824</td>
<td>0x400000000</td>
</tr>
<tr>
<td>$1 \ll 31$</td>
<td>$2^{31}$</td>
<td>2147483648</td>
<td>0x8000000000</td>
</tr>
</tbody>
</table>

Diese Konstanten (Bitmasken) tauchen im Code oft auf und ein Reverse Engineer muss in der Lage sein, sie schnell und sicher zu erkennen.

Es dazu jedoch nicht notwendig, die Dezimalzahlen (Zweierpotenzen) größer 65535 auswendig zu kennen. Die hexadezimalen Zahlen sind leicht zu merken.

Die Konstanten werden häufig verwendet um Flags einzelnen Bits zuzuordnen. Hier ist zum Beispiel ein Auszug aus ssl_private.h aus dem Quellcode von Apache 2.4.6:

```c
/*
 * Define the SSL options
 */
#define SSL_OPT_NONE (0)
#define SSL_OPT_RELSET (1<<0)
#define SSL_OPT_STDENVVARS (1<<1)
#define SSL_OPT_EXPORTCERTDATA (1<<3)
#define SSL_OPT_FAKEBASICAUTH (1<<4)
#define SSL_OPT_STRICTREQUIRE (1<<5)
#define SSL_OPT_OPTRENEGOTIATE (1<<6)
#define SSL_OPT_LEGACYDNFORMAT (1<<7)
```

Zurück zu unserem Beispiel.

Das Makro `IS_SET` prüft auf Anwesenheit von Bits in `a`. 

297
Das Makro IS_SET entspricht dabei dem logischen (AND) und gibt 0 zurück, wenn das entsprechende Bit nicht gesetzt ist, oder die Bitmaske, wenn das Bit gesetzt ist. Der Operator if() wird in C/C++ ausgeführt, wenn der boolesche Ausdruck nicht null ist (er könnte sogar 123456 sein), weshalb es meistens richtig funktioniert.

**x86**

**MSVC**

Kompilieren wir (MSVC 2010):

<table>
<thead>
<tr>
<th>Listing 1.260: MSVC 2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>_rt$ = -8 ; size = 4</td>
</tr>
<tr>
<td>_i$ = -4 ; size = 4</td>
</tr>
<tr>
<td>_a$ = 8 ; size = 4</td>
</tr>
<tr>
<td>PROC</td>
</tr>
<tr>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>sub esp, 8</td>
</tr>
<tr>
<td>mov DWORD PTR _rt$[ebp], 0</td>
</tr>
<tr>
<td>mov DWORD PTR _i$[ebp], 0</td>
</tr>
<tr>
<td>jmp SHORT $LN4@f</td>
</tr>
<tr>
<td>$LN3@f:</td>
</tr>
<tr>
<td>mov eax, DWORD PTR _i$[ebp] ; erhöhe i</td>
</tr>
<tr>
<td>add eax, 1</td>
</tr>
<tr>
<td>mov DWORD PTR _i$[ebp], eax</td>
</tr>
<tr>
<td>$LN4@f:</td>
</tr>
<tr>
<td>cmp DWORD PTR _i$[ebp], 32 ; 00000020H</td>
</tr>
<tr>
<td>jge SHORT $LN2@f        ; Schleife beendet?</td>
</tr>
<tr>
<td>mov edx, 1</td>
</tr>
<tr>
<td>mov ecx, DWORD PTR _i$[ebp]</td>
</tr>
<tr>
<td>shl edx, cl             ; EDX=EDX&lt;&lt;CL</td>
</tr>
<tr>
<td>and edx, DWORD PTR _a$[ebp]</td>
</tr>
<tr>
<td>je SHORT $LN1@f         ; was das Ergebnis des AND Befehls 0?</td>
</tr>
<tr>
<td>mov eax, DWORD PTR _rt$[ebp] ; nein: ungleich 0</td>
</tr>
<tr>
<td>add eax, 1              ; erhöhe rt</td>
</tr>
<tr>
<td>mov DWORD PTR _rt$[ebp], eax</td>
</tr>
<tr>
<td>$LN1@f:</td>
</tr>
<tr>
<td>jmp SHORT $LN3@f</td>
</tr>
<tr>
<td>$LN2@f:</td>
</tr>
<tr>
<td>mov eax, DWORD PTR _rt$[ebp]</td>
</tr>
<tr>
<td>mov esp, ebp</td>
</tr>
<tr>
<td>pop ebp</td>
</tr>
<tr>
<td>ret 0</td>
</tr>
<tr>
<td>ENDP</td>
</tr>
</tbody>
</table>
Betrachten wir dieses Beispiel in OllyDbg. Sei der Eingabewert dabei 0x12345678.

Für $i = 1$ sehen wir, wie $i$ nach ECX geladen wird:

Abbildung 1.99: OllyDbg: $i = 1$, $i$ wird nach ECX geladen

EDX ist 1. SHL wird jetzt ausgeführt.
SHL wurde ausgeführt:

Abbildung 1.100: OllyDbg: $i = 1$, $\text{EDX} = 1 \ll 1 = 2$

EDX enthält $1 \ll 1$ (oder 2). Hierbei handelt es sich um eine Bitmaske.
AND setzt ZF auf 1, was bedeutet, dass der Eingabewert (0x12345678) mit 2 verUNDEt wird. Das Ergebnis ist 0:

Abbildung 1.101: OllyDbg: i = 1, ist hier das Bit im Eingabewert gesetzt? Nein. (ZF =1)

Es gibt hier also kein entsprechendes Bit im Eingabewert.

Das Codestück, welches den Zähler erhöht, wird also nicht ausgeführt: Der JZ Befehl überspringt es.
Verfolgen wir den Ablauf ein bisschen weiter bis $i$ den Wert 4 hat. SHL wird jetzt ausgeführt:

Abbildung 1.102: OllyDbg: $i = 4$, $i$ wird nach ECX geladen
Abbildung 1.103: OllyDbg: $i = 4$, EDX $= 1 \ll 4 = 0x10$

Hierbei handelt es sich um eine weitere Bitmaske.
AND wird ausgeführt:

Abbildung 1.104: OllyDbg: \( i = 4 \), ist hier das Bit im Eingabewert gesetzt? Ja. (ZF = 0)

ZF ist 0, da das Bit im Eingabewert gesetzt ist.
Tatsächlich gilt \( 0x12345678 \& 0x10 = 0x10 \).

Das Bit wird gezählt: der Sprung wird nicht ausgeführt und der Zähler wird erhöht.

Die Funktion liefert den Wert 13 zurück. Dies entspricht der Anzahl der in der binären Darstellung von \( 0x12345678 \) gesetzten Bits.

GCC

Kompilieren wir das Beispiel mit GCC 4.4.1:

Listing 1.261: GCC 4.4.1

```
public f
f proc near
rt = dword ptr -0Ch
i = dword ptr -8
arg_0 = dword ptr 8

push ebp
mov ebp, esp
push ebx
sub esp, 10h
mov [ebp+rt], 0
mov [ebp+i], 0
jmp short loc_80483EF

loc_80483D0:
  mov eax, [ebp+i]
  mov edx, 1
  mov ebx, edx
  mov ecx, eax
  shl ebx, cl
  mov eax, ebx
  and eax, [ebp+arg_0]
  test eax, eax
  jz short loc_80483EB
  add [ebp+rt], 1

loc_80483EB:
  add [ebp+i], 1
```
loc_80483EF:
    cmp [ebp+i], 1Fh
    jle short loc_80483D0
    mov eax, [ebp+rt]
    add esp, 10h
    pop ebx
    pop ebp
    retn
f    endp

x64
Verändern wir das Beispiel ein wenig um es auf 64 Bit zu erweitern:

```c
#include <stdio.h>
#include <stdint.h>

#define IS_SET(flag, bit) ((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
}
```

German text placeholder GCC 4.8.2

So weit, so einfach.

```
Listing 1.262: German text placeholder GCC 4.8.2
```

f:
   push   rbp
   mov    rbp, rsp
   mov    QWORD PTR [rbp-24], rdi ; a
   mov    DWORD PTR [rbp-12], 0 ; rt=0
   mov    QWORD PTR [rbp-8], 0 ; i=0
   jmp .L2
 .L4:
   mov    rax, QWORD PTR [rbp-8]
   mov    rdx, QWORD PTR [rbp-24]
   ; RAX = 1, RDX = a
   mov    ecx, eax
   ; ECX = 1
   shr    rdx, cl
   ; RDX = RDX>>CL = a>>i
   mov    rax, rdx
   ; RAX = a>>i
   and    eax, 1
   ; EAX = EAX&1 = (a>>i)&1
   test    rax, rax
   ; ist das letzte Bit 0?
   ; überspringe nächsten ADD Befehl, wenn es so war.
   je .L3
 .L3:
   add    DWORD PTR [rbp-12], 1 ; rt++
   add    QWORD PTR [rbp-8], 1 ; i++
 .L2:
   cmp    QWORD PTR [rbp-8], 63 ; i<63?
   jbe .L4 ; springe zum Beginn der Schleife, falls wahr
   mov    eax, DWORD PTR [rbp-12] ; return rt
   pop    rbp
```
der Code ist kürzer, birgt aber eine Besonderheit.


Deshalb wird die Erhöhung des Zählers in jedem Durchlauf der Schleife durchgeführt, d.h. 64 mal, ohne dass eine Abhängigkeit vom Eingabewert besteht.

Der Vorteil dieses Code ist, dass er nur einen bedingten Sprung enthält (am Ende der Schleife) anstatt zwei Sprüngen (Überspringen des Erhöhens von „rt“ und Ende der Schleife). Der Code ist somit auf modernen CPUs mit Branch Predictors möglicherweise schneller: ?? on page ??.

Hier wird der Befehl ROL anstelle von SHL verwendet, welches einer „Linksrotation“ anstatt einer „Linksverschiebung“ entspricht. In diesem Beispiel entspricht ROL einem SHL.

Für mehr Informationen zu Rotationsbefehlen siehe: .1.6 on page 539.

R8 zählt hier von 64 auf 0 herunter. Dies entspricht dem invertierten i.

Hier ist eine Tabelle einiger Register während der Ausführung des Programms:

<table>
<thead>
<tr>
<th>RDX</th>
<th>R8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000000001</td>
<td>64</td>
</tr>
<tr>
<td>0x0000000000000002</td>
<td>63</td>
</tr>
<tr>
<td>0x0000000000000004</td>
<td>62</td>
</tr>
<tr>
<td>0x0000000000000008</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0x4000000000000000</td>
<td>2</td>
</tr>
<tr>
<td>0x8000000000000000</td>
<td>1</td>
</tr>
</tbody>
</table>

Am Ende finden wir den Befehl FATRET, der hier schon erklärt wurde: 1.19.5 on the previous page.

**German text placeholder MSVC 2012**

Übertragung des MSVC 2012Listing 1.265: German text placeholder MSVC 2012

Der optimierende MSVC 2012 erzeugt fast den gleichen Code wie MSVC 2012, generiert aber aus irgendeinem Grund zwei identischen Rümpfe für die Schleifen und die Schleife zählt nun bis 32 anstatt 64.


Trotzdem ist solcher Code relevant um zu zeigen, dass der Output des Compilers manchmal sehr merkwürdig und unlogisch sein kann und dennoch tadellos funktioniert.

**ARM + German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**
Listing 1.266: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)

`loc_2E54`

```
MOV R1, R0
MOV R0, #0
MOV R2, #1
MOV R3, R0
loc_2E54
TST R1, R2,LSL R3 ; setze Flags entsprechend R1 & (R2 << R3)
(R2<<R3) ADD R3, R3, #1 ; R3++
ADDNE R0, R0, #1 ; wenn ZF Flag von TST gelöscht wurde, dann R0++
CMP R3, #32
BNE loc_2E54
BX LR
```

TST entspricht dem Befehl TEST in x86.

Wie bereits in (?? on page ??) besprochen gibt es zwei verschiedene Schiebebefehle im ARM mode. Zusätzlich gibt es aber noch die Suffixe LSL (Logical Shift Left), LSR (Logical Shift Right), ASR (Arithmetic Shift Right), ROR (Rotate Right) und RRX (Rotate Right with Extend), die an Befehle wie MOV, TST, CMP, ADD, SUB, RSB135 angehängt werden können.

Diese Suffixe legen fest, wie und um wie viele Bits der zweite Operand verschoben werden soll.

Dadurch entspricht der Befehl „TST R1, R2,LSL R3“ hier $R1 \land (R2 \ll R3)$.

**ARM + German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**

Fast das gleiche, aber hier werden zwei LSL.W/TST Befehle anstelle eines einzelnen TST verwendet, da es im Thumb mode nicht möglich ist, den Suffix LSL direkt in TST zu definieren.

`loc_2F7A`

```
MOV R1, R0
MOVS R0, #0
MOVS R3, R3, #0
loc_2F7A
LSL.W R2, R9, R3
TST R2, R1
ADD.W R3, R3, #1
IT NE ADDNE R0, #1
CMP R3, #32
BNE loc_2F7A
BX LR
```

**ARM64 + German text placeholder GCC 4.9**

Betrachten wir ein 64-Bit-Beispiel, das wir bereits kennen: 1.19.5 on page 305.

Listing 1.267: German text placeholder GCC (Linaro) 4.8

```
f:
  mov w2, 0 ; rt=0
  mov x5, 1
  mov w1, w2
.L2:
  lsl x4, x5, x1 ; w4 = w5<<w1 = 1<i
  add w3, w2, 1 ; new rt=rt+1
  tst x4, x0 ; (1<i) & a
  add w1, w1, 1 ; i++
  ; war das Ergebnis von TST ungleich null?
  ; dann w2=w3 oder rt=new Rt.
  ; sonst: w2=w2 oder rt=rt (leeres Befehl)
  csel w2, w3, w2, ne
  cmp w1, 64 ; i<64?
  bne .L2 ; yes
  mov w0, w2 ; return rt
ret
```

135German text placeholder
Das Ergebnis ist ähnlich dem was GCC für x64 erzeugt: 1.263 on page 306.
Der Befehl CSEL steht für „Conditional SEl ect“. Er wählt eine von zwei Variablen abhängig von den durch TST gesetzten Flags aus und kopiert deren Wert nach W2, wo die Variable „rt“ gespeichert wird.

**ARM64 + German text placeholder GCC 4.9**

Wieder werden wir hier mit dem bereits bekannten 64-Bit-Beispiel arbeiten: 1.19.5 on page 305. Der Code ist umfangreicher als gewöhnlich.

Listing 1.268: German text placeholder GCC (Linaro) 4.8

```
f:       sub    sp, sp, #32  ; speichere "a" in der Register Save Area
str     x0, [sp,8]        ; rt=0
str     wrzr, [sp,24]     ; i=0
b       
.L2:
  ldr    w0, [sp,28]      ; X0 = X1<<X0 = 1<<i
  mov    x1, 1
  lsl    x0, x1, x0
  mov    x1, x0
  X1 = 1<<i
  ldr    x0, [sp,8]
  X0 = a
  and    x0, x1, x0
  X0 = X1&X0 = (1<<i) & a
  enthält X0 null? Dann springe zu .L3, lasse "rt" Inkrement aus
  cmp    x0, xzr
  beq    .L3
  rt++   
  ldr    w0, [sp,24]      
  add    w0, w0, 1
  str    w0, [sp,24]
.L3:
  i++   
  ldr    w0, [sp,28]
  add    w0, w0, 1
  str    w0, [sp,28]
.L2:
  ; i<=63? Dann springe zu .L4
  cmp    w0, 63
  ble    .L4
; return rt
  ldr    w0, [sp,24]
  add    sp, sp, 32
ret
```

**MIPS**

**German text placeholder GCC**

Listing 1.269: German text placeholder GCC 4.4.5 (IDA)

```
f:
    ; IDA vergibt keine Variablennamen, diese wurden manuell hinzugefügt:
    rt    = -0x10
    i     = -0xC
    var_4 = -4
    a     = 0

    addiu $sp, -0x18
    sw     $fp, 0x18+var_4($sp)
    move   $fp, $sp
    sw     $a0, 0x18+a($fp)
; initialisiere rt und i mit 0:
    sw     $zero, 0x18+rt($fp)
```
Umständlich: alle lokalen Variablen liegen auf dem lokalen Stack und werden bei jedem Zugriff neu geladen.

Der Befehl SLLV bedeutet „Shift Word Left Logical Variable“. Er unterscheidet sich von SLL nur dadurch, dass die Anzahl der Verschiebungen im SLL Befehl kodiert sind (und dadurch nicht veränderbar). SLLV hingegen erhält die Anzahl der Verschiebungen aus einem Register.

German text placeholder GCC

Hier ist es knapper gehalten. Warum aber gibt es zwei Schiebebefehle anstatt eines?

Es ist möglich, den ersten SLLV Befehl durch einen unbedingte Verzweigungsbebfehl zu ersetzen, der direkt zum zweiten SLLV springt. Das zieht aber einen zweiten Verzweigungsbebfehl nach sich und es ist stets vorteilhaft sich dieser zu entledigen: ?? on page ??.

Listing 1.270: German text placeholder GCC 4.4.5 (IDA)

```
mv   $a0, $a
; rt bleibt in $v:
mv   $v0, $zero
; i bleibt in $v1:
mv   $v1, $zero
```
**1.19.6 Fazit**

Analog zu den Schiebebefehlen `«` und `»` in C/C++ gibt es in x86 die Befehle SHR/SHL (für vorzeichenlose Werte) und SAR/SHL (für vorzeichenbehaftete Werte).

Die Schiebebefehle in ARM sind LSR/LSL (für vorzeichenlose Werte) und ASR/LSL (für vorzeichenbehaftete Werte).

Es sind bei manchen Befehlen auch mögliche Suffixe für die Verschiebung anzuhängen (diese heiße „data processing instructions“).

**Prüfung auf spezifisches Bit (zur Compilezeit bekannt)**

Prüfung, ob das Bit 0b10000000 (0x40) sich im Registerwert befindet:

**Listing 1.271: C/C++**

```c
if (input&0x40) ...
```

**Listing 1.272: x86**

```assembly
TEST REG, 40h
JNZ is_set
; Bit ist nicht gesetzt
```

**Listing 1.273: x86**

```assembly
TEST REG, 40h
JZ is_cleared
; Bit ist gesetzt
```

**Listing 1.274: ARM (German text placeholder)**

```assembly
TST REG, #0x40
BNE is_set
; Bit ist nicht gesetzt
```

Manchmal wird AND anstelle von TEST verwendet, aber die gesetzten Flags sind die gleichen.
Prüfung auf spezifisches Bit (zur Laufzeit angegeben)

Dies wird normalerweise durch den folgenden C/C++ Code gelöst (verschiebe Wert um $n$ Bits nach rechts und schneide dann niederwertigstes Bit ab):

**Listing 1.275: C/C++**

```c
if (((value)>>n)&1)
    ....
```

In x86 Code wird dies gewöhnlich wie folgt implementiert:

**Listing 1.276: x86**

```assembly
; REG=input_value
; CL=n
SHR REG, CL
AND REG, 1
```

Eine andere Möglichkeit: (verschiebe 1 Bit $n$-mal nach links, isoliere dieses Bit im Eingabewert und prüfe, ob es nicht 0 ist):

**Listing 1.277: C/C++**

```c
if (value & (1<<n))
    ....
```

In x86 Code wird dies gewöhnlich wie folgt implementiert:

**Listing 1.278: x86**

```assembly
; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG
```

Setzen eines spezifischen Bits (zur Compilerzeit bekannt)

**Listing 1.279: C/C++**

```c
value=value|0x40;
```

**Listing 1.280: x86**

```assembly
OR REG, 40h
```

**Listing 1.281: ARM (German text placeholder) and ARM64**

```assembly
ORR R0, R0, #0x40
```

Setzen eines spezifischen Bits (zur Laufzeit angegeben)

**Listing 1.282: C/C++**

```c
value=value|(1<<n);
```

In x86 Code wird dies gewöhnlich wie folgt implementiert:

**Listing 1.283: x86**

```assembly
; CL=n
MOV REG, 1
SHL REG, CL
OR input_value, REG
```
Löschen eines spezifischen Bits (zur Compilezeit bekannt)

Man verwendet einfach den AND Befehl mit dem invertierten Wert:

**Listing 1.284: C/C++**

```c
value = value & (~0x40);
```

**Listing 1.285: x86**

```assembly
AND REG, 0FFFFFFBFh
```

**Listing 1.286: x64**

```assembly
AND REG, 0FFFFFFFFFFFFFFBFh
```

Dies sorgt dafür, dass alle Bits bis auf eines gesetzt werden.

ARM im ARM mode verfügt über den Befehl BIC, der wie ein NOT + AND Befehlspaar arbeitet:

**Listing 1.287: ARM**

```assembly
BIC R0, R0, #0x40
```

Löschen eines spezifischen Bits (zur Laufzeit angegeben)

**Listing 1.288: C/C++**

```c
value = value & (~(1<<n));
```

**Listing 1.289: x86**

```assembly
; CL=n
MOV REG, 1
SHL REG, CL
NOT REG
AND input_value, REG
```

### 1.19.7 Übungen

- [http://challenges.re/67](http://challenges.re/67)
- [http://challenges.re/68](http://challenges.re/68)
- [http://challenges.re/69](http://challenges.re/69)
- [http://challenges.re/70](http://challenges.re/70)

### 1.20 Linearer Kongruenzgenerator als Pseudozufallszahlengenerator

Ein linearer Kongruenzgenerator ist die wohl einfachste Form der Zufallszahlenerzeugung. Er wird heutzutage nicht mehr so häufig eingesetzt\(^{136}\), kann aber, weil er so einfach ist (nur eine Multiplikation, eine Addition und eine AND-Operation), dennoch gut als Beispiel dienen.

\(^{136}\)Der Mersenne-Twister ist besser
Es gibt hier zwei Funktionen: die erste wird verwendet und den internen Zustand zu initialisieren und die zweite wird zum Erzeugen der Pseudozufallszahlen aufgerufen.


Definieren wir sie über den `#define` C/C++ Ausdruck. Es handelt sich um ein Makro.

Der Unterschied zwischen einem C/C++ Makro und einer Konstanten ist, dass alle Makros durch den C/C++ Präprozessor durch mit ihrem Wert ersetzt werden und dadurch im Gegensatz zu Variablen keinen Speicherplatz verbrauchen.

Eine Konstante ist im Gegensatz dazu eine nur lesbare Variable.

Es ist möglich einen Pointer (oder eine Adresse) einer Konstanten zu verwenden; das ist mit einem Makro nicht möglich.

Die letzte AND-Operation wird benötigt, da `my_rand()` gemäß C-Standard einen Wert zwischen 0 und 32767 zurückgeben muss.

Wenn man 32-Bit-Pseudozufallszahlen benötigt, kann die AND-Operation einfach weggelassen werden.

### 1.20.1 x86

Listing 1.290: German text placeholder MSVC 2013

Hier sehen wir, dass beide Konstanten in den Code eingebettet wurden. Es wurde kein Speicher für sie reserviert.
Die Funktion `my_srand()` kopiert ihren Eingabewert in die interne Variable `rand_state`.
`my_rand()` nimmt diese, berechnet den nächsten `rand_state`, schneidet ihn ab und belässt ihn im EAX Register.

Die nicht optimierte Version ist umfangreicher:

Listing 1.291: German text placeholder MSVC 2013

```
_BSS    SEGMENT
_rand_state DD 01H DUP (?)
_BSS     ENDS

_init$ = 8
_my_srand PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _init$[ebp]
    mov    DWORD PTR _rand_state, eax
    pop    ebp
    ret    0
_my_srand ENDP

_TEXT   SEGMENT
_my_rand PROC
    push   ebp
    mov    ebp, esp
    imul   eax, DWORD PTR _rand_state, 1664525
    mov    DWORD PTR _rand_state, eax
    add    ecx, DWORD PTR _rand_state
    add    ecx, 1013904223 ; 3c6ef35fH
    mov    DWORD PTR _rand_state, ecx
    mov    eax, DWORD PTR _rand_state
    and    eax, 32767 ; 00007fffH
    pop    ebp
    ret    0
_my_rand ENDP

_TEXT    ENDS
```

1.20.2 x64

Die x64 Version ist größtenteils identisch und verwendet 32-Bit-Register anstelle der 64-Bit-Register (wir arbeiten hier mit `int` Werten).

Die Funktion `my_srand()` nimmt seinen Eingabewert aus dem Register ECX und nicht vom Stack:

Listing 1.292: German text placeholder MSVC 2013 x64

```
_BSS    SEGMENT
_rand_state DD 01H DUP (?)
_BSS     ENDS

_init$ = 8
_my_srand PROC
    ; ECX = Eingabewert
    mov    DWORD PTR rand_state, ecx
    ret    0
_my_srand ENDP

_TEXT   SEGMENT
_my_rand PROC
    imul   eax, DWORD PTR rand_state, 1664525 ; 0019660dH
    add    eax, 1013904223 ; 3c6ef35fH
    mov    DWORD PTR rand_state, eax
    and    eax, 32767 ; 00007fffH
    ret    0
_my_rand ENDP

_TEXT    ENDS
```
GCC erzeugt fast den gleichen Code.

### 1.20.3 32-bit ARM

Listing 1.293: German text placeholder Keil 6/2013 (German text placeholder)

```assembly
my_srand PROC
    LDR r1,|L0.52| ; lade Pointer auf rand_state
    STR r0,[r1,#0] ; speichere rand_state
    BX lr
ENDP

my_rand PROC
    LDR r0,|L0.52| ; lade Pointer auf rand_state
    LDR r2,|L0.56| ; lade RNG_a
    LDR r1,[r0,#0] ; lade rand_state
    MUL r1,r2,r1
    LDR r2,|L0.60| ; lade RNG_c
    ADD r1,r1,r2
    STR r1,[r0,#0] ; speichere rand_state
    ; AND mit 0x7FFF:
    LSL r0,r1,#17
    LSR r0,r0,#17
    BX lr
ENDP

|L0.52|
DCD ||.data||

|L0.56|
DCD 0x0019660d

|L0.60|
DCD 0x3c6ef35f

AREA ||.data||, DATA, ALIGN=2

rand_state
DCD 0x00000000

Es ist nicht möglich 32-Bit-Konstanten in ARM Befehle einzubetten, sodass Keil diese extern speichern und dann zusätzlich laden muss. Eine interessante Sache ist, dass es ebenfalls nicht möglich ist, die Konstante 0x7FFF einzubetten. Was Keil dann tut, ist rand_state um 17 Bits nach links zu verschieben und dann um 17 Bits nach rechts zu Verschieben. Die entspricht Ausdruck (rand_state ≪ 17) ≫ 17 in C/C++. Es scheint eine nutzlose Operation zu sein, löscht aber die oberen 17 Bits und lässt die 15 niederen Bits intakt und das ist genau was wir wollen.

German text placeholder Keil für Thumb mode erzeugt fast den gleichen Code.

### 1.20.4 MIPS

Listing 1.294: German text placeholder GCC 4.4.5 (IDA)

```assembly
my_srand:
    ; speichere $a0 in rand_state:
    lui $v0, (rand_state >> 16)
    jr $ra
    sw $a0, rand_state

my_rand:
    ; lade rand_state nach $v0:
    lui $v1, (rand_state >> 16)
    lw $v0, rand_state
    or $at, $zero ; lade delay slot
    ; multipliziere rand_state in $v0 mit 1664525 (RNG_a):
    sll $a1, $v0, 2
    sll $a0, $v0, 4
    addu $a0, $a1, $a0
    sll $a1, $a0, 6
    subu $a0, $a1, $a0
```

316
addu $a0, $v0  
sll $a1, $a0, 5  
addu $a0, $a1  
sll $a0, $a0, 3  
addu $v0, $a0, $v0  
sll $a0, $v0, 2  
addu $v0, $a0

; addiere 1013904223 (RNG_c)
; der LI Befehl ist von IDA aus LUI und ORI zusammengesetzt
li $a0, 0x3C6EF35F  
addu $v0, $a0

; speichere in rand_state:
sw $v0, (rand_state & 0xFFFF)($v1)  
jr $ra  
andi $v0, 0x7FFF ; branch delay slot

Hier sehen wir nur eine Konstante (0x3C6EF35F oder 1013904223). Wo befindet sich die andere (1664525)?
Es scheint, dass die Multiplikation mit 1664525 nur durch Verschieben und Addieren durchgeführt wird.
Überprüfen wir diese Vermutung:

#define RNG_a 1664525
int f (int a)
{
    return a*RNG_a;
}

Listing 1.295: German text placeholder GCC 4.4.5 (IDA)

f:
    sll $v1, $a0, 2  
sll $v0, $a0, 4  
addu $v0, $v1, $v0  
sll $v1, $v0, 6  
subu $v0, $v1, $v0  
addu $v0, $a0  
sll $v1, $v0, 5  
addu $v0, $v1  
sll $v0, 3  
addu $a0, $v0, $a0  
sll $v0, $a0, 2  
jr $ra  
addu $v0, $a0, $v0 ; branch delay slot

Tatsächlich!

**MIPS Relocation**

Wir werden uns auch anschauen wie solche Operationen wie das Laden und Werten aus dem Speicher und das Speichern tatsächlich funktionieren.

Die Listings wurden mit IDA erzeugt, was einige Details versteckt.

Wir lassen objdump zweimal laufen: um das disassemblierte Listing und die Relocation List zu erhalten:

Listing 1.296: German text placeholder GCC 4.4.5 (objdump)

# objdump -D rand_O3.o  
...  
00000000 <my_srand>:  
0:  3c020000 lui v0,0x0  
4:  03e00008 jr ra  
8:  ac440000 sw a0,0(v0)

00000000c <my_rand>:  
c:  3c030000 lui v1,0x0  
10:  8c620000 lw v0,0(v1)  
14:  002000825 move at,at

317
Betrachten wir die beiden Relocations für die Funktion `my_srand()`.

Die erste für die Adresse 0 hat den Typ `R_MIPS_HI16`. Die zweite für die Adresse 8 hat den Typ `R_MIPS_LO16`.

Das bedeutet, dass die Adresse zu Beginn des `.bss` Segments zu den Befehlen an der Adresse 0 (höherer Teil der Adresse) bzw. 8 (niederer Teil der Adresse) geschrieben wird.

Die Variable `rand_state` befindet sich ganz am Anfang des `.bss` Segments. Wir finden hier Nullen in den Operanden der Befehle `LUI` und `SW`, da sich hier noch nichts befindet—der Compiler weiß noch nicht was hier hingeschrieben werden soll.

Der Linker wird dieses Problem beheben und der höhere Teil der Adresse wird in den Operanden von `LUI` geschrieben und der niedere Teil der Adresse in den Operanden von `SW`.

`SW` addiert den niederen Teil der Adresse und den Inhalt des Registers $V0 (hier befindet sich der höhere Teil).

Das gleiche geschieht mit der Funktion `my_rand()`: Die `R_MIPS_HI16` Relocation teilt dem Linker mit, dass der höhere Teil der Adresse des `.bss` Segments in den Befehl `LUI` geschrieben wird.

Der höhere Teil der Adresse der Variablen `rand_state` befindet sich im Register $V1.

Der Befehl `LW` an der Adresse 0x10 addiert den höheren und niederen Teil und lädt den Wert der Variablen `rand_state` nach $V0.

Der Befehl `SW` an der Adresse 0x54 summiert erneut auf und speichert dann den neuen Wert in der globalen Variable `rand_state`.

IDA arbeitet die Relocations beim Laden ab, sodass diese Details verborgen bleiben, aber wir sollten wissen, dass es sie gibt.

### 1.20.5 Threadsichere Version des Beispiels

Eine threadsichere Version des Beispiels wird später hier gezeigt: 6.2.1 on page 457.
1.21 German text placeholder

Ein C/C++ struct ist lediglich eine Menge von Variablen (nicht unbedingt gleichen Typs), die zusammen gespeichert werden.\(^{137}\)

1.21.1 MSVC: SYSTEMTIME Beispiel

Betrachten wir das SYSTEMTIME\(^{138}\) struct in win32, das die Systemzeit beschreibt.

Das struct ist folgendermaßen definiert:

```c
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Schreiben wir eine C-Funktion, um die aktuelle Zeit auszugeben:

```c
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);
    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);
    return;
}
```

Wir erhalten das Folgende (MSVC 2010):

```c
_t$ = -16 : size = 16
__main PROC
    push ebp
    mov ebp, esp
    sub esp, 16
    lea eax, DWORD PTR _t$[ebp]
    push eax
    call DWORD PTR __imp__GetSystemTime@4
    movzx ecx, WORD PTR _t$[ebp+12] ; wSecond
    push ecx
    movzx edx, WORD PTR _t$[ebp+10] ; wMinute
    push edx
    movzx eax, WORD PTR _t$[ebp+8] ; wHour
    push eax
    movzx ecx, WORD PTR _t$[ebp+6] ; wDay
    push ecx
    movzx edx, WORD PTR _t$[ebp+2] ; wMonth
    push edx
    movzx eax, WORD PTR _t$[ebp] ; wYear
    push eax
    push OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
```

\(^{137}\)AKA „heterogener Container”

\(^{138}\)MSDN: SYSTEMTIME structure
Für dieses struct werden 16 Byte im lokalen Stack reserviert — das entspricht genau sizeof(WORD) * 8 (es gibt 8 WORD Variablen in diesem struct).

Man beachte, dass dieses struct mit dem wYear Feld beginnt. Man kann sagen, dass ein Pointer auf das SYSTEMTIME struct an die Funktion GetSystemTime() übergeben wird, aber man könnte auch sagen, dass ein Pointer auf das Feld wYear übergeben wird, denn dabei handelt es sich um dasselbe! GetSystemTime() schreibt das aktuelle Jahr in den WORD Pointer, verschiebt um 2 Byte, schreibt den aktuellen Monat, usw. usf.

139 MSDN: SYSTEMTIME structure
Kompilieren wir dieses Beispiel in MSVC 2010 mit /GS- /MD und laden es in OllyDbg.

Öffnen wir die Fenster für Daten und Stack an der Adresse, die als erstes Argument der Funktion `GetSystemTime()` übergeben wird und warten, bis das Programm an dieser Stelle ist. Wir sehen das folgende:

![OllyDbg Ausgabe](image)

Abbildung 1.105: OllyDbg: `GetSystemTime()` wurde gerade ausgeführt


Listing 1.299: `printf()` output

```
2014-12-09 22:29:52
```

Wir sehen also diese 16 Byte im Datenfenster:

```
DE 07 0C 00 02 00 09 00 16 00 10 00 34 00 04 03
```

Je zwei Byte repräsentieren ein Feld des structs. Da die **Endianess** hier **little Endian** ist, finden wir das niederwertige Byte zuerst und danach das höherwertige.

Es werden also die folgenden Werte aktuell im Speicher gehalten:

<table>
<thead>
<tr>
<th>Hexadezimalzahl</th>
<th>Dezimalzahl</th>
<th>Feldname</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07DE</td>
<td>2014</td>
<td>wYear</td>
</tr>
<tr>
<td>0x000C</td>
<td>12</td>
<td>wMonth</td>
</tr>
<tr>
<td>0x0002</td>
<td>2</td>
<td>wDayOfWeek</td>
</tr>
<tr>
<td>0x0009</td>
<td>9</td>
<td>wDay</td>
</tr>
<tr>
<td>0x0016</td>
<td>22</td>
<td>wHour</td>
</tr>
<tr>
<td>0x001D</td>
<td>29</td>
<td>wMinute</td>
</tr>
<tr>
<td>0x0034</td>
<td>52</td>
<td>wSecond</td>
</tr>
<tr>
<td>0x03D4</td>
<td>980</td>
<td>wMilliseconds</td>
</tr>
</tbody>
</table>

Wir finden die gleichen Werte im Stackfenster, aber sie werden als 32-Bit-Werte gruppiert.

Die Funktion `printf()` nimmt sich die Werte, die sie braucht und schreibt sie in die Konsole.

Manche Werte werden von `printf()` nicht ausgegeben (wDayOfWeek und wMilliseconds), aber sie sind im Speicher jederzeit für uns verfügbar.
Ein struct durch ein Array ersetzen

Die Tatsache, dass die Felder eines structs Variablen sind, die nebeneinander angeordnet sind, kann leicht durch folgendes Beispiel belegt werden. Wir erinnern uns an die Beschreibung des `SYSTEMTIME` structs und schreiben unser Beispiel wie folgt um:

```c
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);
    printf ("%04d-%02d-%02d %02d:%02d:%02d", 
            array[0] /* wYear */ , array[1] /* wMonth */ , array[3] /* wDay */ ,
    return;
}
```

Der Compiler meckert ein wenig:

```c
systemtime2.c(7) : warning C4133: 'function' : incompatible types - from 'WORD [8]' to 'LPSYSTEMTIME'
```

Trotzdem erzeugt er den folgenden Code:

Listing 1.300: German text placeholder MSVC 2010

```assembly
$SG78573 DB '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
_array$ = -16 ; size = 16
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 16
    lea eax, DWORD PTR _array$[ebp]
    push eax
    call DWORD PTR __imp__GetSystemTime@4
    movzx ecx, WORD PTR _array$[ebp+12] ; wSecond
    push ecx
    movzx edx, WORD PTR _array$[ebp+10] ; wMinute
    push edx
    movzx eax, WORD PTR _array$[ebp+8] ; wSecond
    push eax
    movzx ecx, WORD PTR _array$[ebp+6] ; wDay
    push ecx
    movzx edx, WORD PTR _array$[ebp+2] ; wMonth
    push edx
    movzx eax, WORD PTR _array$[ebp] ; wYear
    push eax
    push OFFSET $SG78573 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
    call _printf
    add esp, 28
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
```

Dieses Programm funktioniert genau wie das erste!

Sehr interessant ist, dass dieser Assemblercode nicht vom entsprechenden Code des ersten Beispiels unterschieden werden kann.

Beim bloßen Ansehen des Codes kann man also nicht feststellen, ob ein struct oder ein Array deklariert wurde.

Trotzdem würde man letzteres nicht annehmen, da es sehr ungebräuchlich ist.
Die Felder des structs können durch den Entwickler ausgetauscht oder verändert werden, etc.
Wir untersuchen dieses Beispiel nicht in OllyDbg, da es mit dem Beispiel mit dem struct identisch ist.

### 1.21.2 Reservieren von Platz für ein struct mit malloc()

Manchmal ist es einfacher structs nicht im lokalen Stack, sondern im Heap abzulegen:

```c
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;
    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));
    GetSystemTime (t);
    printf ("%04d-%02d-%02d %02d:%02d:%02d\n", 
               t->wYear, t->wMonth, t->wDay, 
               t->wHour, t->wMinute, t->wSecond);
    free (t);
    return;
}
```

Kompilieren wir das Beispiel nun mit Optimierung (/Ox), sodass leicht zu erkennen ist, was wir brauchen.

Listing 1.301: German text placeholder MSVC

```assembly
_main PROC
    push esi
    push 16
    call _malloc
    add esp, 4
    mov esi, eax
    push esi
    call DWORD PTR __imp__GetSystemTime@4
    movzx eax, WORD PTR [esi+12]; wSecond
    movzx ecx, WORD PTR [esi+10]; wMinute
    movzx edx, WORD PTR [esi+8]; wHour
    push eax
    movzx eax, WORD PTR [esi+6]; wDay
    push ecx
    movzx ecx, WORD PTR [esi+2]; wMonth
    push edx
    movzx edx, WORD PTR [esi]; wYear
    push eax
    push ecx
    push edx
    push OFFSET $SG78833
    call _printf
    push esi
    call _free
    add esp, 32
    xor eax, eax
    pop esi
    ret 0
_main ENDP
```


Wir finden einen neuen Befehl — MOVZX (Move with Zero eXtend). Er wird in den meisten Fällen als MOVSX verwendet, setzt aber die übrigen Bits auf 0. Das wird benötigt, da printf() einen 32-Bit int erwartet, wir
aber ein WORD im struct haben —also einen 16-Bit Typ ohne Vorzeichen. Das ist der Grund dafür, dass beim Kopieren eines Wertes aus einem WORD in einen int die Bits 16 bis 31 gelöscht werden müsse: hier könnten sich Zufallswerte vom Ergebnis einer vorherigen Operation auf diesem Register befinden.

In diesem Beispiel ist es möglich das struct als Array von 8 WORDs darzustellen:

```c
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD *t;
    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,

    free (t);
    return;
}
```

Wir erhalten:

Listing 1.302: German text placeholder MSVC

```assembly
$SG78594 DB '%%04d-%02d-%02d %02d:%02d:%02d', 0AH, 00H
_main    PROC
    push    esi
    push    16
    call    _malloc
    add     esp, 4
    mov     esi, eax
    push    esi
    call    DWORD PTR __imp__GetSystemTime@4
    movzx   eax, WORD PTR [esi+12]
    movzx   ecx, WORD PTR [esi+10]
    movzx   edx, WORD PTR [esi+8]
    push    eax
    movzx   eax, WORD PTR [esi+6]
    push    ecx
    movzx   ecx, WORD PTR [esi+2]
    push    edx
    movzx   edx, WORD PTR [esi]
    push    eax
    push    ecx
    push    edx
    push    OFFSET $SG78594
    call    _printf
    push    esi
    call    _free
    add     esp, 32
    xor     eax, eax
    pop     esi
    ret     0
_main    ENDP
```

Wieder erhalten wir Code, der nicht vom vorhergehenden zu unterscheiden ist.

Ebenfalls halten wir fest, dass man dies in der Praxis nicht macht, außer man weiß genau, was man tut.

### 1.21.3 UNIX: struct tm

**Linux**

Nehmen wir das struct tm aus time.h in Linux als Beispiel:
```c
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);
    localtime_r(&unix_time, &t);

    printf("Year: %d\n", t.tm_year+1900);
    printf("Month: %d\n", t.tm_mon);
    printf("Day: %d\n", t.tm_mday);
    printf("Hour: %d\n", t.tm_hour);
    printf("Minutes: %d\n", t.tm_min);
    printf("Seconds: %d\n", t.tm_sec);
}
```

Kompilieren wir das Beispiel mit GCC 4.4.1:

**Listing 1.303: GCC 4.4.1**

```asm
main proc near
    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
    sub esp, 40h
    mov dword ptr [esp], 0 ; erstes Argument für time()
    call time
    mov [esp+3Ch], eax
    lea eax, [esp+3Ch] ; nimm Pointer auf Rückgabewert von time()
    lea edx, [esp+10h] ; bei ESP+10h beginnt das struct tm
    mov [esp+4], edx ; übergib pointer auf den Beginn des structs
    mov [esp], eax ; übergib pointer auf Ergebnis von time()
    call localtime_r
    mov eax, [esp+24h] ; tm_year
    lea edx, [eax+76Ch] ; edx=eax+1900
    mov eax, offset aMonthD ; "Month: %d"
    mov [esp+4], edx
    mov [esp], eax
    call printf
    mov edx, [esp+20h] ; tm_mon
    mov eax, offset aDayD ; "Day: %d"
    mov [esp+4], edx
    mov [esp], eax
    call printf
    mov edx, [esp+1Ch] ; tm_mday
    mov eax, offset aHourD ; "Hour: %d"
    mov [esp+4], edx
    mov [esp], eax
    call printf
    mov edx, [esp+18h] ; tm_hour
    mov eax, offset aMinuteD ; "Minutes: %d"
    mov [esp+4], edx
    mov [esp], eax
    call printf
    mov edx, [esp+14h] ; tm_min
    mov eax, offset aSecondsD ; "Seconds: %d"
    mov [esp+4], edx
    mov [esp], eax
    call printf
    leave
    retn
```
Aus irgendwelchen Gründen hat IDA hier nicht die Namen der lokalen Variablen im lokalen Stack hinzugeschrieben. Da wir aber bereits erfahrene Reverse Engineers sind, können wir in diesem einfachen Beispiel auch ohne diese Information auskommen.

Man beachte besonders den Befehl \texttt{lea edx, [eax+76Ch]} — disser Befehl addiert lediglich 0x76C (1900) zum Wert in EAX, ohne jedoch die Flags zu verändern. Siehe hierzu auch den entsprechenden Abschnitt über LEA (\textit{p.1.6 on page 534}).

\textbf{GDB}

Versuchen wir dieses Beispiel in GDB zu laden\textsuperscript{140}:

\begin{verbatim}
Listing 1.304: GDB
dennis@ubuntuvm:~/polygon$ date
Mon Jun 2 18:10:37 EEST 2014
dennis@ubuntuvm:~/polygon$ gcc GCC_tm.c -o GCC_tm
GNU gdb (GDB) 7.6.1-ubuntu ...
Reading symbols from /home/dennis/polygon/GCC_tm...(no debugging symbols found)...done.
(gdb) b printf
Breakpoint 1 at 0x8048330
(gdb) run
Starting program: /home/dennis/polygon/GCC_tm
Breakpoint 1, __printf (format=0x80485c0 "Year: %d\n") at printf.c:29
29 printf.c: No such file or directory.
(gdb) x/20x $esp
0xbffff0dc: 0x080484c3 0x080485c0 0x000007de 0x00000000
0xbffff0ec: 0x08048301 0x538c93ed 0x00000025 0x0000000a
0xbffff0fc: 0x00000012 0x00000002 0x00000005 0x00000072
0xbffff10c: 0x00000001 0x00000098 0x00000001 0x00002a30
0xbffff11c: 0x0804b090 0x08048530 0x00000000 0x00000000
(gdb)
\end{verbatim}

Wir finden unser struct leicht im Stack. Zuerst schauen wir uns an, wie es in \textit{time.h} definiert ist:

\begin{verbatim}
Listing 1.305: time.h
struct tm
{
  int  tm_sec;
  int  tm_min;
  int  tm_hour;
  int  tm_mday;
  int  tm_mon;
  int  tm_year;
  int  tm_wday;
  int  tm_yday;
  int  tm_isdst;
};
\end{verbatim}

Man beachte, dass hier 32-Bit \texttt{int} anstelle von \texttt{WORD} in \texttt{SYSTEMTIME} verwendet werden, sodass jedes Feld 32 Bit belegt.

Hier sind die Felder unseres structs im Stack:

\begin{verbatim}
0xbffff0dc: 0x080484c3 0x080485c0 0x0000007de 0x00000000
0xbffff0ec: 0x08048301 0x538c93ed 0x00000025 0x0000000a
0xbffff0fc: 0x00000012 0x00000002 0x00000005 0x00000072
0xbffff10c: 0x00000001 0x00000098 0x00000001 0x00002a30
0xbffff11c: 0x0804b090 0x08048530 0x00000000 0x00000000
\end{verbatim}

\textsuperscript{140}Das Ergebnis von \texttt{date} ist zu Demonstrationszwecken leicht korrigiert worden. Natürlich ist es nicht möglich, GDB noch in der gleichen Sekunde auszuführen.
Oder als Tabelle:

<table>
<thead>
<tr>
<th>Hexadezimalzahl</th>
<th>Dezimalzahl</th>
<th>Feldname</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000025</td>
<td>37</td>
<td>tm_sec</td>
</tr>
<tr>
<td>0x0000000a</td>
<td>10</td>
<td>tm_min</td>
</tr>
<tr>
<td>0x00000012</td>
<td>18</td>
<td>tm_hour</td>
</tr>
<tr>
<td>0x00000002</td>
<td>2</td>
<td>tm_mday</td>
</tr>
<tr>
<td>0x00000005</td>
<td>5</td>
<td>tm_mon</td>
</tr>
<tr>
<td>0x000000072</td>
<td>114</td>
<td>tm_year</td>
</tr>
<tr>
<td>0x000000001</td>
<td>1</td>
<td>tm_wday</td>
</tr>
<tr>
<td>0x000000098</td>
<td>152</td>
<td>tm_yday</td>
</tr>
<tr>
<td>0x000000001</td>
<td>1</td>
<td>tm_isdst</td>
</tr>
</tbody>
</table>

Genau wie bei SYSTEMTIME( 1.21.1 on page 319) sind auch hier noch weitere Felder verfügbar, die nicht verwendet werden, wie z.B. tm_wday, tm_yday, tm_isdst.

**ARM**

*German text placeholder Keil 6/2013 (German text placeholder)*

Das gleiche Beispiel:

**Listing 1.306: German text placeholder Keil 6/2013 (German text placeholder)**

```assembly
var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
timer = -0xC

PUSH {LR}
MOVS R0, #0 ; timer
SUB SP, SP, #0x34
BL time
STR R0, [SP,#0x38+timer]
MOV R1, SP ; tp
ADD R0, SP, #0x38+timer ; timer
BL localtime_r
LDR R1, =0x76C
LDR R0, [SP,#0x38+var_24]
ADDS R1, R0, R1
ADR R0, aYearD ; "Year: %d"
BL __2printf
LDR R1, [SP,#0x38+var_28]
ADR R0, aMonthD ; "Month: %d"
BL __2printf
LDR R1, [SP,#0x38+var_2C]
ADR R0, aDayD ; "Day: %d"
BL __2printf
LDR R1, [SP,#0x38+var_30]
ADR R0, aHourD ; "Hour: %d"
BL __2printf
LDR R1, [SP,#0x38+var_34]
ADR R0, aMinutesD ; "Minutes: %d"
BL __2printf
LDR R1, [SP,#0x38+var_38]
ADR R0, aSecondsD ; "Seconds: %d"
BL __2printf
ADD SP, SP, #0x34
POP {PC}
```

**German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**

IDA erkennt das tm struct, da IDA die Typen der Argumente von Funktionen aus Bibliotheken wie localtime_r() kennt, sodass IDA hier die Zugriffe auf die Elemente des structs und deren Namen anzeigt.
var_38 = -0x38
var_34 = -0x34

PUSH {R7,LR}
MOV R7, SP
SUB SP, SP, #0x30
MOVS R0, #0 ; time_t *
BLX _time
ADD R1, SP, #0x38+var_34 ; struct tm *
STR R0, [SP,#0x38+var_38]
MOV R0, SP ; time_t *
BLX _ localtime_r
LDR R1, [SP,#0x38+var_34.tm_year]
MOV R0, 0xF44 ; "Year: %d\n"
ADD R0, PC ; char *
ADDW R1, R1, #0
BLX _ printf
LDR R1, [SP,#0x38+var_34.tm_mon]
MOV R0, 0xF3A ; "Month: %d\n"
ADD R0, PC ; char *
ADDW R1, R1, #0
BLX _ printf
LDR R1, [SP,#0x38+var_34.tm_mday]
MOV R0, 0xF35 ; "Day: %d\n"
ADD R0, PC ; char *
ADDW R1, R1, #0
BLX _ printf
LDR R1, [SP,#0x38+var_34.tm_hour]
MOV R0, 0xF2E ; "Hour: %d\n"
ADD R0, PC ; char *
ADDW R1, R1, #0
BLX _ printf
LDR R1, [SP,#0x38+var_34.tm_min]
MOV R0, 0xF28 ; "Minutes: %d\n"
ADD R0, PC ; char *
ADDW R1, R1, #0
BLX _ printf
ADD SP, SP, #0x30
POP {R7,PC}

...
Dieses hier ist ein Beispiel, in dem die Branch Delay Slots uns verwirren können.
Es gibt zum Beispiel den Befehl `addiu $a1, 1900` in Zeile 35, der 1900 zur Jahreszahl hinzuaddiert. Wir dürfen nicht vergessen, dass er vor dem zugehörigen JALR in Zeile 34 ausgeführt wird.

**Struct als Menge von Werten**

Um zu veranschaulichen, dass ein struct nur eine Menge von nebeneinanderliegenden Variablen ist, überarbeiten wir unser Beispiel, indem wir auf die Definition des `tm` structs schauen:Listing 1.305.

```
#include <stdio.h>
#include <time.h>

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday, tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);
    localtime_r (&unix_time, &tm_sec);

    printf("Year: %d\n", tm_year+1900);
    printf("Month: %d\n", tm_mon);
    printf("Day: %d\n", tm_mday);
    printf("Hour: %d\n", tm_hour);
    printf("Minutes: %d\n", tm_min);
    printf("Seconds: %d\n", tm_sec);
}
```

Der Pointer auf das Feld `tm_sec` wird nach `localtime_r` übergeben, d.h. an das erste Element des structs. Der Compiler warnt uns:

```
Listing 1.309: GCC 4.7.3

GCC_tm2.c: In function 'main':
GC2_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from incompatible pointer type [\~\ enabled by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but argument is of type 'int *'
```

Trotzdem erzeugt er folgenden Code:

```
Listing 1.310: GCC 4.7.3

main proc near
var_30 = dword ptr -30h
var_2C = dword ptr -2Ch
unix_time = dword ptr -1Ch
tm_sec = dword ptr -18h
tm_min = dword ptr -14h
tm_hour = dword ptr -10h
tm_mday = dword ptr -0Ch
tm_mon = dword ptr -8
tm_year = dword ptr -4

push   ebp
mov    ebp, esp
and    esp, 0FFFFFFF0h
sub    esp, 30h
call   main
mov    [esp+30h+var_30], 0 ; arg 0
call   time
mov    [esp+30h+unix_time], eax
lea    eax, [esp+30h+tm_sec]
mov    [esp+30h+var_2C], eax
lea    eax, [esp+30h+unix_time]
mov    [esp+30h+var_30], eax
```
Dieser Code ist zum vorherigen identisch und es ist unmöglich zu sagen, ob es sich im originalen Quellcode um ein struct oder nur um eine Menge von Variablen handelt.

Es funktioniert also, ist aber in der Praxis nicht empfehlenswert.

Nicht optimierende Compiler legen normalerweise Variablen auf dem lokalen Stack in der Reihenfolge an, in der sie in der Funktion deklariert wurden.

Ein Garantie dafür gibt es freilich nicht.

Andere Compiler könnten an dieser Stelle übrigens davor warnen, dass die Variablen tm_year, tm_mon, tm_mday, tm_hour, tm_min - nicht aber tm_sec - ohne Initialisierung verwendet werden.

Der Compiler weiß nicht, dass diese durch die Funktion localtime_r() befüllt werden.

Wir haben dieses Beispiel ausgewählt, da alle Felder im struct vom Typ int sind.

Es würde nicht funktionieren, wenn die Felder 16 Bit (WORD) groß wären, wie im Beispiel des SYSTEMTIME structs—GetSystemTime() würde sie falsch befüllen (da die lokalen Variablen auf 32-Bit-Grenzen angeordnet sind). Mehr dazu im folgenden Abschnitt: „German text placeholder” (1.21.4 on page 334).


Dieses Beispiel wird nicht im Debugger gezeigt, da es dem gerade gezeigten entspricht.

**Struct als Array aus 32-Bit-Worten**

```c
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i;

    unix_time=time(NULL);
}
```

\(^{141}\)German text placeholder http://go.yurichev.com/17264
localtime_r (&unix_time, &t);

for (i=0; i<9; i++)
{
    int tmp=((int*)t)[i];
    printf ("0x%08X (%d)\n", tmp, tmp);
}


0x0000002D (45)
0x00000033 (51)
0x00000017 (23)
0x0000001A (26)
0x00000006 (6)
0x00000072 (114)
0x00000006 (6)
0x000000CE (206)
0x00000001 (1)

Die Variablen sind hier in der gleichen Reihenfolge, in der die in der Definition des structs aufgezählt werden: 1.305 on page 326.

Hier ist der erzeugte Code:

Listing 1.311: German text placeholder GCC 4.8.1

main  proc near
push ebp
mov ebp, esp
push esi
push ebx
and esp, 0FFFFFFF0h
sub esp, 40h
mov dword ptr [esp], 0 ; timer
lea ebx, [esp+14h]
call _time
lea esi, [esp+38h]
mov [esp+4], ebx ; tp
mov [esp+10h], eax
lea eax, [esp+10h]
mov [esp], eax ; timer
call _localtime_r
nop
lea esi, [esi+0] ; NOP
loc_80483D8:
; EBX ist hier ein Pointer auf das struct, ESI ist der Pointer auf dessen Ende.
mov eax, [ebx] ; hole 32-Bit-Wort aus dem Array
add ebx, 4 ; nächstes Feld im struct
mov dword ptr [esp+4], offset a0x0808xO ; "0x%08X (%d)\n"
mov dword ptr [esp], 1
mov [esp+0Ch], eax ; übergib Wert an printf()
mov [esp+8], eax ; übergib Wert an printf()
call __printf_chk
cmp ebx, esi ; Ende des structs?
 jnz short loc_80483D8 ; nein - dann lade nächsten Wert
lea esp, [ebp-8]
pop ebx
pop esi
pop ebp
retn
main  endp

Tatsächlich: der Platz auf dem lokalen Stack wird zuerst wie in struct und dann wie ein Array behandelt. Es ist sogar möglich, die Felder des structs über diesen Pointer zu verändern.
Und wiederum ist es zweifellos ein seltsamer Weg die Dinge umzusetzen; er ist für produktiven Code definitiv nicht empfehlenswert.

**German text placeholder**

Versuchen Sie als Übung die Monatsnummer zu verändern (um 1 zu erhöhen), indem Sie das struct wie ein Array behandeln.

**Struct als Bytearray**

Wir können sogar noch weiter gehen. Casten wir den Pointer zu einem Bytearray und ziehen einen Dump:

```c
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i, j;

    unix_time=time(NULL);
    localtime_r(&unix_time, &t);

    for (i=0; i<9; i++)
    {
        for (j=0; j<4; j++)
            printf("0x%02X ", ((unsigned char*)&t)[i*4+j]);
        printf("\n");
    }
}
```

Wir haben dieses Beispiel auch zur Systemzeit 23:51:45 26-July-2014 ausgeführt \(^{142}\). Die Werte sind genau dieselben wie im vorherigen Dump (1.21.3 on the preceding page) und natürlich steht das LSB vorne, da es sich um eine Little-Endian-Architektur handelt (?? on page ??).

### Listing 1.312: German text placeholder GCC 4.8.1

```
main proc near
    push ebp
    mov ebp, esp
    push edi
    push esi
    push ebx
    and esp, 0FFFFFFF0h
    sub esp, 40h
    mov dword ptr [esp], 0 ; timer
    lea esi, [esp+14h]
    call _time
    lea edi, [esp+38h] ; Ende des structs
    mov [esp+4], esi ; tp
    mov [esp+10h], eax
    lea eax, [esp+10h]
    mov [esp], eax ; timer
    call _localtime_r
```

\(^{142}\) Datum und Uhrzeit sind zu Demonstrationszwecken identisch. Die Bytewerte sind modifiziert.
Ein wichtiges Thema ist das Packen von Feldern in structs.

Betrachten wir ein einfaches Beispiel:

```c
#include <stdio.h>

struct s {
    char a;
    int b;
    char c;
    int d;
};

void f(struct s s) {
    printf("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
}

int main() {
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
    tmp.c=3;
    tmp.d=4;
    f(tmp);
}
```

Wie wir sehen haben wir zwei char Felder (jedes ist exakt ein Byte groß) und zwei weitere vom Typ int (zu je 4 Byte).

**x86**

Das Beispiel kompiliert zu folgendem Code:

Listing 1.313: MSVC 2012 /GS- /Ob0
Wir übergeben das struct als Ganzes, aber im Code können wir sehen, dass das struct in ein temporäres struct kopiert wird (ein Platz hierfür wird in Zeile 10 auf dem Stack reserviert), und dass dann alle 4 Felder einzeln (in den Zeilen 12...19) kopiert werden und anschließend ihr Pointer (Adresse) übergeben wird.

Das struct wird kopiert, da nicht bekannt ist, ob die Funktion f() das struct verändern wird oder nicht. Wenn es verändert wird, muss das struct in main() auf dem vorherigen Stand bleiben.

Wir könnten C/C++ Pointer verwenden und der erzeugte Code wäre fast der gleiche nur ohne das Kopieren.

Wie wir sehen können, wird die Adresse von jedem Feld auf einer 4 Byte Grenze angeordnet. Das ist der Grund dafür, dass jeder char hier 4 Byte belegt (wie ein int). Es ist für die CPU einfacher auf Speicher an entsprechend angeordneten Adressen zuzugreifen und Daten von dort im Cache zwischenspeichern.

Trotzdem ist dieses Vorgehen nicht besonders ökonomisch.

Komplieren wir mit der Option (/Zp1)/Zp[n] pack structures on n-byte boundary).

Das struct wird auch in main() kopiert. Nicht Feld für Feld, sondern alle 10 Byte direkt durch Verwendung von drei Paaren von MOV Befehlen. Warum aber nicht 4 Paare?

Der Compiler hat entschieden, dass es besser ist, die 10 Byte mit 3 MOV Befehlspaaren zu kopieren als zwei 32-Bit-Worte und dann zwei Byte mit insgesamt 4 MOV Paaren.

Solch eine Implementierung, die zum Kopieren MOV anstelle eines Aufrufs von memcpy() verwendet, ist sehr gebräuchlich, da es schneller ist als ein Funktionsaufruf von memcpy() — zumindest für kurze Blöcke.

Man kann sich leicht überlegen, dass, wenn ein struct in vielen Quellcode und Objektdateien verwendet wird, alle diese mit der gleichen Konvention bezüglich des Packens in structs kompiliert werden müssen.

Neben der Option Zp in MSVC, die die Anordnung der Felder von structs festlegt, gibt es auch die Compileroption #pragma pack, die direkt im Quellcode definiert werden kann. Sie ist sowohl in MSVC als auch in GCC verfügbar.

Gehen wir zurück zum SYSTEMTIME struct, das aus 16-Bit-Feldern besteht. Wie kann unser Compiler wissen, dass diese in 1-Byte-Anordnung gepackt werden müssen?

Die Datei WinNT.h enthält dies:

Listing 1.315: WinNT.h

#include "pshpack1.h"

143 MSDN: Working with Packing Structures
144 Structure-Packing Pragmas
Und dies:

Listing 1.316: WinNT.h
#include "pshpack4.h" // 4 byte packing is the default

Die Datei PshPack1.h sieht wie folgt aus:

Listing 1.317: PshPack1.h
#if !(defined(lint) || defined(RC_INVOKED))
#if (_MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif
/* ! (defined(lint) || defined(RC_INVOKED)) */

Dies sagt dem Compiler wie die structs, die hinter #pragma pack definiert werden, gepackt werden müssen.
OllyDbg + standardmäßig gepackte Felder

Betrachten wir unser Beispiel (in dem die Felder standardmäßig auf 4 Byte angeordnet werden) in OllyDbg:

![CPU - main thread, module packing](image)

Abbildung 1.106: OllyDbg: vor der Ausführung von printf()

Wir sehen unsere 4 Felder im Datenfenster.

Wir fragen uns aber, woher die Zufallsbytes (0x30, 0x37, 0x01) stammen, die neben dem ersten (a) und dritten (c) Feld liegen.

Betrachten wir unser Listing 1.313 on page 334, erkennen wir, dass das erste und dritte Feld vom Typ **char** ist, und daher nur ein Byte geschrieben wird, nämlich 1 bzw. 3 (Zeilen 6 und 8).

Die übrigen 3 Byte des 32-Bit-Wortes werden im Speicher nicht verändert! Deshalb befinden sich hier zufällige Reste.

Diese Reste beeinflussen den Output von printf() in keinerlei Weise, da die Werte für die Funktion mithilfe von MOVsx vorbereitet werden, der Bytes und nicht Worte als Argumente hat: Listing 1.313 (Zeilen 34 und 38). Der vorzeichenerweiternde Befehl MOVsx wird hier übrigens verwendet, da **char** standardmäßig in MSVC und GCC vorzeichenbehaftet ist. Würde hier der Datentyp unsigned char oder uint8_t verwendet, würde der Befehl MOVZX stattdessen verwendet.

Wir sehen unsere 4 Felder im Datenfenster.
**OllyDbg + Felder auf 1 Byte Grenzen angeordnet**

Hier sind die Dinge viel klarer ersichtlich: 4 Felder benötigen 16 Byte und die Werte werden nebeneinander gespeichert.

Abbildung 1.107: OllyDbg: Vor der Ausführung von `printf()`

**ARM**

_**German text placeholder Keil 6/2013**_ (German text placeholder)

Listing 1.318: _German text placeholder Keil 6/2013_ (German text placeholder)

Wir sehen, dass hier ein struct anstelle eines Pointers auf ein struct übergeben wird und da die ersten vier Funktionsargumente in ARM über Register übergeben werden, werden die Felder des structs mittels R0-R3 übergeben.

LDRB lädt ein Byte aus dem Speicher und erweitert es unter Berücksichtigung des Vorzeichens auf 32 Bit. Dies entspricht MOVSX in x86. Hier wird der Befehl verwendet, um die Felder a und c des structs zu laden.
Eine weitere Sache, die ins Auge sticht, ist, dass anstelle eines Funktionsepilogs ein Sprung zum Epilog einer anderen Funktion vorliegt. Bei der anderen handelt es sich um eine komplett unterschiedliche Funktion, die in keiner Weise mit unserer zusammenhängt, aber denselben Epilog hat (möglicherweise, da diese ebenfalls 5 lokale Variablen enthält (\(5 \times 4 = 0\times14\)).

Außerdem befindet sie sich im Code in der Nähe (man vergleiche die Adressen).

Tatsächlich spielt es auch keine Rolle, welcher Epilog ausgeführt wird, solange dieser wie gewünscht funktioniert.

Offensichtlich hat Keil aus ökonomischen Gründen entschieden, einen Teil der anderen Funktion wiederzuverwenden. Der Epilog benötigt 4 Bytes—während der Sprung nur 2 verbraucht.

**ARM + German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)**

Listing 1.319: German text placeholder Xcode 4.6.3 (LLVM) (German text placeholder)

```
var_C = -0xC
PUSH {R7, LR}
MOV R7, SP
SUB SP, SP, #4
MOV R9, R1 ; b
MOV R1, R0 ; a
MOVM R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
SXTB R1, R1 ; prepare a
MOVT.W R0, #0
STR R3, [SP, 0x0C+var_C] ; place d to stack for printf()
ADD R0, PC ; format-string
SXTB R3, R2 ; prepare c
MOV R2, R9 ; b
BLX _printf
ADD SP, SP, #4
POP {R7, PC}
```

SXTB (Signed Extend Byte) ist analog zu MOVsx in x86. Der ganze Rest ist identisch.

**MIPS**

Listing 1.320: German text placeholder GCC 4.4.5 (IDA)

```
f:
var_18 = -0x18
var_10 = -0x10
var_4 = -4
arg_0 = 0
arg_4 = 4
arg_8 = 8
arg_C = 0xC

; $a0=s.a
; $a1=s.b
; $a2=s.c
; $a3=s.d

lui $gp, (_gnu_local_gp >> 16)
addiu $sp, $sp, -0x28
la $gp, (_gnu_local_gp & 0xFFFF)
sw $ra, 0x28+var_4($sp)
sw $gp, 0x28+var_10($sp)
; prepare a byte from 32-bit big-endian integer:
sra $t0, $a0, 24
move $v1, $a1
; prepare a byte from 32-bit big-endian integer:
sra $v0, $a2, 24
lw $t9, (printf & 0xFFFF)($gp)
sw $a0, 0x28+arg_0($sp)
lui $a0, ($LC0 >> 16)  # "a=%d; b=%d; c=%d; d=%d\n"
sw $a3, 0x28+var_18($sp)
```
Die Felder des structs landen in den Registern $A0..A3$ und werden dann für printf() nach $A1..A3$ verschoben, während das vierte Feld (aus $A3$) mit SW über den lokalen Stack übergeben wird.

Wir fragen uns aber, warum es hier zwei SRA („Shift Word Right Arithmetic“) Befehle für die beiden char Felder gibt.

MIPS ist eine Big-Endian-Architektur ?? on page ?? und das Debian Linux, mit dem wir arbeiten, ist ebenfalls Big Endian.

Wenn nun Byte Variablen in 32-Bit-Feldern im struct gespeichert werden, besetzen sie die hohen Bits 24..31.

Wenn ein Variable vom Typ char zu einem 32-Bit-Wert erweitert werden muss, muss sie um 24 Bits nach rechts verschoben werden.

char ist ein vorzeichenbehafteter Typ, sodass hier eine arithmetische Verschiebung anstelle einer logischen erfolgen muss.

Eine Sache noch

Ein struct als Funktionsargument zu übergeben (anstelle eines Pointers auf ein struct) ist das gleiche wie alle Felder des structs einzeln zu übergeben.

Wenn die Felder im struct standardmäßig gepackt werden, kann die Funktion f() wie folgt neu geschrieben werden:

```c
void f(char a, int b, char c, int d)
{
    printf("a=%%d; b=%%d; c=%%d; d=%%d\n", a, b, c, d);
}
```

Das führt schlussendlich zum gleichen Code.

### 1.21.5 Verschachtelte structs

Wir betrachten nun Situationen, in denen ein struct innerhalb eines anderen definiert ist.
void f(struct outer_struct s) {
    printf("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
        s.a, s.b, s.c.a, s.c.b, s.d, s.e);
}

int main() {
    struct outer_struct s;
    s.a=1;
    s.b=2;
    s.c.a=100;
    s.c.b=101;
    s.d=3;
    s.e=4;
    f(s);
}

Beide Felder von inner_struct werden hier zwischen den Feldern a,b und d,e von outer_struct platziert.

Kompilieren wir mit (MSVC 2010):

```
Listing 1.321: German text placeholder MSVC 2010 /Ob0

SG2802 DB 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0AH, 00H

_TEXT SEGMENT
_s$ = 8
_f PROC
    mov eax, DWORD PTR _s$[esp+16]
    movsx ecx, BYTE PTR _s$[esp+12]
    mov edx, DWORD PTR _s$[esp+8]
    push eax
    mov eax, DWORD PTR _s$[esp+8]
    push ecx
    mov ecx, DWORD PTR _s$[esp+8]
    push edx
    movsx edx, BYTE PTR _s$[esp+8]
    push eax
    push ecx
    push edx
    push OFFSET SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d'
    call _printf
    add esp, 28
    ret 0
_f ENDP

_s$ = -24
_main PROC
    sub esp, 24
    push ebx
    push esi
    push edi
    mov ecx, 2
    sub esp, 24
    mov eax, esp
    ; from this moment, EAX is synonymous to ESP:
    mov BYTE PTR _s$[esp+60], 1
    mov ebx, DWORD PTR _s$[esp+60]
    mov DWORD PTR [eax], ebx
    mov DWORD PTR [eax+4], ecx
    lea edx, DWORD PTR [ecx+98]
    lea esi, DWORD PTR [ecx+99]
    lea edi, DWORD PTR [ecx+2]
    mov DWORD PTR [eax+8], edx
    mov BYTE PTR _s$[esp+76], 3
    mov ecx, DWORD PTR _s$[esp+76]
    mov DWORD PTR [eax+12], esi
    mov DWORD PTR [eax+16], ecx
    mov DWORD PTR [eax+20], edi
```
Eine Kuriosität ist hier, dass wir, wenn wir in den Assemblercode schauen, nicht einmal feststellen können, dass hier ein struct innerhalb eines anderen verwendet wurde! Durch diese Beobachtung können wir sagen, dass verschachtelte structs in lineare oder eindimensionale structs ausgefaltet werden.

Wenn wir aber die Deklaration von struct inner_struct c; durch struct inner_struct *c; ersetzen (also einen Pointer verwenden), sieht die Sache ganz anders aus.
OllyDbg

Laden wir dieses Beispiel in OllyDbg und schauen uns `outer_struct` im Speicher an:

Abbildung 1.108: OllyDbg: vor der Ausführung von `printf()`

Die Werte werden wie folgt im Speicher abgelegt:

- `(outer_struct.a)` (byte) 1 + 3 Byte mit zufälligen Werten;
- `(outer_struct.b)` (32-bit word) 2;
- `(inner_struct.a)` (32-bit word) 0x64 (100);
- `(inner_struct.b)` (32-bit word) 0x65 (101);
- `(outer_struct.d)` (byte) 3 + 3 Byte mit zufälligen Werten;
- `(outer_struct.e)` (32-bit word) 4.

1.21.6 Bitfields in einem struct

CPUID Beispiel

Die Sprache C/C++ erlaubt die Definition der exakten Anzahl von Bits für jedes Feld in einem struct. Das ist sehr nützlich, wenn man Speicherplatz sparen muss. Zum Beispiel genügt ein Bit für eine `bool` Variable. Natürlich ist dieses Vorgehen nicht angebracht, wenn die Geschwindigkeit wichtig ist.

Betrachten wir das Beispiel des CPUID145 Befehls. Dieser Befehl liefert Informationen über die aktuelle CPU und ihre Eigenschaften.

Wenn EAX vor der Ausführung des Befehls auf 1 gesetzt ist, liefert CPuid diese Informationen gepackt in das EAX Register zurück:

<table>
<thead>
<tr>
<th>Adresse</th>
<th>Wert</th>
<th>MSVC</th>
<th>CPUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0035:00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:03</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:08</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:09</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:0A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:0B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:0C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:0D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:0E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035:0F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MSVC 2010 verfügt über ein CPUID Makro, aber GCC 4.4.1 nicht. Erstellen wir also für uns eine solche Funktion in GCC, indem wir den built-in Assembler146 verwenden.

145 wikipedia
146 Mehr zum internen GCC Assembler
```c
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
    asm volatile("cpuid:"="a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"(code));
}
#endif

#ifdef __GNUC__
#endif
#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX {
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main() {
    struct CPUID_1_EAX *tmp;
    int b[4];

    #ifdef _MSC_VER
    __cpuid(b,1);
    #endif

    #ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
    #endif

    tmp=(struct CPUID_1_EAX *)&b[0];

    printf ("stepping=%d\n", tmp->stepping);
    printf ("model=%d\n", tmp->model);
    printf ("family_id=%d\n", tmp->family_id);
    printf ("processor_type=%d\n", tmp->processor_type);
    printf ("extended_model_id=%d\n", tmp->extended_model_id);
    printf ("extended_family_id=%d\n", tmp->extended_family_id);

    return 0;
}
```

Nachdem CPUID die Register EAX/EBX/ECX/EDX befüllt hat, werden deren Inhalte in das Array b[ ] geschrieben. Danach haben wir einen Pointer auf das CPUID_1_EAX struct und zeigen auf den Wert in EAX aus dem Array b[ ].

Mit anderen Worten: wir behandeln einen 32-Bit int wie ein struct. Danach lesen wir spezifische Bits aus dem struct.

**MSVC**
Kompilieren wir das Beispiel in MSVC 2008 mit der Option /Ox:
Der Befehl SHR verschiebt den Wert in EAX um die Anzahl der Bits die überprungen werden müssen, d.h. wir ignorieren einige Bits am rechten Rand.

Der Befehl AND löscht die nicht benötigten Bits am linken Rand bzw. belässt nur die Bits in EAX, die wir auch benötigen.
MSVC + OllyDbg

Laden wir unser Beispiel in OllyDbg und schauen welche Werte sich nach der Ausführung von CPUID in den Register EAX/EBX/ECX und EDX befinden:

Abbildung 1.109: OllyDbg: Nach Ausführung von CPUID

EAX enthält 0x0800206A7 (meine CPU ist ein Intel Xeon E3-1220). Dies entspricht 0b00000000000000100000011010100111 in Binärdarstellung.

Die Bits werden wie folgt durch die Felder aufgeteilt:

<table>
<thead>
<tr>
<th>Feld</th>
<th>in binär</th>
<th>in dezimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved2</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>extended_family_id</td>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>extended_model_id</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>reserved1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>processor_id</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>family_id</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>model</td>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>stepping</td>
<td>0111</td>
<td>7</td>
</tr>
</tbody>
</table>

Listing 1.323: Console output

Versuchen wir es mit GCC 4.4.1 mit der Option -O3

Listing 1.324: German text placeholder GCC 4.4.1
Fast das gleiche. Das einzig Bemerkenswerte ist, dass GCC die Berechnung von extended_model_id und extended_family_id in einem Block kombiniert, anstatt sie vor jedem Aufruf von printf() getrennt zu berechnen.

**German text placeholder**

Wie wir bereits im Abschnitt über die FPU ([1.17 on page 195](#1.17)) besprochen haben, bestehen float und double jeweils aus einem Vorzeichen, einem Signifikanden (oder Bruch) und einem Exponenten. Wir wollen am Beispiel des Typs float untersuchen, ob wir direkt mit diesen Feldern arbeiten können.

```c
#include <stdio.h>
```

(S — German text placeholder)
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // Bruch
    unsigned int exponent : 8; // Exponent + 0x3FF
    unsigned int sign : 1;     // Vorzeichenbit
};

float f(float _in)
{
    float f=_in;
    struct float_as_struct t;
    
    assert (sizeof (struct float_as_struct) == sizeof (float));
    memcpy (&t, &f, sizeof (float));

    t.sign=1; // set negative sign
    t.exponent=t.exponent+2; // multipliziere d mit 2^n (n ist hier 2)
    memcpy (&f, &t, sizeof (float));

    return f;
};

int main()
{
    printf ("%f\n", f(1.234));
};

Das struct float_as_struct belegt den gleichen Speicherplatz wie ein float, d.h., 4 Byte oder 32 Bit. Wir setzen das negative Vorzeichen im Eingabewert und multiplizieren die gesamte Zahl mit 2^n, d.h. 4, indem wir zum Exponenten 2 hinzuzufügendieren.

Kompilieren wir das Beispiel mit MSVC 2008 ohne Optimierung:

Listing 1.325: German text placeholder MSVC 2008

<table>
<thead>
<tr>
<th>t$ = -8 ; size = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>f$ = -4 ; size = 4</td>
</tr>
<tr>
<td>_in$ = 8 ; size = 4</td>
</tr>
</tbody>
</table>

?f@YAMM@Z PROC ; f
push ebp
mov ebp, esp
sub esp, 8

fld DWORD PTR _in$[ebp]
fstp DWORD PTR _f$[ebp]
push 4
lea eax, DWORD PTR _f$[ebp]
push eax
lea ecx, DWORD PTR _t$[ebp]
push ecx
call _memcpy
add esp, 12

mov edx, DWORD PTR _t$[ebp]
or edx, -2147483648 ; 80000000H - setze Minuszeichen
mov DWORD PTR _t$[ebp], edx

mov eax, DWORD PTR _t$[ebp]
shr eax, 23 ; 000000017H - entferne Signifikanden
and eax, 255 ; 000000ffH - lasse nur Exponenten hier
add eax, 2
addi eax, 2
and eax, 255 ; 000000ffH
shl eax, 23 ; 000000017H - verschiebe Ergebnis auf Bits 30:23
Ein wenig redundant. Hätten wir mit dem Flag /Ox kompiliert, wäre dort ein Aufruf von TTmemcpy(), sondern die Variable \textit{f} wäre direkt verwendet worden. Einfacher ist es aber auf jeden Fall, sich die nicht optimierte Version anzuschauen.

Was würde GCC 4.4.1 mit -O3 hier tun?

Listing 1.326: \textit{German text placeholder} GCC 4.4.1

```
; \textit{f}(float)
\textit{public}_Z1ff
_Z1ff proc near
var_4 = dword ptr -4
arg_0 = dword ptr 8

push ebp
mov ebp, esp
sub esp, 4
mov eax, [ebp+arg_0]
or eax, 80000000h ; setze Minuszeichen
mov edx, eax
and edx, 807FFFFFFh ; lasse nur Vorzeichen und Signifikanden in EAX
shr edx, 23 ; bereite Exponenten vor
add edx, 2 ; addiere 2
movzx edx, dl ; löse alle Bits außer 7:0 in EDX
shl edx, 23 ; verschiebe berechneten Exponenten
or eax, edx ; kombiniere neuen Exponenten und Originalwert ohne Exponenten
mov [ebp+var_4], eax
fld [ebp+var_4]
leave
retn
_Z1ff endp

\textit{public} main
main proc near
push ebp
mov ebp, esp
and esp, 0FFFFFFh
sub esp, 10h
fld ds:dword_8048614 ; -4.936
fstp qword ptr [esp+8]
mov dword ptr [esp+4], offset asc_8048610 ; "%f\n"
mov dword ptr [esp], 1
call __printf_chk
xor eax, eax
leave
retn
main endp
```
Die Funktion \( f() \) ist fast verständlich. Viel interessanter ist jedoch, dass GCC trotz des Sammelsuriums an Feldern in der Lage war, das Ergebnis von \( f(1.234) \) während des Kompiliierens zu berechnen und der Funktion \( printf() \) während der Compilezeit als vorberechneten Wert bereitzustellen.

### 1.21.7 Übungen
- [http://challenges.re/71](http://challenges.re/71)
- [http://challenges.re/72](http://challenges.re/72)

### 1.22 Unions

Die Cpp \textit{union} wird hauptsächlich verwendet um eine Variable (oder einen Speicherblock) eines Datentyps als Variable eines anderen Datentyps zu interpretieren.

#### 1.22.1 Pseudozufallszahlengenerator Beispiel

Wenn wir Zufallszahlen vom Typ \textit{float} zwischen 0 und 1 brauchen, ist die einfachste Möglichkeit einen \textit{PRNG} wie den Mersenne-Twister zu verwenden. Er produziert vorzeichenlose 32-Bit-Werte (mit anderen Worten: er erzeugt 32 zufällige Bits). Diesen Wert können wir in einen \textit{float} umwandeln und dann durch \textit{RAND\_MAX} (0xFFFFFFFF in unserem Fall) teilen—wir erhalten einen Wert im Intervall 0..1.

Wie wir jedoch wissen, ist die Division langsam. Auch würden wir gerne so wenig FPU Operation wie möglich verwenden. Deshalb fragen wir uns, ob wir die Division loswerden können.

Erinnern wir uns an den Aufbau einer Fließkommazahl: sie besteht aus einem Vorzeichenbit, Bits im Signifikanden und Bits im Exponenten. Wir müssen also lediglich Zufallsbits in allen Bits des Signifikanden speichern um einen zufälligen Fließkommazahl zu erhalten!

Der Exponent kann nicht null sein (in diesem Fall ist die Fließkommazahl denormalisiert); also speichern wir 0b01111111 im Exponenten—das bedeutet, dass der Exponent 1 ist. Dann füllen wir den Signifikanden mit Zufallsbits, setzen das Vorzeichenbit auf 0 (entspricht einer positiven Zahl) und volà. Die erzeugte Zahl liegt zwischen 1 und 2; wir müssen also am Ende noch 1 abziehen.

Ein sehr einfacher linearer Kongruenzgenerator für Zufallszahlen wird in meinem Beispiel\textsuperscript{147} vorgestellt: er erzeugt 32-Bit-Zahlen. Der \textit{PRNG} wird mit der aktuellen Zeit um UNIX-Timestamp-Format initialisiert.

Wir stellen hier den Typ \textit{float} als \textit{union} dar–das ist die C/C++ Konstruktion, die es uns ermöglicht, einen Speicherblock als unterschiedliche Typen aufzufassen. In unserem Fall sind wir in der Lage eine Variable vom Typ \textit{union} zu erzeugen und dann auf diese wie auf einen \textit{float} oder \textit{uint32\_t} zuzugreifen. Man kann sagen, dass es sich dabei um einen Hack, d.h. Trick handelt. Sogar um einen sehr schmutzigen.

Der \textit{PRNG} Code für Integer ist der bereits betrachtete: \texttt{1.20 on page 313}. Deshalb verzichten wir an dieser Stelle auf ein erneutes Listing des kompilierten Codes.

```c
#include <stdio.h>
#include <stdint.h>
#include <time.h>

// integer PRNG definitions, data and routines:

// constants from the Numerical Recipes book
const uint32\_t RNG\_a=1664525;
const uint32\_t RNG\_c=1013904223;
uint32\_t RNG\_state; // global variable

void my\_srand(uint32\_t i)
{
    RNG\_state=i;
};

uint32\_t my\_rand()
{
    RNG\_state=RNG\_state*RNG\_a+RNG\_c;
    return RNG\_state;
}
```

\textsuperscript{147}die Idee stammt von: [http://go.yurichev.com/17308](http://go.yurichev.com/17308)
union uint32_t_float {
    uint32_t i;
    float f;
};

float float_rand() {
    union uint32_t_float tmp;
    tmp.i = my_rand() & 0x007fffffff | 0x3f800000;
    return tmp.f-1;
}

int main() {
    my_srand(time(NULL)); // PRNG initialization

    for (int i=0; i<100; i++)
        printf("%f\n", float_rand());

    return 0;
};

x86

Listing 1.327: German text placeholder MSVC 2010

$SG4238 DB 'f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff000000000000r ; 1
tv130 = -4
tmp$ = -4
?float_rand@YAMXZ PROC
    push ecx
    call ?my_rand@YAIWX
    EAX=Pseudozufallswert
        and eax, 0388607 ; 007ffffffH
        or eax, 1065353216 ; 3f800000H
    EAX=Pseudozufallswert & 0x007fffffff | 0x3f800000
    ; speichere ihn auf lokalem Stack:
        mov DWORD PTR _tmp$[esp+4], eax
    ; lade ihn erneut als float:
        fld DWORD PTR _tmp$[esp+4]
    ; subtrahiere 1.0:
        fsub QWORD PTR __real@3ff0000000000000
    ; speichere erhaltenen Wert auf dem lokalen Stack und lade ihn erneut:
        fstp DWORD PTR tv130[esp+4] ; \ diese Befehle sind redundant
        fld DWORD PTR tv130[esp+4]; /
        pop ecx
        ret 0
?float_rand@YAMXZ ENDP

_main PROC
    push esi
    xor eax, eax
    call _time
    push eax
    call ?my_srand@YAXI@Z
    add esp, 4
    mov esi, 100
$LL3@main:
call ?float_rand@YAMXZ
Die Namen der Funktionen sind hier so seltsam, weil das Beispiel als C++ kompiliert wurde, und hier name mangling in C++ vorliegt. Dies werden wir später besprechen: ?? on page ?? Wenn wir das Beispiel mit MSVC 2012 kompilieren, verwendet es SIMD Befehle für die FPU; mehr dazu unter: 1.26.5 on page 400.

**ARM (German text placeholder)**

Listing 1.328: German text placeholder GCC 4.6.3 (IDA)

```
float_rand
  STMFD SP!, {R3,LR}
  BL my_rand
; R0=Pseudorzufallswert
    FLDS S0, =1.0
; S0=1.0
    BIC R3, R0, #0xFF000000
    BIC R3, R3, #0x800000
    ORR R3, R3, #0x3F800000
; R3=Pseudorzufallswert & 0x007fffff | 0x3f800000
; kopiere aus R3 zur FPU (Register S15).
; entspricht bitweisem Kopieren, Umwandlung findet nicht statt:
    FMSR S15, R3
; subtrahiere 1.0 und lasse Ergebnis in S0:
    FSUBS S0, S15, S0
    LDMFD SP!, {R3,PC}
flt_5C
  DCFS 1.0
main
  STMFD SP!, {R4,LR}
  MOV R0, #0
  BL time
  BL my_srand
  MOV R4, #0x64 ; 'd'
loc_78
  BL float_rand
; S0=Pseudorzufallswert
  LDR R0, =aF ; "%f"
; kovertiere float in double (printf() benötigt dieses Format):
    FCVTDS D7, S0
; bitwises Kopieren von D7 nach R2/R3 (für printf()):
    FMRRD R2, R3, D7
    BL printf
    SUBS R4, R4, #1
    BNE loc_78
    MOV R0, R4
    LDMFD SP!, {R4,PC}
aF
  DCB "%f",0xA,0
```

Wir ziehen auch einen Dump in objdump und shen, dass die FPU Befehle andere Namen als in IDA haben. Offenbar haben die Entwickler von IDA und binutils unterschiedliche Handbücher verwendet. Möglicherweise ist es hilfreich, beide Varianten der Befehlsnamen zu kennen.

Listing 1.329: German text placeholder GCC 4.6.3 (objdump)

```
00000038 <float_rand>:
```
Die Befehle an den Stellen 0x5c in float_rand() und 0x38 in main() sind (Pseudo-)Zufallsrauschen.

1.22.2 Berechnung der Maschinengenauigkeit


Interessant ist, wie einfach die Maschinengenauigkeit berechnet werden kann:

```c
#include <stdio.h>
#include <stdint.h>

union uint_float
{
  uint32_t i;
  float f;
};

float calculate_machine_epsilon(float start)
{
  union uint_float v;
  v.f=start;
  v.i++;
  return v.f-start;
}

void main()
{
  printf ("\%g\n", calculate_machine_epsilon(1.0));
};
```

Was wir hier machen ist, den Bruch in der IEEE 754 Zahl als Integer zu behandeln und 1 hinzuzuaddieren. Die resultierende Fließkommazahl ist gleich starting_value + machine_epsilon, sodass wir nur den Startwert (in der Fließkommaarithmetik) abziehen müssen um zu messen, welchen Unterschied ein Bit in der einfachen Genauigkeit (float) ausmacht. Die union dient hier als Mittel, um auf die IEEE 754 Zahl als regulären Integer zuzugreifen. Die Addition von 1 entspricht hier einer Addition von 1 zum Bruch in der Zahl, aber natürlich kann hier ein Overflow auftreten, was eine Addition von 1 zum Exponenten nach sich zieht.

x86

**ARM64**

Erweitern wir unser Beispiel auf 64 Bit:

```c
#include <stdio.h>
#include <stdint.h>

typedef union
{
    uint64_t i;
    double d;
} uint_double;

double calculate_machine_epsilon(double start)
{
    uint_double v;
    v.d=start;
    v.i++;
    return v.d-start;
}

void main()
{
    printf ("%g\n", calculate_machine_epsilon(1.0));
};
```

ARM64 kennt keinen Befehl, der eine Zahl zu einem FPU D-Register addieren kann, sodass der Eingabewert (in D0) zunächst nach GPR kopiert wird, dann inkrementiert wird und schließlich in das FPU Register D1 kopiert wird, bevor die Subtraktion ausgeführt wird.

**MIPS**

Der neue Befehl ist hier MTC1 („Move To Coprocessor 1“): er überträgt Daten von GPR in die Register der FPU.
Fazit
Es ist schwer zu sagen, ob jemand eine solche Trickserei in echtem Produktivcode benötigt, aber wie bereits mehrfach erwähnt, ist dieses Beispiel gut geeignet, um das IEEE 754 Format und unions in C/C++ zu erklären.

1.22.3 FSACLE Ersatz
Agner Fog schreibt in seiner Abhandlung *Optimizing subroutines in assembly language / An optimization guide for x86 platforms*[^148], dass der Befehl FSACLE FPU (der $2^n$ berechnet) auf vielen CPUs langsam ist und bietet einen schnelleren Ersatz an.

Hier ist meine Übersetzung von seinem Assemblercode in C/C++:

```c
#include <stdint.h>
#include <stdio.h>

union uint_float
{
    uint32_t i;
    float f;
};

float flt_2n(int N)
{
    union uint_float tmp;
    tmp.i=(N<<23)+0x3f800000;
    return tmp.f;
}

struct float_as_struct
{
    unsigned int fraction : 23;
    unsigned int exponent : 8;
    unsigned int sign : 1;
};

float flt_2n_v2(int N)
{
    struct float_as_struct tmp;
    tmp.fraction=0;
    tmp.sign=0;
    tmp.exponent=N+0x7f;
    return *(float*)(&tmp);
}

union uint64_double
{
    uint64_t i;
    double d;
};

double dbl_2n(int N)
{
    union uint64_double tmp;
```

Der Befehl FSSCALE kann zwar in bestimmten Umgebungen schneller sein, ist aber vor allem ein gutes Beispiel für unions und die Tatsache, dass der Exponent in der Form $2^n$ gespeichert wird, sodass ein Eingabewert $n$ zum Exponenten nach IEEE 754 Standard verschoben wird. Der Exponent wird dann durch Addition von 0x3f800000 oder 0x3ff0000000000000 korrigiert.

Das gleiche kann ohne Verschiebung durch ein struct erreicht werden, aber intern werden stets Schiebeprobleme verwendet.

### 1.22.4 Schnelle Berechnung der Quadratwurzel

Ein anderer bekannter Algorithmus, in dem float als int interpretiert wird, ist die schnelle Berechnung einer Quadratwurzel.

Listing 1.333: Quellcode stammt aus der Wikipedia: [http://go.yurichev.com/17364](http://go.yurichev.com/17364)
return *(float*)&val_int; /* Interpret again as float */
}

Versuchen Sie als Übung, diese Funktion zu kompilieren und zu verstehen, wie sie funktioniert.

Es gibt auch einen bekannten Algorithmus zur schnellen Berechnung von \( \sqrt{\frac{1}{x}} \). Der Algorithmus wurde vermutlich so populär, weil er in Quake III Arena verwendet wurde. Eine Beschreibung des Algorithmus' findet man bei Wikipedia: http://go.yurichev.com/17360.

### 1.23 64-Bit-Werte in 32-Bit-Umgebungen

In einer 32-Bit-Umgebung sind GPR 32 Bit groß. Also werden 64-Bit-Werte in 32-Bit-Wertepaaren gespeichert und übergeben\(^\text{149}\).

#### 1.23.1 Rückgabe von 64-Bit-Werten

```c
#include <stdint.h>

uint64_t f ()
{
    return 0x1234567890ABCDEF;
}
```

#### x86

In einer 32-Bit-Umgebung werden 64-Bit-Werte von Funktionen über das Registerpaar EDX:EAX zurückgegeben:

<table>
<thead>
<tr>
<th>Listing 1.334: German text placeholder MSVC 2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC f</td>
</tr>
<tr>
<td>mov eax, -1867788817 ; 90abcdefH</td>
</tr>
<tr>
<td>mov edx, 305419896 ; 12345678H</td>
</tr>
<tr>
<td>ret 0</td>
</tr>
<tr>
<td>ENDP</td>
</tr>
</tbody>
</table>

#### ARM

Ein 64-Bit-Wert wird über das R0-R1 Registerpaar zurückgegeben (R1 enthält dabei den höheren und R0 den niederen Teil):

<table>
<thead>
<tr>
<th>Listing 1.335: German text placeholder Keil 6/2013 (German text placeholder)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROC f</td>
</tr>
<tr>
<td>LDR r0,</td>
</tr>
<tr>
<td>LDR r1,</td>
</tr>
<tr>
<td>BX lr</td>
</tr>
<tr>
<td>ENDP</td>
</tr>
</tbody>
</table>

| |L0.12| |
|---|
| DCD 0x90abcdef |

| |L0.16| |
|---|
| DCD 0x12345678 |

#### MIPS

Ein 64-Bit-Wert wird über das V0-V1 ($2$-$3$) Registerpaar zurückgegeben (V0 ($2$) enthält dabei den höheren und V1 ($3$) den niederen Teil):

<table>
<thead>
<tr>
<th>Listing 1.336: German text placeholder GCC 4.4.5 (assembly listing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>li $3,-1867841536 # 0xffffffff90ab0000</td>
</tr>
<tr>
<td>li $2, 305397760 # 0x12340000</td>
</tr>
<tr>
<td>ori $3,$3,0xabcdef</td>
</tr>
</tbody>
</table>

\(^\text{149}\)Übrigens, 32-Bit-Werte werden als Paare in 16-Bit-Umgebungen auf der gleiche Art übergeben: ?? on page ??
1.23.2 Übergabe von Argumenten bei Addition und Subtraktion

```c
#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b) {
    return a+b;
};

void f_add_test () {
    #ifdef __GNUC__
    printf("%lld\n", f_add(12345678901234, 23456789012345));
    #else
    printf("%I64d\n", f_add(12345678901234, 23456789012345));
    #endif
};

uint64_t f_sub (uint64_t a, uint64_t b) {
    return a-b;
};
```

**x86**

```assembly
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_add PROC
    mov  eax, DWORD PTR _a$[esp-4]
    add  eax, DWORD PTR _b$[esp-4]
    mov  edx, DWORD PTR _a$[esp]
    adc  edx, DWORD PTR _b$[esp]
    ret 0
_f_add ENDP

_f_add_test PROC
    push 5461 ; 00001555H
    push 1972608889 ; 75939f79H
    push 2874 ; 00000b3ah
    push 1942892530 ; 73ce2ff2H
    call _f_add
    push edx
    push eax
    push OFFSET $SG1436 ; %I64d", 0aH, 00H
    call _printf
    add  esp, 28
    ret 0
_f_add_test ENDP

_f_sub PROC
    mov  eax, DWORD PTR _a$[esp-4]
    sub  eax, DWORD PTR _b$[esp-4]
    mov  edx, DWORD PTR _a$[esp]
    sbb  edx, DWORD PTR _b$[esp]
```

Listing 1.337: German text placeholder GCC 4.4.5 (IDA)

```assembly
lui $v1, 0x90AB
lui $v0, 0x1234
li $v1, 0x90ABCDEF
jr $ra
li $v0, 0x12345678
```

Listing 1.338: German text placeholder MSVC 2012 /Ob1
Wir sehen, dass in der Funktion \texttt{f_add_test()} jeder 64-Bit-Wert über zwei 32-Bit-Werten übergeben wird: zuerst der höhere Teil, dann der niedere Teil.

Addition und Subtraktion werden auch mit Paaren ausgeführt.

Bei einer Addition werden die niederen 32-Bit-Teile zuerst addiert. Tritt hierbei ein Übertrag auf, wird das \texttt{CF} Flag gesetzt.

Der folgende \texttt{ADC} Befehl addiert die höheren Teile der Operanden und addiert 1, falls \texttt{CF} = 1.

Subtraktion wird auch mit den Wertepaaren durchgeführt. Das erste \texttt{SUB} setzt ggf. das \texttt{CF} Flag, das von dem folgenden \texttt{SBB} Befehl geprüft wird: wenn das Carryflag gesetzt ist, wird am Ende 1 vom Ergebnis abgezogen.

Man erkennt im Code leicht, wie das Ergebnis der Funktion \texttt{f_add()} an printf() übergeben wird.

\begin{verbatim}
Listing 1.339: GCC 4.8.1 -O1 -fno-inline

\_f_add:
    mov    eax, DWORD PTR [esp+12]
    mov    edx, DWORD PTR [esp+16]
    add    eax, DWORD PTR [esp+4]
    adc    edx, DWORD PTR [esp+8]
    ret

\_f_add_test:
    sub    esp, 28
    mov    DWORD PTR [esp+4], 1972608889 ; 75939f79H
    mov    DWORD PTR [esp+12], 5461 ; 00001555H
    mov    DWORD PTR [esp], 1942892530 ; 73ce2ff2H
    mov    DWORD PTR [esp+4], 2874 ; 00000b3aH
    call   \_f_add
    mov    DWORD PTR [esp+4], eax
    mov    DWORD PTR [esp+8], edx
    mov    DWORD PTR [esp], OFFSET FLAT:LC0 ; "%lld\n"
    call   _printf
    add    esp, 28
    ret

_\_sub:
    mov    eax, DWORD PTR [esp+4]
    mov    edx, DWORD PTR [esp+8]
    sub    eax, DWORD PTR [esp+12]
    sbb    edx, DWORD PTR [esp+16]
    ret

Der Code von GCC ist identisch.

\textbf{ARM}

\begin{verbatim}
Listing 1.340: German text placeholder Keil 6/2013 (German text placeholder)

_\_f_add PROC
    ADDS    r0, r0, r2
    ADC     r1, r1, r3
    BX \_\_\_f_add
ENDP

_\_f_sub PROC
    SUBS    r0, r0, r2
    SBC     r1, r1, r3
    BX \_\_\_f_sub
ENDP

_\_f_add_test PROC
    PUSH    {r4, lr}
    LDR    r2, [L0.68] ; 0x75939f79
    LDR    r3, [L0.72] ; 0x00001555
ENDP
\end{verbatim}
Der erste 64-Bit-Wert wird über das Registerpaar R0 und R1 übergeben, der zweite über R2 und R3. ARM verfügt ebenfalls über die Befehle ADC und SBC (die das Carryflag beachten). Man beachte: wenn die niederen Teile addiert bzw. subtrahiert werden, werden ADDS und SUBS Befehle mit dem Suffix -S verwendet. Dieser Suffix steht für „set flags“ (dt. setze Flags) und wird von den folgenden ADC/SBC Befehlen unbedingt benötigt. Würden die Flags nicht weiter beachtet werden, könnten hier ADD und SUB verwendet werden.

**MIPS**

Listing 1.341: German text placeholder GCC 4.4.5 (IDA)

```
f_add:
    ; $a0 - höherer Teil von a
    ; $a1 - niedriger Teil von a
    ; $a2 - höherer Teil von b
    ; $a3 - niedriger Teil von b
    addu $v1, $a3, $a1 ; addiere niedere Teile
    addu $a0, $a2, $a0 ; addiere höhere Teile
    ; wird Übertrag durch Addition der niederen Teile erzeugt?
    ; falls ja, setze $v0 auf 1
    sltu $v0, $v1, $a3
    jr $ra
    ; add 1 to high part of result if carry should be generated:
    addu $v0, $a0 ; branch delay slot
    ; $v0 - höherer Teil des Ergebnisses
    ; $v1 - niedriger Teil des Ergebnisses

f_sub:
    ; $a0 - höherer Teil von a
    ; $a1 - niedriger Teil von a
    ; $a2 - höherer Teil von b
    ; $a3 - niedriger Teil von b
    subu $v1, $a1, $a3 ; subtrahiere niedere Teile
    subu $v0, $a0, $a2 ; subtrahiere höhere Teile
    ; wird Übertrag durch Subtrahieren der niederen Teile erzeugt?
    ; falls ja, setze $a0 auf 1
    sltu $a1, $v1
    jr $ra
    ; subtrahiere 1 vom höheren Teil des Ergebnisses, falls Übertrag vorliegt:
    subu $v0, $a1 ; branch delay slot
    ; $v0 - höherer Teil des Ergebnisses
    ; $v1 - niedriger Teil des Ergebnisses

f_add_test:
    var_10 = -0x10
    var_4  = -4
    lui $gp, (__gnu_local_gp >> 16)
```
MIPS besitzt kein Register für die Flags, sodass keine derartige Information nach der Ausführung von arithmetischen Operationen verfügbar ist. Es gibt also keine Befehle wie ADC oder SBB in x86. Um zu prüfen, ob das Carryflag gesetzt werden muss, wird ein SLTU Befehl verwendet, der das Zielregister auf 1 oder 0 setzt. Diese 1 oder 0 wird dann zum Ergebnis addiert bzw. davon subtrahiert.

### 1.2.3 Multiplikation und Division

```c
#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b)
{
    return a*b;
};

uint64_t f_div (uint64_t a, uint64_t b)
{
    return a/b;
};

uint64_t f_rem (uint64_t a, uint64_t b)
{
    return a % b;
};
```

x86

Listing 1.342: German text placeholder MSVC 2013 /Ob1

```
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_mul PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _b$[ebp+4]
    push   eax
    mov    ecx, DWORD PTR _b$[ebp]
    push   ecx
    mov    edx, DWORD PTR _a$[ebp+4]
    push   edx
    mov    eax, DWORD PTR _a$[ebp]
    push   eax
    call   __allmul ; long long Multiplikation
    pop    ebp
    ret 0
_f_mul ENDP
```
Da Multiplikation und Division komplexere Operationen sind, verwendet der Compiler für deren Ausführung in der Regel Bibliotheksfunctionen.

Diese Funktionen werden hier beschrieben: .3 on page 544.

Listing 1.343: German text placeholder GCC 4.8.1 -fno-inline

```assembly
_f_div PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    push    eax
    call    __aulldiv ; long long Division ohne Vorzeichen
    pop     ebp
    ret 0
_f_div ENDP

_f_rem PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    push    eax
    call    __aullrem ; long long Rest ohne Vorzeichen
    pop     ebp
    ret 0
_f_rem ENDP
```

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
```

_f_mul:
    push    ebx
    mov     edx, DWORD PTR [esp+8]
    mov     eax, DWORD PTR [esp+16]
    mov     ebx, DWORD PTR [esp+12]
    imul    ebx, eax
    imul    ecx, edx
    mul     edx
    add     ecx, ebx
    add     edx, ecx
    pop     ebx
    ret

_f_div:
    sub     esp, 28
    mov     eax, DWORD PTR [esp+40]
    mov     edx, DWORD PTR [esp+44]
    mov     DWORD PTR [esp+8], eax
    mov     eax, DWORD PTR [esp+32]
    mov     DWORD PTR [esp+12], edx
    mov     edx, DWORD PTR [esp+36]
    mov     DWORD PTR [esp], eax
    mov     DWORD PTR [esp+4], edx
    call    ___udivdi3 ; Division ohne Vorzeichen
    add     esp, 28
    ret
```
GCC verhält sich wie erwartet, aber der Code für die Multiplikation wird direkt in der Funktion platziert und ist so effizienter. In GCC haben die Bibliotheksfunctionen andere Namen: .2 on page 544.

ARM

Keil für Thumb mode fügt Aufrufe von Routinen aus Bibliotheken ein:

Listing 1.344: German text placeholder Keil 6/2013 (German text placeholder)

```assembly
||f_mul|| PROC
  PUSH   {r4,lr}
  BL   _aeabi_lmul
  POP   {r4,pc}
ENDP

||f_div|| PROC
  PUSH   {r4,lr}
  BL   _aeabi_uldivmod
  POP   {r4,pc}
ENDP

||f_rem|| PROC
  PUSH   {r4,lr}
  BL   _aeabi_uldivmod
  MOVS   r0,r2
  MOVS   r1,r3
  POP   {r4,pc}
ENDP
```

Keil für ARM mode ist wiederum in der Lage Code für 64-Bit-Multiplikation zu erzeugen:

Listing 1.345: German text placeholder Keil 6/2013 (German text placeholder)

```assembly
||f_mul|| PROC
  PUSH   {r4,lr}
  UMULL  r12,r4,r0,r2
  MLA    r1,r2,r1,r4
  MLA    r1,r0,r3,r1
  MOV    r0,r12
  POP   {r4,pc}
ENDP

||f_div|| PROC
  PUSH   {r4,lr}
  BL   _aeabi_uldivmod
  POP   {r4,pc}
ENDP

||f_rem|| PROC
  PUSH   {r4,lr}
  BL   _aeabi_uldivmod
  MOV    r0,r2
  MOV    r1,r3
  POP   {r4,pc}
ENDP
```
MIPS

Der optimierende GCC für MIPS kann Code für 64-Bit-Multiplikation erzeugen, muss aber für die Division auf eine Programmbibliothek zurückgreifen:

Listing 1.346: German text placeholder GCC 4.4.5 (IDA)

```c
uint64_t f (uint64_t a)
{
    return a >> 7;
}
```

Hier gibt es eine Menge NOPs; möglicherweise sind dies delay slots, die nach dem Multiplikationsbefehl eingefügt wurden (diese Vorgehensweise ist jedoch langsamer als andere Befehle).

1.23.4 Verschiebung nach rechts
Das Verschieben geschieht ebenfalls zweigeteilt: zunächst wird der niedere Teil verschoben, danach der höhere. Der niedere Teil wird mithilfe des Befehls SHRD verschoben; er verschiebt den Wert in EAX um 7 Bits, holt aber die nachrutschenden Bits aus EDX, d.h. aus dem höheren Teil. Mit anderen Worten: der 64-Bit-Wert aus EDX:EAX wird als ganzes um 7 Bits verschoben und die niederen 32 Bits des Ergebnisses werden in EAX abgelegt. Der höhere Teil wird mit dem häufig verwendeten SHR Befehl verschoben, da die frei werdenden Bits im höheren Teil mit Nullen aufgefüllt werden müssen.

**ARM**

ARM verfügt im Gegensatz zu x86 nicht über einen SHRD Befehl, sodass der Keil Compiler die Aufgabe mit einer Kombination aus einfachen Schiebebefehlen und OR-Operationen durchführen muss:

**MIPS**

GCC für MIPS folgt dem gleichen Algorithmus wie Keil für Thumb mode:
1.23.5 32-Bit-Werte in 64-Bit-Werte umwandeln

```c
#include <stdint.h>

int64_t f (int32_t a)
{
    return a;
};
```

### x86

Listing 1.352: German text placeholder MSVC 2012

```
_a$ = 8
f PROC
    mov eax, DWORD PTR _a$[esp-4]
    cdq
    ret 0
f ENDP
```


### ARM

Listing 1.353: German text placeholder Keil 6/2013 (German text placeholder)

```
||f|| PROC
    ASR r1,r0,#31
    BX lr
ENDP
```

Keil für ARM geht anders vor: er verschiebt den Eingabewert um 31 Bits nach rechts. Wie wir wissen ist das Vorzeichenbit das MSB! und die arithmetische Verschiebung kopiert das Vorzeichenbit in die ausgeschobenen Bits. Nach Ausführung von „ASR r1,r0,#31“ enthält R1 also 0xFFFFFFFF, falls der Eingabewert negativ war und ansonsten 0. R1 enthält den höheren Teil des resultierenden 64-Bit-Wertes. Mit anderen Worten: dieser Code kopiert das MSB! (Vorzeichenbit) vom Eingabewert in R0 in alle Bits der höheren 32 Bits des Ergebnisses.

### MIPS

GCC für MIPS erzeugt das gleich wie Keil für ARM mode:

Listing 1.354: German text placeholder GCC 4.4.5 (IDA)

```
f:
   sra $v0, $a0, 31
   jr $ra
   move $v1, $a0
```

1.24 SIMD

SIMD ist ein Akronym: Single Instruction, Multiple Data. Wie der Name schon sagt, verarbeitet SIMD mehrere Daten in nur einem Befehl.

Wie die FPU sieht das CPU-Subsystem wie ein separater Prozessor innerhalb von x86 aus. SIMD begann als MMX in x86. 8 neue 64-Bit-Register erschienen: MM0-MM7.

Jedes MMX Register kann 2 32-Bit-Werte, 4 16-Bit-Werte oder 8 Byte aufnehmen. Es ist zum Beispiel möglich, 8 8-Bit-Werte gleichzeitig zu addieren, indem zwei Werte in MMX Registern addiert werden.

Das ist übrigens der Grund dafür, dass es in SIMB die Sättigungsbefehle gibt. Wenn der User die Helligkeit im Graphikeditor verändert, sind Überlauf und Unterlauf nicht erwünscht, weshalb es in SIMD Additionsbefehle gibt, die nicht weiter addieren, wenn der Maximalwert bereits erreicht ist, etc.

Als MMX erschien befanden sich diese Register räumlich innerhalb der FPU Register. Es war möglich entweder die FPU oder MMX zur gleichen Zeit zu benutzen. Man könnte meinen, dass Intel dieses Layout gewählt hat um Transistoren zu sparen, aber der wirkliche Grund für diese Symbiose ist trivialer —ältere Betriebssysteme, die die zusätzlichen CPU Register nicht erwarteten, hätten deren Inhalte nicht gesichert, sicherten aber sehr wohl den Inhalt der FPU Register. Dadurch funktionierte die Kombination aus MMX CPU und altem Betriebssystem, wenn MMX Features verwendet werden.


Ein weiteres Beispiel: der DES-Verschlüsselungsalgorithmus nimmt einen 64-Bit-Block und einen 56-Bit-Key, verschlüsselt den Block und erzeugt ein 64-Bit-Ergebnis. Der DES-Algorithmus kann als sehr großer Schaltkreis aus Drähten und AND/OR/NOT-Gattern aufgefasst werden.

Hinter Bitslice DES steckt die Idee, mehrere Gruppen von Blöcken und Schlüsseln simultan zu verarbeiten. Eine Variable vom Typ unsigned int umfasst in x86 beispielsweise 32 Bit, sodass es möglich ist, in ihr die Zwischenergebnisse für 32 Block-Schlüssel-Paare gleichzeitig zu speichern, indem \( 64 \div 56 \) Variablen vom Typ unsigned int verwendet werden.

Es gibt es Tool um Oracle RDBMS Passwörter und Hashes (die auf DES basieren) per Brute-Force zu knacken. Hierbei wird ein nur wenig veränderter Bitslice-DES-Algorithmus für SSE2 und AVX verwendet—nun ist es möglich, 128 oder 256 Block-Schlüssel-Paare simultan zu verschlüsseln.

http://go.yurichev.com/17313

1.24.1 Vektorisierung

Vektorisierung tritt auf, wenn man beispielsweise eine Schleife hat, die mehrere Arrays als Input und ein Array als Output hat. Der Rumpf der Schleife nimmt Werte aus den Eingabearrays, vearbeitet sie und legt das Ergebnis im Outputarray ab. Vektorisierung wird hier verwendet um mehrere Elemente gleichzeitig zu verarbeiten.


Vectorization is not very fresh technology: the author of this textbook saw it at least on the Cray Y-MP supercomputer line from 1988 when he played with its „lite“ version Cray Y-MP EL .

Ein Beispiel:

```c
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

Dieses Codefragment nimmt Elemente aus A und B, multipliziert sie und speichert das Ergebnis in C. Wenn jedes Arrayelement ein 32-Bit int ist, ist es möglich 4 Elemente aus A in ein 128-bit-XMM-Register zu laden, 4 weitere Elemente aus B in ein anderes, und dann durch die Befehle PMULLD (Multiply Packed Signed Dword Integers and Store Low Result) und PMULHW (Multiply Packed Signed Integers and Store High Result) 4 64-Bit-Produkte auf einmal zu erzeugen.

\(^{150}\)http://go.yurichev.com/17329

\(^{151}\)Wikipedia: vectorization

\(^{152}\)Er befindet sich im Museum für Supercomputer: http://go.yurichev.com/17081
Dadurch wird der Rumpf der Schleife nur \( \frac{1024}{4} = 256 \) Mal ausgeführt. Das ist nur ein Viertel der normalen Anzahl an Durchläufen und geht natürlich schneller.

**Beispiel zur Addition**

Manche Compiler können Vektorsierung in einfachen Fällen automatisch anwenden, zum Beispiel Intel C++.\(^{153}\)

Hier ist eine kleine Funktion:

```c
int f (int sz, int *ar1, int *ar2, int *ar3)
{
    for (int i=0; i<sz; i++)
        ar3[i]=ar1[i]+ar2[i];
    return 0;
}
```

**Intel C++**

Kompilieren wir das Programm mit Intel C++ 11.1.051 win32:

```"
icl intel.cpp /QaxSSE2 /Faintel.asm /Ox
```

Wir erhalten (in *IDA*) folgenden Code:

```assembly
; int __cdecl f(int, int *, int *, int *)
public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10 = dword ptr -10h
sz = dword ptr 4
ar1 = dword ptr 8
ar2 = dword ptr 0Ch
ar3 = dword ptr 10h
push edi
push esi
push ebx
push esi
mov edx, [esp+10h+sz]
test edx, edx
jle loc_15B
mov eax, [esp+10h+ar3]
 cmp edx, 6
jle loc_143
 cmp eax, [esp+10h+ar2]
 jbe short loc_36
 mov esi, [esp+10h+ar2]
 sub esi, eax
 lea ecx, ds:0[edx*4]
 neg esi
 cmp ecx, esi
 jbe short loc_55
loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
 cmp eax, [esp+10h+ar2]
 jnb loc_143
 mov esi, [esp+10h+ar2]
 sub esi, eax
 lea ecx, ds:0[edx*4]
 cmp esi, ecx
 jb loc_143
loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
 cmp eax, [esp+10h+ar1]
```

\(^{153}\)Mehr zur automatischen Vektorsierung in Intel C++ unter: *Auszug: Effektive automatische Vektorisierung*
jbe short loc_67  
mov esi, [esp+10h+ar1]  
sub esi, eax  
neg esi  
cmp ecx, esi  
jbe short loc_7F

loc_67: ; CODE XREF: f(int,int *,int *,int *)+59  
cmp eax, [esp+10h+ar1]  
jnb loc_143  
mov esi, [esp+10h+ar1]  
sub esi, eax  
cmp esi, ecx  
jb loc_7F  

loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65  
mov edi, eax  
	; edi = ar3  
and edi, 0Fh  
	; liegt ar3 auf 16-Byte-Grenze?  
jz short loc_9A  
ja loc_9A  
test edi, 3  
jnz loc_162  
neg edi  
add edi, 10h  
shr edi, 2

loc_9A: ; CODE XREF: f(int,int *,int *,int *)+84  
lea ecx, [edi+4]  
cmp edx, ecx  
jl loc_162  
mov ecx, edx  
sub ecx, edi  
and ecx, 3  
neg ecx  
add ecx, edx  
test edi, edi  
jbe short loc_D6  
mov ebx, [esp+10h+ar2]  
mov [esp+10h+var_10], ecx  
mov ecx, [esp+10h+ar1]  
xor esi, esi

loc_C1: ; CODE XREF: f(int,int *,int *,int *)+CD  
mov edx, [ecx+esi*4]  
add edx, [ebx+esi*4]  
mov [eax+esi*4], edx  
inc esi  
cmp esi, edi  
jb short loc_C1  
mov ecx, [esp+10h+var_10]  
mov edx, [esp+10h+sz]

loc_D6: ; CODE XREF: f(int,int *,int *,int *)+B2  
mov esi, [esp+10h+ar2]  
lea esi, [esi+edi*4]  
	; liegt ar2+i*4 auf 16-Byte-Grenze?  
test esi, 0Fh  
jz short loc_D6  
mov ebx, [esp+10h+ar1]  
mov esi, [esp+10h+ar2]

loc_ED: ; CODE XREF: f(int,int *,int *,int *)+105  
movdqu xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4  
movdqu xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 nicht liegt auf 16-Byte-Grenze: lade es nach XMM0  
paddd xmm1, xmm0  
movdqa xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4  
add edi, 4  
cmp edi, ecx  
jb short loc_ED  
jmp short loc_127
Die SSE2-Befehle sind:

- **MOVQQU (Move Unaligned Double Quadword)**—lädt 16 Byte aus dem Speicher in ein XMM-Register.

- **PADDD (Add Packed Integers)** addiert 4 Paare von 32-Bit-Zahlen und speichert das Ergebnis im ersten Operanden. Hier wird übrigens bei einem Überlauf keine Exception geworfen und auch keine Flags gesetzt, aber es werden dann nur die niederen 23 Bit des Ergebnisses gespeichert. Wenn einer der Operanden von PADDD die Speicheradresse eines Wertes ist, muss sich die Adresse auf einer 16-Byte-Grenze befinden. Ist dies nicht der Fall, wird eine Exception ausgelöst.

- **MOVQQA (Move Aligned Double Quadword)** entspricht MOVQQU, verlangt aber, dass die Adresse des Wertes im Speicher auf einer 16-Byte-Grenze liegt. Ist dies nicht der Fall, wird eine Exception ausgelöst. MOVQQA ist schneller als MOVQQU, hat aber die genannte zusätzliche Vorbedingung.
Diese SSE2-Befehle werden nur dann ausgeführt, wenn auf mehr als 4 Paaren gearbeitet werden muss und sich der Pointer ar3 auf einer 16-Byte-Grenze befindet.

Wenn sich ar2 ebenfalls auf einer 16-Byte-Grenze befindet, wird das folgende Codefragment ebenfalls ausgeführt:

```
movdqu xmm0, xmmword ptr [ebx+edi*4]; ar1+i*4
padd  xmm0, xmmword ptr [esi+edi*4]; ar2+i*4
movdqa xmmword ptr [eax+edi*4], xmm0; ar3+i*4
```

Ansonsten wird der Wert aus ar2 mit MOVQDU nach XMM0 geladen. MOVQDU benötigt keinen angeordneten Pointer, arbeitet aber möglicherweise langsamer:

```
movdqu xmm1, xmmword ptr [ebx+edi*4]; ar1+i*4
movdqu xmm0, xmmword ptr [esi+edi*4]; ar2+i*4 liegt nicht auf einer 16-Byte-Grenze: lade es nach XMM0
padd  xmm1, xmm0
movdqa xmmword ptr [eax+edi*4], xmm1; ar3+i*4
```

In allen anderen Fällen wird nicht-SSE2-Code ausgeführt.

**GCC**

GCC kann in einfachen Fällen ebenfalls Vektorisierung durchführen\(^{154}\), wenn die Option -O3 verwendet und SSE2 Unterstützung aktiviert ist: -msse2.

Wie erhalten den folgenden Code (GCC 4.4.1):

```
; f(int, int *, int *, int *)
    public _Z1fiPiS_S_
    _Z1fiPiS_S_ proc near

var_18 = dword ptr -18h
var_14 = dword ptr -14h
var_10 = dword ptr -10h
arg_0 = dword ptr  8
arg_4 = dword ptr  0Ch
arg_8 = dword ptr 10h
arg_C = dword ptr 14h

    push    ebp
    mov     ebp, esp
    push    edi
    push    esi
    push    ebx
    sub     esp, 0Ch
    mov     ecx, [ebp+arg_0]
    mov     esi, [ebp+arg_4]
    mov     edi, [ebp+arg_8]
    mov     ebx, [ebp+arg_C]
    test    ecx, ecx
    jle    short loc_80484D8
    cmp     ecx, 6
    lea     eax, [ebx+10h]
    ja    short loc_80484E8

loc_80484C1: ; CODE XREF: f(int, int *, int *, int *)+4B
            ; f(int, int *, int *, int *)+61 ...
    xor     eax, eax
    nop
    lea     esi, [esi+0]

loc_80484CB: ; CODE XREF: f(int, int *, int *, int *)+36
    mov     edx, [edi+eax*4]
    add     edx, [esi+eax*4]
    mov     [ebx+eax*4], edx

```

\(^{154}\)Mehr zur Unterstützung von Vektorisierung in GCC: [http://go.yurichev.com/17083](http://go.yurichev.com/17083)
add    eax, 1
cmp    eax, ecx
jnz    short loc_80484C8

loc_80484DB: ; CODE XREF: f(int,int *,int *,int *)+17
            ; f(int,int *,int *,int *)+A5
add    esp, 0Ch
xor    eax, eax
pop    ebx
pop    esi
pop    edi
pop    ebp
retn

align 8

loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
test   bl, 0Fh
jnz    short loc_80484C1
lea    edx, [esi+10h]
jmp    loc_8048503
lea    edx, [edi+10h]
cmp    ebx, edx
jbe    short loc_8048578

loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
lea    edx, [edi+10h]
cmp    ebx, edx
ja     short loc_8048503
cmp    edi, eax
jbe    short loc_80484C1

loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
mov    eax, ecx
shr    eax, 2
mov    [ebp+var_14], eax
shr    eax, 2
test   eax, eax
mov    [ebp+var_10], eax
jz     short loc_8048547
mov    [ebp+var_18], ecx
mov    ecx, [ebp+var_14]
xor    eax, eax
xor    edx, edx
nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
movdqu  xmm1, xmmword ptr [edi+eax]
movdqu  xmm0, xmmword ptr [esi+eax]
add    edx, 1
paddd  xmm0, xmm1
movdqa  xmmword ptr [ebx+eax], xmm0
add    eax, 10h
cmp    edx, ecx
jb     short loc_8048520
mov    ecx, [ebp+var_18]
mov    eax, [ebp+var_10]
cmp    ecx, eax
jz     short loc_80484D8

loc_8048547: ; CODE XREF: f(int,int *,int *,int *)+73
lea    edx, ds:0[eax*4]
add    esi, edx
add    edi, edx
add    ebx, edx
lea    esi, [esi+0]

loc_8048558: ; CODE XREF: f(int,int *,int *,int *)+CC
mov    edx, [edi]
add    eax, 1
add    edi, 4
add    edx, [esi]

373
Fast der gleiche wie Intel C++, aber nicht ganz so kleinschrittig.

**Beispiel zum Kopieren von Speicherblöcken**

Betrachten wir erneut das Beispiel zum einfachen memcpy(1.14.2 on page 174):

```c
#include <stdio.h>

void my_memcpy(unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i] = src[i];
}
```

Der optimierende GCC 4.9.1 erzeugt folgenden Code:

**Listing 1.355: German text placeholder** GCC 4.9.1 x64

```c
my_memcpy:
    ; RDI = destination address
    ; RSI = source address
    ; RDX = size of block
    test    rdx, rdx
    je      .L41
    lea     rax, [rdi+16]
    cmp     rsi, rax
    lea     rax, [rsi+16]
    setae   cl
    cmp     rdi, rax
    setae   al
    or      cl, al
    je      .L13
    cmp     rdx, 22
    jbe     .L13
    mov     rcx, rsi
    push    rbp
    push    rbx
    neg     rcx
    and     ecx, 15
    cmp     rcx, rdx
    cmova   rcx, rdx
    xor     eax, eax
    test    rcx, rcx
    je      .L4
    movzx   eax, BYTE PTR [rsi]
    cmp     rcx, 1
    mov     BYTE PTR [rdi], al
    je      .L15
    movzx   eax, BYTE PTR [rsi+1]
    cmp     rcx, 2
```
mov  BYTE PTR [rdi+1], al
je    .L16
movzx eax, BYTE PTR [rsi+2]
cmp   rcx, 3
mov  BYTE PTR [rdi+2], al
je    .L17
movzx eax, BYTE PTR [rsi+3]
cmp   rcx, 4
mov  BYTE PTR [rdi+3], al
je    .L18
movzx eax, BYTE PTR [rsi+4]
cmp   rcx, 5
mov  BYTE PTR [rdi+4], al
je    .L19
movzx eax, BYTE PTR [rsi+5]
cmp   rcx, 6
mov  BYTE PTR [rdi+5], al
je    .L20
movzx eax, BYTE PTR [rsi+6]
cmp   rcx, 7
mov  BYTE PTR [rdi+6], al
je    .L21
movzx eax, BYTE PTR [rsi+7]
cmp   rcx, 8
mov  BYTE PTR [rdi+7], al
je    .L22
movzx eax, BYTE PTR [rsi+8]
cmp   rcx, 9
mov  BYTE PTR [rdi+8], al
je    .L23
movzx eax, BYTE PTR [rsi+9]
cmp   rcx, 10
mov  BYTE PTR [rdi+9], al
je    .L24
movzx eax, BYTE PTR [rsi+10]
cmp   rcx, 11
mov  BYTE PTR [rdi+10], al
je    .L25
movzx eax, BYTE PTR [rsi+11]
cmp   rcx, 12
mov  BYTE PTR [rdi+11], al
je    .L26
movzx eax, BYTE PTR [rsi+12]
cmp   rcx, 13
mov  BYTE PTR [rdi+12], al
je    .L27
movzx eax, BYTE PTR [rsi+13]
cmp   rcx, 15
mov  BYTE PTR [rdi+13], al
jne    .L28
movzx eax, BYTE PTR [rsi+14]
mov  BYTE PTR [rdi+14], al
mov  eax, 15
.
.L4:
    lea  r9, [rdx-1]
    sub  r10, rcx
    lea  r8, [r10-16]
    sub  r9, rcx
    shr  r8, 4
    add  r8, 1
    mov  r11, r8
    sal  r11, 4
    cmp  r9, 14
    jbe  .L6
    lea  rbp, [rsi+rcx]
    xor  r9d, r9d
    add  rcx, rdi
    xor  ebx, ebx
.
.L7:
movdqa  xmm0, XMMWORD PTR [rbp+0+r9]
add    rbx, 1
movups XMMWORD PTR [rcx+r9], xmm0
add    r9, 16
cmp    rbx, r8
jb     .L7
add    rax, r11
cmp    r10, r11
je     .L1
.L6:
movzx  ecx, BYTE PTR [rsi+rax]
mov    BYTE PTR [rdi+rax], cl
lea    rcx, [rax+1]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+1+rax]
mov    BYTE PTR [rdi+1+rax], cl
lea    rcx, [rax+2]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+2+rax]
mov    BYTE PTR [rdi+2+rax], cl
lea    rcx, [rax+3]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+3+rax]
mov    BYTE PTR [rdi+3+rax], cl
lea    rcx, [rax+4]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+4+rax]
mov    BYTE PTR [rdi+4+rax], cl
lea    rcx, [rax+5]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+5+rax]
mov    BYTE PTR [rdi+5+rax], cl
lea    rcx, [rax+6]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+6+rax]
mov    BYTE PTR [rdi+6+rax], cl
lea    rcx, [rax+7]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+7+rax]
mov    BYTE PTR [rdi+7+rax], cl
lea    rcx, [rax+8]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+8+rax]
mov    BYTE PTR [rdi+8+rax], cl
lea    rcx, [rax+9]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+9+rax]
mov    BYTE PTR [rdi+9+rax], cl
lea    rcx, [rax+10]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+10+rax]
mov    BYTE PTR [rdi+10+rax], cl
lea    rcx, [rax+11]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+11+rax]
mov    BYTE PTR [rdi+11+rax], cl
lea    rcx, [rax+12]
cmp    rdx, rcx
jbe    .L1
376
movzx ecx, BYTE PTR [rsi+12+rax]
mov BYTE PTR [rdi+12+rax], cl
lea rcx, [rax+13]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+13+rax]
mov BYTE PTR [rdi+13+rax], cl
lea rcx, [rax+14]
cmp rdx, rcx
jbe .L1
movzx edx, BYTE PTR [rsi+14+rax]
mov BYTE PTR [rdi+14+rax], dl
.
.L1:
    pop rbx
    pop rbp
.
.L41:
    rep ret
.
.L13:
    xor eax, eax
.
.L3:
    movzx ecx, BYTE PTR [rsi+rax]
mov BYTE PTR [rdi+rax], cl
add rax, 1
cmp rax, rdx
jne .L3
    rep ret
.
.L28:
    mov eax, 14
    jmp .L4
.
.L15:
    mov eax, 1
    jmp .L4
.
.L16:
    mov eax, 2
    jmp .L4
.
.L17:
    mov eax, 3
    jmp .L4
.
.L18:
    mov eax, 4
    jmp .L4
.
.L19:
    mov eax, 5
    jmp .L4
.
.L20:
    mov eax, 6
    jmp .L4
.
.L21:
    mov eax, 7
    jmp .L4
.
.L22:
    mov eax, 8
    jmp .L4
.
.L23:
    mov eax, 9
    jmp .L4
.
.L24:
    mov eax, 10
    jmp .L4
.
.L25:
    mov eax, 11
    jmp .L4
.
.L26:
    mov eax, 12
    jmp .L4
.
.L27:
    mov eax, 13
    jmp .L4
1.24.2 SIMD strlen() Implementierung

Man beachte, dass die SIMD Befehle über spezielle Makros in den C/C++ Code eingefügt werden können. Einige dieser Makros für MSVC befinden sich in der Datei intrin.h.

Es ist möglich die Funktion `strlen()` mit SIMD Befehlen zu implementieren, sodass sie 2-2.5-mal schneller als die normale Implementierung arbeitet. Diese Funktion lädt 16 Zeichen in ein XMM-Register und prüft jedes auf Null.

```c
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s = str;
    bool str_is_aligned = (((unsigned int)s) & 0xFFFFFFF0) == (unsigned int)s;
    if (str_is_aligned == false)
        return strlen(str);
    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
    int mask = 0;
    for (;;)
    {
        xmm1 = _mm_load_si128((__m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if (((mask = _mm_movemask_epi8(xmm1)) != 0) ||
            { unsigned long pos;
                _BitScanForward(&pos, mask);
                len += (size_t)pos;
            }
        )
            s += sizeof(__m128i);
        len += sizeof(__m128i);
    }
    return len;
}
```

Kompilieren wir das Beispiel in MSVC 2010 mit der Option /Ox:

**Listing 1.356: German text placeholder MSVC 2010**

```assembly
_pos$75552 = -4 ; size = 4
_str$ = 8 ; size = 4
strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2
    push ebp
    mov ebp, esp
    and esp, -16 ; fffffff0H
    mov eax, DWORD PTR _str$[ebp]
    sub esp, 12 ; 0000000cH
    push esi
    mov esi, eax
    and esi, -16 ; fffffff0H
    xor edx, edx
    mov ecx, eax
    cmp esi, eax
    je SHORT $L14@strlen_sse
    lea edx, DWORD PTR [eax+1]
    npad 3 ; align next label
_ll1@strlen_sse:  
mov _cl, BYTE PTR [eax]
    inc eax
    test cl, cl
```

---

155 MSDN: MMX, SSE, und SSE2 Intrinsics
156 `strlen()` — Funktion der Standard-C-Bibliothek, mit der die Länge eines Strings berechnet wird
Um die Funktionsweise zu verstehen müssen wir uns zunächst den Zweck der Funktion klarmachen. Sie berechnet die Länge eines C-Strings; anders ausgedrückt: sie sucht nach dem Nullbyte und berechnet dann dessen Position relativ zum Anfang des Strings.

Zunächst prüfen wir, ob der str Pointer auf einer 16-Byte-Grenze liegt. Wenn nicht, rufen wir die normale Implementierung von strlen() auf.

Danach laden wir die nächsten 16 Byte mit MOVQQA in das Register XMM1.

Der aufmerksame Leser fragt sich vielleicht, warum hier MOVQQA nicht verwendet wird, da dieser Befehl die Daten auch dann aus dem Speicher laden kann, wenn der Pointer nicht auf einer 16-Byte-Grenze liegt. Man könnte es wie folgt lösen: wenn der Pointer entsprechend angeordnet ist, verwenden MOVQQA zum Laden, ansonsten den langsameren Befehl MOVQQU.

Aber an dieser Stelle lauert ein weiterer Fallstrick:

In den Betriebssystemen der Reihe German text placeholder (aber nicht nur dort) wird Speicher in Seiten von 4KiB (4096 Byte) reserviert. Jeder win32-Prozess hat 4GiB zur Verfügung, auch wenn tatsächlich nur einige Teile des Adressraumes wirklich mit dem physischen Speicher verbunden sind. Wenn der Prozess auf einen nicht vorhandenen Speicherblock zugreift, wird eine Exception ausgelöst. So arbeitet VM.

Eine Funktion, die 16 Byte auf einmal lädt, könnte also eine solche Grenze im Speicherblock übertreten. Nehmen wir an, das Betriebssystem hat 8192 (0x2000) Byte ab der Adresse 0x008c000 reserviert. Dieser Block umfasst also die Bytes von Adresse 0x08c0000 bis einschließlich 0x008c1fff.

Hinter diesen Block, d.h. ab Adresse 0x008c2000 befindet sich nichts, d.h. das Betriebssystem hat hier keinen weiteren Speicher reserviert. Jeder Versuch auf diesen Speicherbereich zuzugreifen, wird eine Exception auslösen.

Betrachten wir weiter das Beispiel, in dem ein Programm einen String enthält, der fast am Ende des Blocks 5 Zeichen belegt. Das ist zunächst nichts Schlimmes.

158 wikipedia
Unter normalen Umständen ruft das Programm `strlen()` auf und übergibt einen Pointer auf den String 'hello', der an der Speicheradresse 0x000c1ff0 liegt. `strlen()` liest ein Byte zur Zeit bis zur Adresse 0x000c1ffd, an der sich ein Nullbyte befindet, und hält dann an.

Wenn unser `strlen()` 16 Byte auf einmal von irgendeiner Startadresse aus liest, richtig angeordnet oder nicht, könnte MOVQ versuchen 16 Byte auf einmal aus dem Adressraum 0x000c1ff0 bis 0x000c2000 zu lesen und eine Exception würde ausgelöst. Dies gilt es natürlich zu vermeiden.

Wir arbeiten also nur mit Adressen, die auf einer 16-Byte-Grenze liegen, was uns in Kombination mit dem Wissen, dass die Seitengröße des Betriebssystems normalerweise ebenfalls so angeordnet ist, eine gewisse Sicherheit darü bietet, dass unsere Funktion nicht aus dem nicht reservierten Speicher zu lesen versucht.

Zurück zu unserer Funktion.

__mm_setzero_si128()—ist ein Makro, das `pxor xmm0, xmm0` erzeugt —es löscht das XMM0 Register.

__mm_load_si128()—ist ein Makro für MOVQ: es lädt 16 Bytes von der Adresse in das XMM1 Register.

__mm_cmpeq_epi8()—ist ein Makro für PCMPEQB, ein Befehl, der zwei XMM-Register byteweise miteinander vergleicht.

Wenn ein Byte das gleiche ist wie im anderen Register, wird im Ergebnis hier `0xff` stehen, ansonsten aber `0`.

Ein Beispiel:

XMM1: 0x11223344556677880000000000000000
XMM0: 0x11ab3444007877881111111111111111

Nach der Ausführung von pcmpeqb xmm1, xmm0, enthält das XMM1 Register:

XMM1: 0xff0000ff0000ffff0000000000000000

In unserem Fall vergleicht der Befehl jeden 16-Byte-Block mit einem Block aus 16 Nullbytes, der durch `pxor xmm0, xmm0` im XMM0 Register angelegt wurde.

Das nächste Makro ist __mm movimiento_epi8() —das ist der Befehl PMOVMSKB.

Es ist sehr nützlich im Zusammenspiel mit PCMPEQB.

pmovmskb eax, xmm1 Dieser Befehl setzt das erste Bit in EAX auf 1, falls das MSB des ersten Bytes in XMM1 gleich 1 ist. Mit anderen Worten: wenn das erste Byte im XMM1 Register `0xff` ist, dann wird das erste Bit in EAX ebenfalls auf 1 gesetzt.

Wenn das zweite Byte in XMM1 `0xff` ist, wird das zweite Bit in EAX auf 1 gesetzt. Mit anderen Worten: der Befehl beantwortet die Frage welche Bytes in XMM1 gleich `0xff` sind und gibt 16 Bits im EAX Register zurück. Die anderen Bits im in EAX werden gelöscht.

Vergessen wir aber nicht die Eigenart in unserem Algorithmus. Es könnte im Input 16 Byte wie die folgenden geben:

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>'h'</td>
<td>'e'</td>
<td>'l'</td>
<td>'l'</td>
<td>'o'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Das ist der String 'hello', eine abschließende Null und ein paar Zufallsweite aus dem Speicher.

Wenn wir diese 16 Byte nach XMM1 laden und mit dem genullten XMM0 vergleichen, erhalten wir als Ergebnis etwas, das so ähnlich ist wie der folgende Output: Das bedeutet, dass der Befehl zwei Nullbytes gefunden hat. Dieses Ergebnis war zu erwarten.

159hier wird eine Ordnung vom MSB! zum LSB! verwendet
PMOVMSKB in our case will set EAX to\n\n0b0010000000100000. Offensichtlich muss unsere Funktion nur das erste Nullbit nehmen und den Rest
ignorieren.

Der nächste Befehl ist BSF (Bit Scan Forward). Dieser Befehl erkennt, dass das erste Bit auf 1 gesetzt ist
und speichert dessen Position im ersten Operanden.

EAX=0b0010000000100000

Nach der Ausführung von bsf eax, eax enthält EAX 5, was bedeutet, dass 1 an der 5. Stelle (von 0 aus
gezählt) gefunden wurde.

MSVC verfügt über ein Makro für diesen Befehl: _BitScanForward.

Der Rest ist einfach. Wenn ein Nullbyte gefunden wurde, wird dessen Position zum Zähler hinzuaddiert
und wir erhalten den Rückgabewert.

Das ist fast alles.

Hier ist bemerkenswert, dass der MSVC Compiler zur Optimierung zwei Schleifenrumpfe nebeneinander
erzeugt hat.

SSE 4.2 (erschienen im Intel Core i7) verfügt über noch mehr Befehle, sodass diese Stringmanipulationen
noch einfacher bewerkstelligt werden können: http://go.yurichev.com/17331

1.25 64 Bit

1.25.1 x86-64

Es handelt sich um eine 64-Bit-Erweiterung der x86 Architektur.

Aus Sicht eines Reverse Engineers sind die wichtigsten Veränderungen die folgenden:

- Fast alle Regsiter (außer FPU und SIMD) wurden auf 64 Bit erweitert und erhielten den Präfix R-. 8
  zusätzliche Register wurden hinzugefügt. Die GPR sind jetzt: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI,
  R8, R9, R10, R11, R12, R13, R14, R15. Es ist immer noch möglich die älteren Registerteile wie gewohnt
  anzusprechen. Zum Beispiel ist es möglich, auf den niederen 32-Bit-Teil von RAX mit EAX zuzugreifen:

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
<th>German text placeholder</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td></td>
</tr>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>AH</td>
<td>AL</td>
</tr>
</tbody>
</table>

Die neuen R8-R15 Register besitzen ebenfalls niedere Teile: R8D-R15D (niedere 32 Bit), R8W-R15W
(niedere 16 Bit), R8L-R15L (niedere 8 Bit).

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
<th>German text placeholder</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
<td></td>
</tr>
<tr>
<td>R8D</td>
<td></td>
</tr>
<tr>
<td>R8W</td>
<td></td>
</tr>
<tr>
<td>R8L</td>
<td></td>
</tr>
</tbody>
</table>

Die Anzahl der SIMD Register wurde von 8 auf 16 erhöht: XMM0-XMM15.

- In Win64 ist die Konvention zum Aufruf von Funktionen ein wenig verändert worden und erinnert
  nun an fastcall (6.1.3 on page 449). Die ersten vier Argumente werden in den RCX, RDX, R8 und
  R9 Registern gespeichert —der Rest auf dem Stack. Die aufrufende Funktion muss also 32 Byte
  reservieren, sodass der Aufruf der die ersten 4 Argumente dort speichern und sie für eigene Zwecke
  verwenden kann. Kurze Funktionen können Argumente verwenden, die ausschließlich aus Registern
  stammen, aber größere müssen ihre Variablen auf dem Stack speichern.

System V AMD64 ABI (Linux, *BSD, Mac OS X)[Michael Matz, Jan Hubicka, Andreas Jaeger, Mark
Mitchell, System V Application Binary Interface. AMD64 Architecture Processor Supplement, (2013)]

161erinnert auch an fastcall: es verwendet die 6 Register RDI, RSI, RDX, RCX, R8, R9 für die ersten 6
Argumente. Der Rest wird wie gehabt über den Stack übergeben.

Siehe dazu auch den Abschnitt über Aufrufkonventionen (6.1 on page 448).

161German text placeholder https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf
• Der Typ int in C/C++ hat aus Kompatibilitätsgründen immernoch die Größe 32 Bit.
• Alle Pointer haben jetzt eine Größe von 64 Bit.

Da sich die Anzahl der Register verdoppelt hat, hat der Compiler mehr Spielraum für den . Für uns hat dies zur Folge, dass der erzeugte Code eine geringere Anzahl lokaler Variablen enthält.

Die Funktion, die die erste S-Box im DES Verschlüsselungsalgorithmus berechnet, verarbeitet beispielsweise 32/64/128/256 Werte auf einmal (abhängig vom DES_type: uint32, uint64, SSE2 oder AVX)) mithilfe der Bitslice DES Methode (mehr zu dieser Technik unter (1.24 on page 368)):

```c
#include <stdio.h>

void s1 (DES_type a1, DES_type a2, DES_type a3, DES_type a4, DES_type a5, DES_type a6, DES_type *out1, DES_type *out2, DES_type *out3, DES_type *out4) {

  DES_type x1, x2, x3, x4, x5, x6, x7, x8;
  DES_type x9, x10, x11, x12, x13, x14, x15, x16;
  DES_type x17, x18, x19, x20, x21, x22, x23, x24;
  DES_type x25, x26, x27, x28, x29, x30, x31, x32;
  DES_type x33, x34, x35, x36, x37, x38, x39, x40;
  DES_type x41, x42, x43, x44, x45, x46, x47, x48;
  DES_type x49, x50, x51, x52, x53, x54, x55, x56;

  x1 = a3 & ~a5;
  x2 = x1 ^ a4;
  x3 = a3 & ~a4;
  x4 = x3 | a5;
  x5 = a6 & x4;
  x6 = x2 ^ x5;
  x7 = a4 & ~a5;
  x8 = a3 ^ a4;
  x9 = a6 & ~x8;
  x10 = x7 ^ x9;
  x11 = a2 | x10;
  x12 = x6 ^ x11;
  x13 = a5 ^ x5;
  x14 = x13 & x8;
  x15 = a5 & ~a4;
  x16 = x3 ^ x14;
  x17 = a6 | x16;
  x18 = x15 ^ x17;
  x19 = a2 | x18;
  x20 = x14 ^ x19;
  x21 = a1 & x20;
  x22 = x12 ^ ~x21;
  *out2 ^= x22;
  x23 = x1 | x5;
} /*
 * Generated S-box files.
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 * Produced by Matthew Kwan - March 1998 */
```

382
Es gibt eine Menge lokaler Variablen. Natürlich werden diese nicht alle auf dem lokalen Stack gespeichert.

Kompilieren wir mit MSVC 2008 mit der Option 0x:

Listing 1.357: German text placeholder MSVC 2008

PUBLIC _s1
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_x6$ = -20 ; size = 4
_x3$ = -16 ; size = 4
_x1$ = -12 ; size = 4
_x8$ = -8 ; size = 4
_x4$ = -4 ; size = 4
_a1$ = 8 ; size = 4
_a2$ = 12 ; size = 4
_a3$ = 16 ; size = 4
_x33$ = 20 ; size = 4
_x7$ = 20 ; size = 4
_a4$ = 20 ; size = 4
_a5$ = 24 ; size = 4
tv326 = 28 ; size = 4
_x36$ = 28 ; size = 4
_x28$ = 28 ; size = 4
_a6$ = 28 ; size = 4
_out1$ = 32 ; size = 4
_x24$ = 36 ; size = 4
_out2$ = 36 ; size = 4
_out3$ = 40 ; size = 4
_out4$ = 44 ; size = 4

_s1 PROC
    sub esp, 20 ; 00000014H
    mov edx, DWORD PTR _a5$[esp+16]
push ebx
mov ebx, DWORD PTR _a4$[esp+20]
push ebp
push esi
mov esi, DWORD PTR _a3$[esp+28]
push edi
mov edi, ebx
not edi
mov ebp, edi
and edi, DWORD PTR _a5$[esp+32]
mov ecx, edx
not ecx
and ebp, esi
mov eax, ecx
and eax, esi
and ecx, ebx
mov DWORD PTR _x1$[esp+36], eax
xor eax, ebx
mov esi, ebp
or esi, edx
mov DWORD PTR _x4$[esp+36], esi
and esi, DWORD PTR _a6$[esp+32]
mov DWORD PTR _x7$[esp+32], ecx
mov edx, esi
xor edx, eax
mov DWORD PTR _x6$[esp+36], edx
mov edx, DWORD PTR _a3$[esp+32]
xor edx, ebx
mov ebx, esi
xor ebx, DWORD PTR _a5$[esp+32]
mov DWORD PTR _x8$[esp+36], edx
and ebx, edx
mov ecx, edx
mov edx, ebx
xor edx, ebp
or edx, DWORD PTR _a6$[esp+32]
not ecx
and ecx, DWORD PTR _a6$[esp+32]
xor edx, edi
mov edi, edx
or edi, DWORD PTR _a2$[esp+32]
mov DWORD PTR _x3$[esp+36], ebp
mov ebp, DWORD PTR _a2$[esp+32]
xor edi, ebx
and edi, DWORD PTR _a1$[esp+32]
mov ebx, ecx
xor ebx, DWORD PTR _x7$[esp+32]
not edi
or ebx, ebp
xor edi, ebx
mov ebx, edi
mov edi, DWORD PTR _out2$[esp+32]
xor ebx, DWORD PTR [edi]
not eax
xor ebx, DWORD PTR _x6$[esp+36]
and eax, edx
mov DWORD PTR [edi], ebx
mov ebx, DWORD PTR _x7$[esp+32]
or ebx, DWORD PTR _x6$[esp+36]
mov edi, esi
or edi, DWORD PTR _x1$[esp+36]
mov DWORD PTR _x28$[esp+32], ebx
xor edi, DWORD PTR _x8$[esp+36]
mov DWORD PTR _x24$[esp+32], edi
xor edi, ecx
not edi
and edi, edx
mov ebx, edi
and ebx, ebp
xor ebx, DWORD PTR _x28$[esp+32]
Der Compiler legt auf dem lokalen Stack 5 Variablen an.

Untersuchen wir dasselbe Programm in der 64-Bit-Version mit MSVC 2008:

Listing 1.358: German text placeholder MSVC 2008

a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$s1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104  
out2$ = 112  
out3$ = 120  
out4$ = 128  
s1 PROC
$LN3:  
    mov  QWORD PTR [rsp+24], rbx
    mov  QWORD PTR [rsp+32], rbp
    mov  QWORD PTR [rsp+16], rdx
    mov  QWORD PTR [rsp+8], rcx

    push rsi
    push rdi
    push r12
    push r13
    push r14
    push r15

    mov  r15, QWORD PTR a5$[rsp]
    mov  rcx, QWORD PTR a6$[rsp]
    mov  rbp, r8
    mov  r10, r9
    mov  rax, r15
    mov  rdx, rbp
    not  rax
    xor  rdx, r9
    not  r10
    mov  r11, rax
    and  rax, r9
    mov  rsi, r10

    mov  QWORD PTR x36$1$[rsp], rax
    and  r11, r8
    and  rsi, r8
    and  r10, r15
    mov  r13, rdx

    xor  rbx, r11
    xor  rbx, r9

    mov  r9, QWORD PTR a2$[rsp]
    mov  r12, rsi
    or   r12, r15
    not  r13
    and  r13, rcx
    mov  r14, r12
    and  r14, rcx
    mov  rax, r14
    mov  r8, r14
    xor  r8, rbx

    xor  rax, r15
    not  rbx

    and  rax, rdx
    mov  rdi, rax
    xor  rdi, rsi
    or   rdi, rcx
    xor  rdi, r10
    and  rbx, rdi

    mov  rcx, rdi
    or   rcx, r9
    xor  rcx, rax
    mov  rax, r13
    xor  rax, QWORD PTR x36$1$[rsp]
    and  rcx, QWORD PTR a1$[rsp]
    or   rax, r9
    not  rcx
    xor  rcx, rax

    mov  rax, QWORD PTR out2$[rsp]
    xor  rcx, QWORD PTR [rax]
    xor  rcx, r8

    mov  QWORD PTR [rax], rcx
    mov  rax, QWORD PTR x36$1$[rsp]
    mov  rcx, r14
    or   rax, r8
    or   rcx, r11
Hier wurde im lokalen Stack nichts durch den Compiler angelegt, x36 ist ein Synonym für a5. Es gibt übrigens CPUs mit viel mehr GPRs, z.B. Itanium (128 Register).

1.25.2 ARM

64-Bit-Befehle erschienen in ARMv8.

1.25.3 Fließkommazahlen


1.25.4 Kritik an der 64-Bit-Architektur

Einige Leute sind manchmal am Zweifeln bezüglich der Vorteil von 64-Bit-Architekturen: man benötigt nun doppelt soviel Platz für das Speichern von Pointern, inklusive Cachespeicher, obwohl die x64 CPUs nur 48 Bit an externem RAM adressieren können.

Einige leute schrieben ihre eigenen Speicherreservierungen.

Interessant ist dabei der Fall CryptoMiniSat\(^\text{162}\). Dieses Programm benötigt fast nie mehr als 4GiB RAM, macht aber starken Gebrauch von Pointern. Es benötigt also auf einer 32-Bit-Architektur weniger Speicher als unter einer 64-Bit-Architektur. Um dieses Problem zu lösen, hat der Autor seine eigene Speicherreservierung (in den Dateien clauseallocator.(h|cpp)) geschrieben, welche erlaubt, Speicher mit 32-Bit-Identifiern anstelle von 64-Bit-Pointern zu reservieren.

1.26 Arbeiten mit Fließkommazahlen und SIMD

Natürlich verblieb die FPU in x86-kompatiblen Prozessoren als die SIMD Erweiterungen hinzugefügt wurden. Die SIMD Erweiterungen (SSE2) bieten einen einfacheren Weg um mit Fließkommazahlen zu arbeiten. Das Zahlenformat bleibt dabei das gleiche (IEEE 754).

Moderne Compiler (inklusive derer für x86-64) verwenden normalerweise SIMD Befehle anstatt FPU-Befehle. Wir werden hier die Beispiele aus dem Abschnitt über die FPU recyclen: 1.17 on page 195.

1.26.1 Ein einfaches Beispiel

```c
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
};

int main()
{
    printf ("%f\n", f(1.2, 3.4));
};
```

x64

Listing 1.359: German text placeholder MSVC 2012 x64

__real@4010666666666666666 DQ 0401066666666666666r ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14

a$ = 8
b$ = 16
f PROC

\(^{162}\)https://github.com/msoos/cryptominisat/
Die eingegebenen Fließkommawerte werden in die Register XMM0–XMM3 übergeben und der Rest über den Stack.

\(a\) wird nach XMM0 übergeben und \(b\) nach XMM1.

Die XMM Register sind 128 Bit breit (wie wir bereits aus dem Abschnitt über SIMD wissen: 1.24 on page 367), aber die double Werte umfassen nur 64 Bit, sodass nur die niedere Hälfte der Register benötigt wird.

DIVSD ist ein SSE-Befehl, der für „Divide Scalar Double-Precision Floating-Point Values“ steht; er teilt einen double Wert durch einen anderen, wobei beide in den niederer Hälften der Operanden gespeichert sind.

Die Konstanten werden durch den Compiler im IEEE 754 Format kodiert.

MULSD und ADDSD funktionieren genauso und führen Multiplikation und Addition durch.

Das Ergebnis der Funktionsausführung ist von Typ double und wird im XMM0 Register abgelegt.

So arbeitet der nicht optimierende MSVC:

Listing 1.360: MSVC 2012 x64

```
divsd xmm0, QWORD PTR __real@40091eb851eb851f
mulsd xmm1, QWORD PTR __real@4010666666666666
addsd xmm0, xmm1
ret 0
```

Ein wenig redundant. Die Eingabewerte bzw. deren niedereRegisterhälften, d.h. die 64-Bit-Werte vom Typ double, werden im „shadow space“( 1.9.2 on page 84) gespeichert. GCC erzeugt identischen Code.

**x86**

Kompilieren wir das Beispiel für x86. Obwohl der Code für x86 erzeugt wird, verwendet MSVC 2012 SSE2 Befehle:

Listing 1.361: German text placeholder MSVC 2012 x86

```
tv70 = -8 ; size = 8
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f PROC
    push ebp
    mov ebp, esp
    sub esp, 8
    movsd xmm0, QWORD PTR a$[ebp]
    divsd xmm0, QWORD PTR __real@40091eb851eb851f
    movsd xmm1, QWORD PTR b$[ebp]
    mulsd xmm1, QWORD PTR __real@4010666666666666
    addsd xmm0, xmm1
    fld QWORD PTR tv70[ebp]
    mov esp, ebp
    pop ebp
    ret 0
_f ENDP
```

389
Listing 1.362: German text placeholder MSVC 2012 x86

tv67 = 8 ; size = 8
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8

PROC

movsd xmm1, QWORD PTR _a$[esp-4]
divsd xmm1, QWORD PTR __real@40091eb851eb851f
movsd xmm0, QWORD PTR _b$[esp-4]
mulsd xmm0, QWORD PTR __real@4010666666666666
addsd xmm1, xmm0
movsd QWORD PTR tv67[esp-4], xmm1
fld QWORD PTR tv67[esp-4]
ret 0

ENDP

Es ist fast der gleiche Code, aber es gibt einige Unterschiede in Bezug auf die Aufrufkonventionen: 1) die Argumente werden nicht über die XMM Register, sondern über den Stack übergeben, wie in den FPU Beispielen (1.17 on page 195); 2) das Ergebnis der Funktion wird über ST(0) zurückgegeben—um dies zu erreichen wird es (durch die lokale Variable tv) aus einem der XMM Register nach ST(0) kopiert.
Untersuchen wir das optimierte Beispiel mit OllyDbg:

Abbildung 1.110: OllyDbg: MOVSD lädt den Wert von $a$ nach XMM1
Abbildung 1.111: OllyDbg: DIVSD hat den Quotienten berechnet und in XMM1 gespeichert
Abbildung 1.112: OllyDbg: MULSD hat das Produkt berechnet und in XMM0 gespeichert
Abbildung 1.113: OllyDbg: ADDSD addiert den Wert in XMM0 zu XMM1
Wir sehen, dass OllyDbg die XMM Register als Paare von double Zahlen anzeigt, aber nur der niedere Teil davon verwendet wird.

Offenbar zeigt OllyDbg sie in diesem Format an, weil die SSE2 Befehle (die mit dem Suffix -SD) jetzt ausgeführt werden.

Natürlich ist es auch möglich das Registerformat zu ändern und sich die Inhalte als 4 float-Zahlen oder nur als 16 Byte anzeigen zu lassen.
1.26.2 Fließkommazahlen als Argumente übergeben

```c
#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));
    return 0;
}
```

Sie werden über die niederen Hälften der Register XMM0-XMM3 übergeben.

Listing 1.363: German text placeholder MSVC 2012 x64

```asm
$SG1354 DB '32.01 ^ 1.54 = %lf', 0AH, 00H

__real@40400147ae147ae1 r DQ 040400147ae147ae1r ; 32.01
__real@3ff8a3d70a3d70a4 r DQ 03ff8a3d70a3d70a4r ; 1.54

main PROC
    sub rsp, 40 ; 00000028H
    movsd xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
    movsd xmm0, QWORD PTR __real@40400147ae147ae1
    call pow
    lea rcx, OFFSET FLAT:$SG1354
    call printf
    ret
main ENDP
```

Es gibt in Intel keinen MOVSDX Befehl und AMD-Handbücher (11.1.4 on page 520) bezeichnen ihn nur mit MOVSD. Es gibt insgesamt zwei Befehle, die sich in x86 denselben Namen teilen (für den anderen siehe: .1.6 on page 535). Offenbar wollten die Microsoft Entwickler hier aufräumen und haben ihn deshalb in MOVSDX umbenannt. Er lädt lediglich einen Wert in die niedere Hälfte eines XMM Registers.

`pow()` nimmt Argumente aus XMM0 und XMM1 und gibt das Ergebnis in XMM0 zurück. Es wird dann für `printf()` nach RDX verschoben. Der Grund dafür ist möglicherweise, dass `printf()` eine Funktion mit einer variablen Anzahl an Argumenten ist.

Listing 1.364: German text placeholder GCC 4.4.6 x64

```asm
.LC2:
    .string "32.01 ^ 1.54 = %lf\n"

main:
    sub rsp, 8
    movsd xmm1, QWORD PTR .LC0[rip]
    movsd xmm0, QWORD PTR .LC1[rip]
    call pow
    ; result is now in XMM0
    mov edi, OFFSET FLAT:.LC2
    mov eax, 1 ; Anzahl der übergebenen Vektorregister
    call printf
    xor eax, eax
    add rsp, 8
    ret

.LC0:
    .long 171798692
    .long 1073259479

.LC1:
    .long 2920577761
    .long 1077936455
```

GCC erzeugt klarer strukturierten Output. Der Wert für `printf()` wird in XMM0 übergeben. Hier ist übrigens ein Fall, in dem 1 nach EAX für `printf()` geschrieben wird um anzuzeigen, dass ein Argument in den Vektorregistern übergeben wird, genau wie es der Standard [Michael Matz, Jan Hubicka, Andreas Jaeger,
1.26.3 Beispiel mit Vergleich

```c
#include <stdio.h>

double d_max(double a, double b)
{
    if (a > b)
        return a;
    return b;
};

int main()
{
    printf ("%f\n", d_max(1.2, 3.4));
    printf ("%f\n", d_max(5.6, -4));
};
```

**x64**

Listing 1.365: German text placeholder MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max
PROC
    comisd xmm0, xmm1
    ja SHORT $LN2@d_max
    movaps xmm0, xmm1
$LN2@d_max:
    fatret 0
    jmp SHORT $LN2@d_max
$LN1@d_max:
    movsdx QWORD PTR [rsp+16], xmm1
    movsdx QWORD PTR [rsp+8], xmm0
    movsdx xmm0, QWORD PTR a$[rsp]
    movsdx xmm0, QWORD PTR b$[rsp]
    jbe SHORT $LN1@d_max
$LN2@d_max:
    fatret 0
    jmp SHORT $LN2@d_max

ENDP
```

**German text placeholder** MSVC erzeugt leicht verständlichen Code.

COMISD bedeutet „Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS“. Das beschreibt ziemlich genau, was der Befehl tatsächlich tut.

**German text placeholder** MSVC erzeugt Code mit mehr Redundanzen, der aber immer noch gut verständlich ist:

Listing 1.366: MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max
PROC
    movsdx QWORD PTR [rsp+16], xmm1
    movsdx QWORD PTR [rsp+8], xmm0
    movsdx xmm0, QWORD PTR a$[rsp]
    comisd xmm0, QWORD PTR b$[rsp]
    jbe SHORT $LN1@d_max
    movsdx xmm0, QWORD PTR a$[rsp]
    jmp SHORT $LN2@d_max
$LN1@d_max:
    movsdx xmm0, QWORD PTR b$[rsp]
$LN2@d_max:
    fatret 0
    jmp SHORT $LN2@d_max

ENDP
```

GCC 4.4.6 hat mehr Optimierungen durchgeführt und den Befehl MAXSD („Return Maximum Scalar Double-Precision Floating-Point Value“) verwendet, der einfach den größten Wert auswählt!

Listing 1.367: German text placeholder GCC 4.4.6 x64
maxsd  xmm0, xmm1
ret
Kompiliieren wir dieses Beispiel in MSVC 2012 mit aktiverer Optimierung:

```
Listing 1.368: German text placeholder MSVC 2012 x86

a$ = 8 ; size = 8
b$ = 16 ; size = 8

d_max PROC
    movsd xmm0, QWORD PTR a$[esp-4]
    comisd xmm0, QWORD PTR b$[esp-4]
    jbe SHORT $LN1@d_max
    fld QWORD PTR a$[esp-4]
    ret 0

$LN1@d_max:
    fld QWORD PTR b$[esp-4]
    ret 0

d_max ENDP
```

Fast identisch, nur dass die Werte \( a \) und \( b \) vom Stack geholt werden und das Funktionsergebnis in ST(0) gelassen wird.

Wenn wir dieses Beispiel in OllyDbg laden, erkennen wir, wie der Befehl `COMISD` Werte vergleicht und die CF und PF Flags setzt bzw. löscht:

Abbildung 1.115: OllyDbg: COMISD hat die CF und PF Flags verändert

### 1.26.4 Berechnen der Maschinengenauigkeit: x64 und SIMD

Betrachten wir erneut das Beispiel zur Berechnung der Maschinengenauigkeit für `double` Listing.1.22.2. Wir kompilieren es jetzt für x64:
v$ = 8
calculate_machine_epsilon PROC
movsdx QWORD PTR v$[rsp], xmm0
movaps xmm1, xmm0
inc QWORD PTR v$[rsp]
movsdx xmm0, QWORD PTR v$[rsp]
subsd xmm0, xmm1
ret 0
calculate_machine_epsilon ENDP

Es gibt keinen Weg 1 zum einem Wert in einem 128-Bit XMM Register zu addieren, also muss der Wert im Speicher abgelegt werden.

Es gibt zwar den Befehl ADDSD (Add Scalar Double-Precision Floating-Point Values), der einen Wert zu niederen 64-Bit-Hälfte eines XMM Registers addieren kann und die höheren Bits ignoriert, aber MSVC 2012 scheint an dieser Stelle nicht gut genug zu sein, um diese Möglichkeit zu erkennen.\(^\text{164}\)

Nichtsdestotrotz wird der Wert dann wieder in ein XMM Register geladen und eine Subtraktion wird durchgeführt. SUBSD steht für „Subtract Scalar Double-Precision Floating-Point Values“, d.h. er arbeitet nur auf dem niederen 64-Bit-Teil des 128-Bit XMM Registers. Das Ergebnis wird in das XMM0-Register zurückgegeben.

### Erneute Betrachtung des Beispiels zum Pseudzufallszahlengenerator

Betrachten wir erneut das Beispiel zum Pseudzufallszahlengenerator Listing.1.22.1.

Wenn wir es in MSVC 2012 kompilieren, werden SIMD Befehle für die FPU benutzt.

Listing 1.370: German text placeholder MSVC 2012

__real@3f800000 DD 03f800000r ; 1
tv128 = -4
_tmp$ = -4
?float_rand@YAMXZ PROC
push ecx
call ?my_rand@YAI3XZ
EAX=Pseudzufallswert
and eax, 8388607 ; 007fffffffH
or eax, 165353216 ; 3f800000H
EAX=Pseudzufallswert & 0x007fffffff | 0x3f800000
; speichere ihn auf lokalem Stack:
mov DWORD PTR _tmp$[esp+4], eax
; lade ihn erneut als Fließkommazahl:
movss xmm0, DWORD PTR _tmp$[esp+4]
; subtrahiere 1.0:
subss xmm0, DWORD PTR __real@3f800000
; verschiebe Wert nach ST0 durch Ablegen in temporärer Variable...
movss DWORD PTR tv128[esp+4], xmm0
; ... und lade ihn erneut nach ST0:
fld DWORD PTR tv128[esp+4]
pop ecx
ret 0
?float_rand@YAMXZ ENDP


„Scalar“ bedeutet, dass nur ein Wert im Register gespeichert ist. „Single“\(^\text{165}\) steht für den Datentyp float.

### Zusammenfassung

In den Beispielen hier wird nur die niedere Hälfte der XMM Register verwendet, um eine Zahl im IEEE 754 Format zu speichern.

Im Prinzip arbeiten alle Befehle, die um den Präfix -SD („Scalar Double-Precision“) ergänzt wurden, mit Fließkommazahlen im IEEE 754 Format, die in der niederen 64-Bit-Hälfte eines XMM Registers gespeichert werden.

\(^{164}\)Als Übung können Sie versuchen, den Code so umzugestalten, dass der lokale Stack nicht mehr verwendet wird

\(^{165}\)Single precision
Dies ist einfacher als in der FPU, da die SIMD sich in weniger chaotischer Weise als die FPU entwickelt hat. Das Stackregister Modell wird nicht verwendet.
Wenn man in diesen Beispielen double durch float ersetzen würde, würden die gleichen Befehle verwendet werden, aber jeweils mit -SS(„Scalar Single-Precision“) Präfix, z.B. MOVSS, COMISS, ADDSS, etc.
„Scalar“ bedeutet, dass die SIMD Register nur einen anstatt mehreren Werten enthalten.
Befehle, die mit mehreren Werten in einem Register gleichzeitig arbeiten, haben ein „Packed“ in ihrem Namen.
Unnötig extra zu erwähnen, dass die SSE2 Befehle mit 64-Bit IEEE 743 Werten (double) arbeiten, wohingegen die interne Repräsentation von Fließkommazahlen in der FPU 80-Bit-Zahlen verwendet.
Die FPU wird deshalb manchmal weniger Rundungsfehler machen und als Konsequenz daraus möglicherweise präzisere Berechnungsergebnisse liefern.

1.27 ARM-spezifische Details

1.27.1 Zeichen (#) vor einer Zahl
Der Keil-Compiler, IDA und objdump versehen alle Zahlen mit dem Präfix „#“, siehe hier: Listing.1.14.1. Wenn GCC 4.9 Output in Assemblersprache erzeugt, tut er dies jedoch ohne den Präfix: Listing.??.
Die ARM-Listings in diesem Buch sind gemischt.
Es ist schwer zu sagen, welche Methode die richtige ist. Am einfachsten ist es, die Regeln, die in der Umgebung, mit der man arbeitet, vorherrschen, zu akzeptieren.

1.27.2 Adressierungsmodi
Der folgende Befehl ist in ARM64 zulässig:

```
ldr     x0, [x29,24]
```

Das bedeutet, wir addieren 24 zum Wert in X29 und laden den Wert an dieser Adresse

Man beachte, dass die 24 sich innerhalb der Klammern befindet. Die Bedeutung ist eine andere, wenn die Zahl außerhalb der Klammer steht:

```
ldr     w4, [x1],28
```

Das bedeutet, wir laden den Wert an der Adresse in X1 und addieren dann 28 zu X1.

ARM erlaubt die Addition oder Subtraktion einer Konstanten zu bzw. von einer für einen Ladebefehl benötigten Adresse.
Es ist möglich dies vor und nach dem Laden zu tun.
Es gibt keinen derartigen Adressierungsmodus in x86, aber auch in anderen Prozessoren, sogar auf PDP-11.
Es gibt die Legende, dass die Pre-Inkrement-, Post-Inkrement-, Pre-Dekrement und Post-Dekrement-Modi in PDP-11 für das Erscheinen von C-Konstrukten (welche auf PDP-11 entwickelt wurden) wie *ptr++, *++ptr, *ptr--, *--ptr verantwortlich sind.
Dabei handelt es sich übrigens um ein schwer zu merkendes Feature von C. Es funktioniert wie folgt:
Pre-indexing wird in der ARM Assemblersprache mit einem Ausrufezeichen kenntlich gemacht. Zum Beispiel in der Zeile 2 in Listing 1.27.


Da durch können C-Compiler das Feature nutzen, wenn es auch auf dem Zielprozessor implementiert ist. Es ist sehr nützlich und gebräuchlich beim Verarbeiten von Arrays.

### 1.27.3 Laden einer Konstante in ein Register

#### 32-Bit ARM

Wie wir bereits wissen, haben alle Befehle im ARM mode eine Länge von 4 Byte bzw. 2 Byte im Thumb mode.

Wie können wir nun einen 32-Bit-Wert in ein Register laden, wenn es nicht möglich ist, ihn in einen Befehl hineinzukodieren?

Versuchen wir es:

```c
unsigned int f()
{
    return 0x12345678;
}
```

Listing 1.371: GCC 4.6.3 -O3

<table>
<thead>
<tr>
<th>C Term</th>
<th>ARM Term</th>
<th>C Ausdruck</th>
<th>so funktioniert er</th>
</tr>
</thead>
<tbody>
<tr>
<td>German text placeholder</td>
<td>post-indexed Adressierung</td>
<td>*ptr++</td>
<td>verwende *ptr Wert, dann inkrementiere ptr Pointer</td>
</tr>
<tr>
<td>German text placeholder</td>
<td>post-indexed Adressierung</td>
<td>*ptr--</td>
<td>verwende *ptr Wert, dann dekrementiere ptr Pointer</td>
</tr>
<tr>
<td>German text placeholder</td>
<td>pre-indexed Adressierung</td>
<td>++ptr</td>
<td>Inkrementiere ptr Pointer, dann verwende *ptr Wert</td>
</tr>
<tr>
<td>German text placeholder</td>
<td>pre-indexed Adressierung</td>
<td>--ptr</td>
<td>Dekrementiere ptr Pointer, dann verwende *ptr Wert</td>
</tr>
</tbody>
</table>

Der Wert 0x12345678 wird im Speicher geparkt und wenn nötig geladen. Es ist aber möglich, den zusätzlichen Speicherzugriff loszuwerden.

Listing 1.372: GCC 4.6.3 -O3 -march=armv7-a

Wir sehen, dass der Wert stückweise in das Register geladen wird: zuerst der niedere Teil (mit MOVW), dann der höhere (mit MOVT).

Das bedeutet, dass im ARM mode 2 Befehle nötig sind, um einen 32-Bit-Wert in ein Register zu laden.

166[http://yurichev.com/mirrors/C/c_dmr_postincrement.txt](http://yurichev.com/mirrors/C/c_dmr_postincrement.txt)
167[German text placeholder http://go.yurichev.com/17264](http://go.yurichev.com/17264)
Das stellt kein wirkliches Problem dar, denn es gibt in realen Code nicht allzu viele Konstanten (außer 0 und 1).
Hat dies auch zur Folge, dass die Version mit zwei Befehlen langsamer ist als die mit einem?
Eher nicht. Wahrscheinlicher ist, dass moderne ARM Prozessoren in der Lage sind, solche Sequenzen zu erkennen und schnell auszuführen.
Auch IDA vermag solche Muster im Code zu erkennen und disassembliert die Funktion wie folgt:

```
MOV R0, 0x12345678
BX LR
```

**ARM64**

```c
uint64_t f()
{
    return 0x12345678ABCDEF01;
}
```

Listing 1.373: GCC 4.9.1 -O3

```
mov x0, 61185 ; 0xef01
movk x0, 0xabcd, lsl 16
movk x0, 0x5678, lsl 32
movk x0, 0x1234, lsl 48
ret
```

MOVK steht für „MOV Keep“, d.h. der Befehl schreibt einen 16-Bit-Wert in das Register und lässt die übrigen Bits unverändert. Der Suffix LSL verschiebt den Wert um 16, 32 und 48 Bits in jedem Schritt nach links. Das Verschieben wird vor dem Laden durchgeführt.
Das bedeutet, dass 4 Befehle notwendig sind, um einen 64-Bit-Wert in ein Register zu laden.

**Fließkommazahl in einem Register speichern**
Es ist möglich mit nur einem Befehl eine Fließkommazahl in einem D-Register zu speichern.
Ein Beispiel:

```c
double a()
{
    return 1.5;
}
```

Listing 1.374: GCC 4.9.1 -O3 + objdump

```
0000000000000000 <a>:  
   0:  1ef10000          fmov d0, #1.500000000000000000e+000
   4:  d65f03c0          ret
```

Die Zahl 1.5 war tatsächlich in dem 32-Bit-Befehl kodiert. Aber wie ist das möglich? In ARM64 stehen 8 Bits im FMOV Befehl für das Kodieren von Fließkommazahlen zur Verfügung. Der Algorithmus heißt VFPExpandImm() in [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)]

Dieses Vorgehen wird auch *minifloat* genannt[169].
Wir können verschiedene Werte ausprobieren: der Compiler ist in der Lage 30.0 und 31.0 zu kodieren, aber nicht 32.0, da hier 8 Bytes für diese Zahl im IEEE 754 Format reserviert werden müssen.

[169]wikipedia
double a()
{
    return 32;
};

Listing 1.375: GCC 4.9.1 -O3

a:
    ldr    d0, .LC0
    ret
.LC0:
    .word  0
    .word  1077936128

1.27.4 Relocs in ARM64

Wie wir wissen, gibt es 4-Byte-Befehle in ARM64, sodass es unmöglich ist, eine große Zahl mit einem einzigen Befehl in ein Register zu schreiben.

Nichtsdestotrotz kann eine Executable an jeder Adresse des Speichers geladen werden und das ist der Grund dafür, dass Relocs existieren. Mehr dazu (in Bezug auf Win32 PE) unter: 6.5.2 on page 474.

Die Adresse wird in ARM64 aus einem ADRP und ADD Befehlspar paar zusammengesetzt. Der erste Befehl lädt eine 4KiB-Seitenadresse und der zweite den Rest. Kompilieren wir das Beispiel aus „Hallo, Welt!“ (Listing.1.7) in GCC (Linaro) 4.9 unter win32:

Listing 1.376: GCC (Linaro) 4.9 und objdump der Objekdatei

...>aarch64-linux-gnu-gcc.exe hw.c -c
...>aarch64-linux-gnu-objdump.exe -d hw.o
...

0000000000000008 <main>:
  0:  a9bf7bfd stp x29, x30, [sp,#-16]!
  4:  910003fd mov x29, sp
  8:  90000000 adrp x0, 0 <main>
 c:  91000000 add x0, x0, #0x0
10:  94000000 bl  0 <printf>
14:  52800000 mov w0, #0x0   // #0
18:  a8c17bfd ldp x29, x30, [sp],#16
1c:  d65f03c0 ret
...

RELOCATION RECORDS FOR [.text]:
OFFSET TYPE VALUE
0000000000000008 R_AARCH64_ADR_PREL_PG_HI21 .rodata
0000000000000000c R_AARCH64_ADD_ABS_L012_NC .rodata
0000000000000010 R_AARCH64_CALL26 _printf

Es gibt hier 3 Relocs in dieser Datei.

- Der erste nimmt die Seitenadresse, schneidet die niedersten 12 Bits aus und schreibt die übrigen 21 Bit in das Bitfield für den ADRP Befehl. Das wird getan, weil wir die niedersten 12 Bit nicht kodieren und der Befehl ADRP nur Platz für 21 Bit hat.
- Der zweite legt die 12 Bit der Adresse relativ zum Seitenanfang in das Bitfield des ADD Befehls.
- Der letzte mit 26 Bit wird auf den Befehl an der Adresse 0x10 angewendet, an der sich der Sprung zur Funktion printf() befindet.
Alle ARM64 (und ARM im ARM mode) Befehlsadressen haben Nullen in den zwei niederwertigsten Bits (da alle Befehle eine Größe von 4 Byte haben), sodass man nur die höchsten 26 Bit eines 28-Bit-Adressraums (±128MB) kodieren muss.

Solche Relocs gibt es in der ausführbaren Datei nicht: das liegt daran, dass bekannt ist, wo der „Hello!“ String abgelegt wurde; die Seiten und die Adresse von puts() sind ebenfalls bekannt.

In den Befehlen ADRP, ADD und BL sind also bereits Werte gesetzt (der Linker hat diese während des Linkens geschrieben):

Listing 1.377: objdump der ausführbaren Datei

<table>
<thead>
<tr>
<th>Adresse</th>
<th>Befehl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400590</td>
<td>stp x29, x30, [sp,-16]!</td>
</tr>
<tr>
<td>0x400594</td>
<td>mov x29, sp</td>
</tr>
<tr>
<td>0x400598</td>
<td>adrp x0, 400000 &lt; init-0x3b8&gt;</td>
</tr>
<tr>
<td>0x40059c</td>
<td>add x0, x0, #0x648</td>
</tr>
<tr>
<td>0x4005a0</td>
<td>bl 400420 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>0x4005a4</td>
<td>mov w0, #0 // #0</td>
</tr>
<tr>
<td>0x4005a8</td>
<td>ldp x29, x30, [sp],#16</td>
</tr>
<tr>
<td>0x4005ac</td>
<td>ret</td>
</tr>
</tbody>
</table>

Zur Veranschaulichung wollen wir den BL Befehl manuell disassemblieren. 0x97ffffa0 entspricht 0b100101111111111111111110100000. Gemäß [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)C5.6.26] sind imm26 die letzten 26 Bit: imm26 = 0b11111111111111111110100000. Das ist 0x3FFFFA0, aber da das MSB 1 ist, handelt es sich um eine negative Zahl und wir können diese manuell in eine handlichere Form umwandeln. Nach den Regeln für Negation (2.1 on page 409) invertieren wir alle Bits: (das ergibt 0b1011111=0x5F) und addieren dann 1 (0x5F+1=0x60). Die vorzeichenbehaftete Form ist also -0x60. Multiplizieren wir -0x60 mit 4 (da die im Opcode gespeicherte Adresse durch 4 geteilt worden ist): das ergibt -0x180. Berechnen wir nun die Zieladresse: 0x4005a0 + (-0x180) = 0x400420 (man beachte: wir betrachten die Adresse des BL-Befehls, nicht den aktuellen Wert von PC, der auch ein anderer sein könnte!). Die Zieladresse ist also 0x400420.

Für mehr Informationen zu Relocs in ARM64 siehe: [ELF für die ARM 64-Bit-Architektur (AArch64), (2013)]

1.28 MIPS-spezifische Details

1.28.1 Laden einer 32-Bit-Konstante in ein Register

```c
unsigned int f()
{
    return 0x12345678;
}
```

Alle Anweisungen in MIPS sind genau wie bei ARM 32-Bit groß. Somit ist es nicht möglich eine 32-Bit-Variable in einer Anweisung unterzubringen.

Dementsprechend müssen mindestens zwei Anweisungen genutzt werden: die erste lädt den höherwertigen Teil der 32-Bit-Zahl und die zweite führt eine ODER-Anweisung aus, welche den niederwertigen 16-Bit-Teil des Zielregisters setzt:ret register:

Listing 1.378: GCC 4.4.5 -O3 (German text placeholder)

```
li $2,385397760 # 0x12340000
j $31
ori $2,$2,0x5678 ; branch delay slot
```

170 [German text placeholder http://go.yurichev.com/17288](http://go.yurichev.com/17288)
IDA kennt dieses oft genutzte Muster sehr gut. Aus Gründen der Übersichtlichkeit wird hier die letzte ORI-Anweisung als LI-Pseudo-Anweisung angezeigt, welche vermeintlich eine komplette 32-Bit-Zahl in das $V0-Register lädt.

Die GCC-Assembler-Ausgabe beinhaltet ebenfalls die LI-Pseudo-Anweisung, allerdings ist hier auch die Anweisung LUI („Load Upper Immediate“), die einen 16-Bit-Wert im höherwertigen Teil des Registers sichert.

Nachfolgend die objdump-Ausgabe:

Laden einer globalen 32-Bit-Variable in ein Register

unsigned int global_var=0x12345678;

unsigned int f2()
{
    return global_var;
};

Dies ist leicht unterschiedlich: LUI lädt die oberen 16 Bit von global_var in $2 (or $V0). Anschließend lädt LW die unteren 16 Bit und addiert sie mit dem Inhalt von $2:

IDA kennt auch das oft genutzte Anweisungspaar LUI/LW und fasst beide in der einzelten Anweisung LW zusammen:

...
Die Ausgabe von objdump ist die selbe wie die Assembler-Ausgabe von GCC. Nachfolgend die Speicherauszüge der Objekdateien:

Listing 1.383: objdump

```
objdump -D filename.o

... 

0000000c <f2>:
  c:  3c020000  lui  v0,0x0
  10: 8c420000  lw  v0,0(v0)
  14: 03e00008  jr  ra
  18: 00200825  move  at,at  ; branch delay slot
  1c: 00200825  move  at,at

Disassembly of section .data:

00000000 <global_var>:
  0: 12345678  beq  s1,s4,159e4 <f2+0x159d8>

objdump -r filename.o

... 

RELOCATION RECORDS FOR [.text]:
OFFSET  TYPE      VALUE 
0000000c  R_MIPS_HI16  global_var
00000010  R_MIPS_LO16  global_var

... 

Es ist zu sehen, dass die Adresse von global_var direkt mit den Anweisungen LUI und LW beim Laden der ausführbaren Datei geschrieben wird: der höherwertige 16-Bit-Teil von global_var mit der ersten Anweisung (LUI), der niederwertige 16-Bit-Teil mit der zweiten Anweisung (LW).

1.28.2 Weitere Literatur über MIPS

Kapitel 2

Wichtige Grundlagen

2.1 German text placeholder

Es gibt verschiedene Arten vorzeichenbehaftete Zahlen darzustellen, jedoch ist das „Zweierkomplement“ die am weitesten verbreitete in Computern.

Hier ist eine Tabelle für einige Byte-Werte:

<table>
<thead>
<tr>
<th>binär</th>
<th>hexadezimal</th>
<th>vorzeichenlos</th>
<th>vorzeichenbehaftet</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111111</td>
<td>0x7f</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>01111110</td>
<td>0x7e</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000110</td>
<td>0x6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>00000101</td>
<td>0x5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>00000100</td>
<td>0x4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>00000011</td>
<td>0x3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>00000010</td>
<td>0x2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>00000001</td>
<td>0x1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>00000000</td>
<td>0x0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11111111</td>
<td>0xff</td>
<td>255</td>
<td>-1</td>
</tr>
<tr>
<td>11111110</td>
<td>0xfe</td>
<td>254</td>
<td>-2</td>
</tr>
<tr>
<td>11111101</td>
<td>0xfd</td>
<td>253</td>
<td>-3</td>
</tr>
<tr>
<td>11111100</td>
<td>0xfc</td>
<td>252</td>
<td>-4</td>
</tr>
<tr>
<td>11111011</td>
<td>0xfb</td>
<td>251</td>
<td>-5</td>
</tr>
<tr>
<td>11111010</td>
<td>0xfa</td>
<td>250</td>
<td>-6</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000010</td>
<td>0x82</td>
<td>130</td>
<td>-126</td>
</tr>
<tr>
<td>10000001</td>
<td>0x81</td>
<td>129</td>
<td>-127</td>
</tr>
<tr>
<td>10000000</td>
<td>0x80</td>
<td>128</td>
<td>-128</td>
</tr>
</tbody>
</table>

Der Unterschied zwischen vorzeichenbehafteten und vorzeichenlosen Zahlen ist, dass wenn 0xFFFFFE und 0x00000002 ohne Vorzeichen repräsentiert werden, die erste Zahl (4294967294) größer ist als die zweite Zahl (2). Wenn beide Zahlen als vorzeichenbehaftet repräsentiert werden, wird die erste Zahl –2 und ist kleiner als die zweite Zahl (2). Das ist der Grund, warum bedingte Sprünge (1.12 on page 105) sowohl für vorzeichenbehaftete (z.B. JG, JL) als auch vorzeichenlose (JA, JB) Operationen vorhanden sind.

Aus Gründen der Einfachheit ist hier was man wissen muss:

- Zahlen können mit oder ohne Vorzeichen sein.
- vorzeichenbehaftete Datentypen in C/C++:
  - int64_t (-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807) (- 9.2.. 9.2 Quintillionen) oder 0x8000000000000000 .. 0x7FFFFFFFFFFFFFFF,
  - int (-2,147,483,648..2,147,483,647 (- 2.15.. 2.15Gb) oder 0x80000000 .. 0x7FFFFFFF),
  - char (-128..127 oder 0x80 .. 0x7F),
  - ssize_t.
Vorzeichenlos:
- `uint64_t` (0..18,446,744,073,709,551,615 (18 Quintillionen) oder 0..0xFFFFFFFFFFFFFFFF),
- `unsigned int` (0..4,294,967,295 (4,3Gb) oder 0..0xFFFFFFFF),
- `unsigned char` (0..255 oder 0..0xFF),
- `size_t`.
- Vorzeichenbehaftete Typen haben das Vorzeichen am **MSB!** 1 bedeutet „Minus“, 0 bedeutet „Plus“.
- Erweitern auf größere Datentypen ist einfach: [1.23.5 on page 367](#).
- Negieren ist einfach: es müssen lediglich alle Bits invertiert und anschließend 1 addiert werden.
- Die Addition und Substraktion funktioniert sowohl für Zahlen mit als auch ohne Vorzeichen. Für Multiplikation und Division gibt es bei x86 unterschiedliche Anweisungen: `IDIV/IMUL` für vorzeichenbehaftete und `DJV/MUL` für vorzeichenlose Zahlen.
- Hier sind einige weitere Anweisungen die mit vorzeichenbehafteten Zahlen arbeiten: 
  - `CBW/CWD/CWDE/CDQ/CDQE` ([1.6 on page 536](#)),
  - `MOVsx` ([1.15.1 on page 179](#)), 
  - `SAR` ([1.6 on page 539](#)).


Die Addition zweier negativer Zahlen funktioniert wie folgt: Wenn beispielsweise `-2` und `-3` unter Verwendung eines 16-Bit-Registers addiert werden sollen, ist die Darstellung `0xfffe` beziehungsweise `0xfff0`. Wenn diese Werte ohne Vorzeichen addiert werden, ist das Ergebnis `0xffffffff`. Allerdings sollen 16-Bit-Register verwendet werden, also wird beim Ergebnis die erste 1 abgeschnitten. Es bleibt `0xfff` was `5` entspricht. Dies funktioniert, weil `-2` (oder `0xfff`) in natürlicher Sprache wie folgt repräsentiert werden kann: “2 fehlt bei diesem Werte bis zum maximalen Wert in einem 16-Bit-Register + 1”. `-3` kann repräsentiert werden als “…3 fehlt in diesem Wert bis zu …”. Der maximale Wert eines 16-Bit-Registers + 1 ist `0x10000`. Bei der Addition der beiden Zahlen und **Abschneiden** durch `2^10` modulo, wird `2 + 3 = 5` fehlen.

### 2.1.1 Nutzen von IMUL anstatt MUL

Beispiele wie Listing.?? in denen zwei vorzeichenlose Werte miteinander multipliziert werden, werden zu Listing.?? kompiliert, so dass IMUL statt MUL genutzt wird.

Dies ist eine wichtige Eigenschaft der MUL- und IMUL-Anweisung. Zunächst produzieren beide einen 64-Bit-Wert wenn zwei 32-Bit-Werte miteinander multipliziert werden, oder einen 128-Bit-Wert wenn zwei 64-Bit-Werte miteinander multipliziert werden (größtes mögliches Produkt in 32-Bit-Umgebungen ist `0xffffffff*f0xffffffff=0xfffffffff00000001`). Der C/++-Standard kennt keine Möglichkeit auf die höherwertige Hälfte eines Ergebnisses zuzugreifen und ein Produkt hat immer die gleiche Größe wie die Faktoren. Beide Anweisungen MUL und IMUL arbeiten auf die gleiche Weise wenn die höherwertige Hälfte ignoriert wird. Das heißt die niederwertigere Hälfte ist die gleiche. Dies ist eine wichtige Eigenschaft der Repräsentation von vorzeichenbehafteten Zahlen im „Zweierkomplements“.

Somit kann der C/++-Compiler jede dieser Anweisungen nutzen.


Aus diesen Gründen ist es möglich, dass ein C/++-Compiler öfter IMUL-Anweisungen als MUL nutzt.

Trotzdem ist es möglich mit instrinsischen Funktionen (Intrinsics) des Compilers vorzeichenlose Multiplikationen durchzuführen und das **volle** Ergebnis zu erhalten. Dies wird manchmal **erweiterte Multiplikation** genannt. MSVC hat Intrinsics zu diesem Zweck die `_emul2` und `_umul128` genannt werden. GCC stellt einen

2. [https://msdn.microsoft.com/en-us/library/d2s81xt0(v=vs.80).aspx](https://msdn.microsoft.com/en-us/library/d2s81xt0(v=vs.80).aspx)
3. [https://msdn.microsoft.com/library/3dayytw9%28v=vs.100%29.aspx](https://msdn.microsoft.com/library/3dayytw9%28v=vs.100%29.aspx)
__int128-Datentyp zur Verfügung und 64-Bit-Faktoren werden zuerst auf 128-Bit erweitert, Anschließend wird das Produkt in einem anderen __int128 gesichert. Das Ergebnis ist um 64-Bit nach rechts geshiftet um die höherwertigen Hälfte des Ergebnisses zu erhalten⁴.

**MulDiv()-Funktion in Windows**

Windows hat eine MulDiv()-Funktion ⁵, welche die Multiplikation und Division vereint und zwei 32-Bit-Integer in einen temporären 64-Bit-Wert speichert. Anschließend findet eine Division durch eine dritte 32-Bit-Integerzahl statt. Dies ist einfacher als zwei Compiler-Intrinsics zu nutzen, weswegen die Microsoft-Entwickler diese spezielle Funktion dafür einführten. Gemessen an der Häufigkeit der Nutzung ist dies eine populäre Funktion.

### 2.1.2 Einige weitere Anmerkungen zum Zweierkomplement

#### Das Maximum eines German text placeholder finden

Die maximale Zahl in vorzeichenloser Form ist lediglich eine Zahl in der alle Bits gesetzt sind: 0xFF....FF (das entspricht -1 wenn das German text placeholder als vorzeichenbehaftete Zahl behandelt wird). Also nimmt man ein German text placeholder, setzt alle Bits und bekommt den Wert:

```c
#include <stdio.h>

int main()
{
    unsigned int val=~0; // Aendern zu "unsigned char" um den maximalen Wert fur vorzeichenlose 8-Bit-Zahlen
    // 0-1 funktionieren auch oder nur -1
    printf("%u\n", val); //
}
```

Dies entspricht 4294967295 für 32-Bit-Integer.

#### Das Minimum eines vorzeichenbehafteten German text placeholder finden

```c
#include <stdio.h>

int main()
{
    signed int val=1; // Aendern zu signed char um den maximalen Wert fur vorzeichenbehaftete 8-Bit-Zahlen
    while (val!=0)
    {
        printf("%d %d\n", val, ~val);
        val=val<<1;
    }
}
```

Die Ausgabe ist:

```
...  
536870912 -536870913
```

⁴Example: http://stackoverflow.com/a/13187798
Die letzten beiden Zahlen sind minimaler beziehungsweise maximaler Wert eines 32-Bit *int*. 

<table>
<thead>
<tr>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1073741824</td>
<td>1073741825</td>
</tr>
<tr>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
</tbody>
</table>
Kapitel 3

Fortgeschrittenere Beispiele

3.1 strstr()-Beispiel

Erinnern wir uns an die Tatsache, dass GCC manchmal Teile einer Zeichenkette nutzen kann: ?? on page ??.

Die `strstr()` C/C++-Standard-Bibliotheksfunction wird genutzt um das Auftreten einer Zeichenkette in einer anderen zu finden. Nachfolgend ein Beispiel für die Anwendung:

```c
#include <string.h>
#include <stdio.h>

int main()
{
    char *s="Hello, world!";
    char *w=strstr(s, "world");

    printf ("%p, [%s]\n", s, s);
    printf ("%p, [%s]\n", w, w);
}
```

Die Ausgabe ist:

```
0x8048530, [Hello, world!]
0x8048537, [world!]
```


Der zweite `printf()`-Aufruf weiß nicht, dass weitere Zeichen vor der Zeichenkette sind und gibt lediglich die Zeichen von der Mitte der Original-Zeichenkette bis zum Ende aus (markiert durch ein Null-Byte).
Kapitel 4

Java

4.1 Java

4.1.1 Einführung

Es gibt einige bekannte Decompiler für Java (oder JVM-Bytecode allgemein). Der Grund ist, dass das Dekompilieren von JVM-Bytecode einfacher ist als von Low-Level x86-Code:

- Es gibt sehr viel mehr Informationen über die Datentypen.
- Das JVM-Speichermodell ist sehr viel strenger und genauer beschrieben.
- Der Java-Compiler führt keine Optimierungen durch (dies macht der JVM JIT (Just-In-Time compilation) während der Laufzeit, so dass der Bytecode in der Klassendatei normalerweise gut lesbar ist).

Wann kann das Wissen über JVM nützlich sein?

- Quick-and-dirty-Patchen von Klassendateien ohne das Neukompilieren der Decompiler-Ergebnisse.
- Analysieren von obfuskiertem Code.
- Erstellen eines eigenen Obfuscators.
- Erstellen eines Compiler Code-Generators (back-end) mit dem Ziel JVM (wie Scala, Clojure, usw.).

Starten wir mit einigen einfachen Code-Beispielen. Wenn nicht anders erwähnt wird JDK 1.7 verwendet. Das folgende Kommando wird verwendet um Klassen zu decompilieren:

```bash
javap -c -verbose
```


4.1.2 Rückgabe eines Wertes

Die vermutlich einfachste Java-Funktion ist eine, die einen Wert zurück gibt.

Es sollte hier noch beachtet werden, dass es keine „freien“ Funktionen im allgemeinen Sinne in Java gibt sondern „Methoden“.

Jede Methode gehört zu einer Klasse, somit ist es nicht möglich eine Methode außerhalb einer Klasse zu definieren.

Wir werden die Methoden hier trotzdem der Einfachheit halber „Funktionen“ nennen.

```java
public class ret {
   public static int main(String[] args)
```

---

1 Beispielsweise JAD: [http://varaneckas.com/jad/](http://varaneckas.com/jad/)
2 Just-In-Time compilation
4 German text placeholder [https://docs.oracle.com/javase/specs/jvm/se7/jvms7.pdf](https://docs.oracle.com/javase/specs/jvm/se7/jvms7.pdf); [http://docs.oracle.com/javase/specs/jvms/se7/html/](http://docs.oracle.com/javase/specs/jvms/se7/html/)
Kompilieren wir diesen Code:

```java
{    return 0;
}
```

...und dekompilieren ihn mit dem Standard-Java-Tool:

```bash
javap -c -verbose ret.class
```

Und wir bekommen:

Listing 4.1: JDK 1.7 (excerpt)

```java
public static int main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
  0: iconst_0
  1: ireturn
```

Die Java-Entwickler entschieden, dass 0 eine der beliebtesten Konstanten in der Programmierung ist, also gibt es eine separate, kurze Ein-Byte-Anweisung die 0 pushed.

4.1.3 Einfache Berechnungsfunktionen

4.1.4 JVM-Speichermodell

4.1.5 Einfache Funktionsaufrufe

4.1.6 Aufrufen von beep()

Dies ist ein einfacher Aufruf zweier Funktionen ohne Argumente:

```java
public static void main(String[] args)
{
    java.awt.Toolkit.getDefaultToolkit().beep();
};
```

```java
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
  0: invokestatic #2 // Method java.awt.Toolkit.getDefaultToolkit():Ljava/awt/Toolkit;
  3: invokevirtual #3 // Method java.awt.Toolkit.beep():V
  6: return
```

Zuerst ruft invokevirtual bei Offset 0 java.awt.Toolkit.getDefaultToolkit() auf, was eine Referenz auf ein Objekt der Klasse Toolkit zurück gibt. Die invokevirtual-Anweisung bei Offset 3 ruft die beep()-Methode dieser Klasse auf.
4.1.7 Linearer Kongruenzgenerator **PRNG**
4.1.8 Bedingte Sprünge
4.1.9 Argumente übergeben
4.1.10 Bit-Felder
4.1.11 Schleifen
4.1.12 switch()
4.1.13 Arrays
4.1.14 Zeichenketten
4.1.15 Klassen

Einfache Klassen:

Listing 4.2: test.java

```java
public class test {
    public static int a;
    private static int b;
    
    public test() {
        a=0;
        b=0;
    }
    public static void set_a (int input) {
        a=input;
    }
    public static int get_a () {
        return a;
    }
    public static void set_b (int input) {
        b=input;
    }
    public static int get_b () {
        return b;
    }
}
```

Der Konstruktor setzt lediglich beide Felder auf 0:

```
public test();
flags: ACC_PUBLIC
Code:
    stack=1, locals=1, args_size=1
    0: aload_0
    1: invokespecial #1 // Method java/lang/Object."<init>":()V
    4: iconst_0
    5: putstatic    #2 // Feld a:I
    8: iconst_0
    9: putstatic    #3 // Feld b:I
   12: return
```

Setter von `a`:
public static void set_a(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iload_0
1: putstatic #2 // Feld a:I
4: return

Getter von a:

public static int get_a();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=0, args_size=0
0: getstatic #2 // Feld a:I
3: ireturn

Setter von b:

public static void set_b(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iload_0
1: putstatic #3 // Feld b:I
4: return

Getter von b:

public static int get_b();
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=0, args_size=0
0: getstatic #3 // Feld b:I
3: ireturn

Es gibt keinen Unterschied in den Codes die mit public oder private Feldern arbeiten.
Aber diese Information ist in der .class-Datei vorhanden und es ist nicht möglich auf die privaten Felder zuzugreifen.
Erstellen wir ein Objekt und rufen seine Methoden auf:

Listing 4.3: ex1.java

```java
class ex1 {
    public static void main(String[] args) {
        test obj=new test();
        obj.set_a (1234);
        System.out.println(obj.a);
    }
}
```
Die new-Anweisung erstellt ein Objekt aber ruft den Konstruktor nicht auf (dieser wird bei Offset 4 aufgerufen). Die set_a()-Methode wird an Offset 16 aufgerufen.
Auf das Feld a wird mit der getstatic-Anweisung an Offset 21 zugegriffen.

4.1.16 Einfaches Patchen

4.1.17 Zusammenfassung

Was fehlt in Java im Vergleich zu C/C++?

- Strukturen: Nutzen Sie Klassen.
- Unions: Nutzen Sie Klassenhierarchien.
- Vorzeichenlose Datentypen. Dies macht es etwas schwieriger kryptografische Algorithmen in Java zu implementieren.
- Funktionszeiger.
Finden von wichtigen / interessanten Stellen im Code

Minimalismus ist kein beliebtes Feature moderner Software.

Aber nicht weil die Programmierer so viel Code schreiben, sondern weil die Libraries allgemein statisch zu ausführbaren Dateien gelinkt werden. Wenn alle externen Libraries in externe DLL Dateien verschoben werden würden, wäre die Welt ein anderer Ort. (Ein weiterer Grund für C++ sind die STL\(^1\) und andere Template-Libraries.)

Deshalb ist es sehr wichtig den Ursprung einer Funktion zu bestimmen, wenn die Funktion aus einer Standard-Library oder aus einer sehr bekannten Library stammt (wie z.B Boost\(^2\), libpng\(^3\)), oder ob die Funktion sich auf das bezieht was wir im Code versuchen zu finden.

Es ist ein wenig absurd sämtlichen Code in C/C++ neu zu schreiben, um das zu finden was wir suchen.

Eine der Hauptaufgaben eines Reverse Enigneers ist es schnell Code zu finden den er/sie sucht.

Der \texttt{IDA}\(^4\)-Disassembler erlaubt es durch Textstrings, Byte-Sequenzen und Konstanten zu suchen. Es ist sogar möglich den Code in .lst oder .asm Text Dateien zu exportieren und diese mit grep, awk, etc. zu untersuchen.

Wenn man versucht zu verstehen wie ein bestimmter Code funktioniert, kann auch eine einfache Open-Source-Library wie libpng als Beispiel dienen. Wenn man also eine Konstante oder Textstrings findet die vertraut erscheinen, ist es immer einen Versuch wert diese zu google\(\)en . Und wenn man ein Opensource Projekt findet in dem diese Funktion benutzt wird, reicht es meist aus diese Funktionen miteinander zu vergleichen. Es könnte helfen Teile des Problems zu lösen.

Zum Beispiel, wenn ein Programm XML Dateien benutzt, wäre der erste Schritt zu ermitteln welche XML-Library benutzt wird für die Verarbeitung, da die Standard (oder am weitesten verbreitete) libraries normal benutzt werden anstatt selbst geschriebene libraries.

Zum Beispiel, der Autor dieser Zeilen wollte verstehen wie die Kompression/Dekompression von Netzwerkpaketen in SAP 6.0 funktioniert. SAP ist ein gewaltiges Stück Software, aber detaillierte \texttt{German text placeholder} Dateien mit Debug Informationen sind vorhanden, was sehr praktisch ist. Der Autor hat schließlich eine Ahnung gehabt, das eine Funktion genannt \texttt{CsDecomprLZC} die Dekompression der Netzwerkpakete übernahm. Er hat nach dem Namen der Funktion auf google gesucht und ist schnell zum schluss gekommen das diese Funktion in MaxDB benutzt wurde (Das ist ein Open-Source SAP Projekt) \(^4\).

\texttt{http://www.google.com/search?q=CsDecomprLZC}

Erstaunlich, das MaxDB und die SAP 6.0 Software den selben Code geteilt haben für die Kompression/Dekompression der Netzwerkpakete.

\(^1\)(C++) Standard Template Library \(^2\)http://go.yurichev.com/17036 \(^3\)http://go.yurichev.com/17037 \(^4\)Mehr darüber in der relevanten Sektion ( ?? on page ??)
5.1 Ausführbare Dateien Identifizieren

5.1.1 Microsoft Visual C++
MSVC Versionen und DLLs die Importiert werden können:

<table>
<thead>
<tr>
<th>Marketing ver.</th>
<th>Internal ver.</th>
<th>CL.EXE ver.</th>
<th>DLLs imported</th>
<th>Release date</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6.0</td>
<td>12.00</td>
<td>msvcr.dll</td>
<td>June 1998</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>msvcp60.dll</td>
<td></td>
</tr>
<tr>
<td>.NET (2002)</td>
<td>7.0</td>
<td>13.00</td>
<td>msvcr70.dll</td>
<td>February 13, 2002</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>msvcp70.dll</td>
<td></td>
</tr>
<tr>
<td>.NET 2003</td>
<td>7.1</td>
<td>13.10</td>
<td>msvcr71.dll</td>
<td>April 24, 2003</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>msvcp71.dll</td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>8.0</td>
<td>14.00</td>
<td>msvcr80.dll</td>
<td>November 7, 2005</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>msvcp80.dll</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>9.0</td>
<td>15.00</td>
<td>msvcr90.dll</td>
<td>November 19, 2007</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>msvcp90.dll</td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td>10.0</td>
<td>16.00</td>
<td>msvcr100.dll</td>
<td>April 12, 2010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>msvcp100.dll</td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td>11.0</td>
<td>17.00</td>
<td>msvcr110.dll</td>
<td>September 12, 2012</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>msvcp110.dll</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>12.0</td>
<td>18.00</td>
<td>msvcr120.dll</td>
<td>October 17, 2013</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>msvcp120.dll</td>
<td></td>
</tr>
</tbody>
</table>

msvcp*.dll hat C++-bezogene Funktionen, bedeutet wenn die library importiert wird, ist das Programm das sie importiert wahrscheinlich ein C++ program.

Name mangling
Die Namen fangen normal an mit dem ? Symbol.
Hier: ?? on page ?? kann man mehr lesen über MSVC’s German text placeholder.

5.1.2 GCC
Neben *NIX Umgebungen, ist GCC auch in win32 Umgebungen präsent, in der Form von Cygwin and MinGW.

Name mangling
Namen fangen hier normal mit dem _Z Symbolen an.
Man kann mehr lesen über GCC’s German text placeholder hier: ?? on page ??.

Cygwin
cygwin1.dll wird oft importiert.

MinGW
msvcr.dll wird vielleicht importiert.

5.1.3 Intel Fortran
libifcoremd.dll, libifportmd.dll and libiomp5md.dll (OpenMP Support) werden vielleicht importiert.
libifcoremd.dll hat eine menge an Funktionen die das for_ Präfix haben, was Fortran bedeutet.

5.1.4 Watcom, OpenWatcom
Name mangling
Namen fangen normal mit dem W Symbol an.
Zum Beispiel wird so eine Methode benannt „method“ der Klasse „class“ die keine Argumente hat und void zurück gibt:
5.1.5 Borland
Hier ist ein Beispiel für Borland Delphi’s und C++Builder’s German text placeholder:

Die Namen fangen immer mit dem @ Symbol an, dann haben wir den Namen der Klassen Namen, Methoden Namen, und codiert die Typen der Argumente der Methode.
Diese Namen können in den .exe Imports, .dll Exports, Debug Daten und etc existieren.
Borland Visual Component Libraries (VCL) werden in .bpl Dateien gehalten anstatt .dll’s, zum Beispiel vcl50.dll, rtl60.dll.
Eine weitere DLL die vielleicht importiert wird: BORLNDMM.DLL

Delphi
Dies ist ein sehr typischer Anfang für das CODE Segment bei einem Delphi Programm, dieser Block kam direkt nach dem win32 PE Datei header:

```
00000400 04 10 40 00 03 07 42 6f 6f 6c 65 61 6e 01 00 00 |
00000410 00 00 01 00 00 00 00 10 00 05 46 61 6c 65 06 61 |Boolean...
00000420 00 00 58 69 64 65 53 74 72 69 6e 67 30 11 40 00 |..False
00000430 64 65 43 68 61 72 03 00 00 00 ff ff ff 00 00 00 |deChar.......
00000440 44 10 40 00 02 04 43 68 61 72 01 00 00 00 00 ff |D.@...Char.....
00000450 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |Int.........
00000460 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |...@...Wo
00000470 72 64 03 00 00 00 00 ff ff ff 00 00 00 90 10 40 00 |rd...........
00000480 ff 00 00 ff 00 00 00 00 9c 10 40 00 01 04 57 6f |..OleVariant...
00000490 00 00 00 00 9c 10 40 00 01 04 57 6f 72 64 03 00 |..OleVariant...
000004a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................
000004b0 28 4d 40 00 2c 4d 40 00 07 54 4f 62 6a 65 63 74 |(M@.,M@. TObject.
000004c0 a4 11 40 00 04 06 46 6f 6f 69 70 6c 65 05 00 |..Double@....
000004d0 04 11 40 00 04 08 07 7f 52 65 66 65 6c 65 03 00 |..Currency...
000004e0 14 11 40 00 04 08 43 75 72 72 65 6e 63 79 04 90 |..IInterface...
000004f0 f4 10 40 00 04 06 44 64 64 64 64 64 64 64 64 90 |..Extended...
00000500 04 11 40 00 04 08 43 75 72 72 65 6e 63 79 04 90 |..Currency...
00000510 1e 07 56 61 72 69 61 6e 74 8d 40 00 40 00 00 00 |..Variant.@@.
00000520 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
00000530 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
00000540 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
00000550 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
00000560 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
00000570 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
00000580 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
00000590 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
000005a0 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
000005b0 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
000005c0 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
000005d0 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
000005e0 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
000005f0 0c 07 56 61 72 69 61 6e 74 8d 40 00 00 00 00 00 |...Object...
```
Die ersten 4 Bytes des Daten Segments (DATA) können 00 00 00 00, 32 13 8B C0 oder FF FF FF FF sein.
Diese Informationen können nützlich sein wenn man mit gepackten oder verschlüsselten Delphi executables arbeiten muss.

5.1.6 Other known DLLs
- vcomp*.dll—Microsoft’s Implementierung von OpenMP.

5.2 Kommunikation mit der außen Welt (Funktion Level)
Oft ist es empfehlenswert die Funktionsargumente und die Rückgabewerte im Debugger oder DBI zu überwachen. Zum Beispiel hat der Autor einmal versucht die Bedeutung einer obskuren Funktion zu verstehen, die einen inkorrekten Bubblesort-Algorithmus implementiert hatte (Sie hat funktioniert, jedoch viel langsamer als normal). Die Eingaben und Ausgaben zur Laufzeit der Funktion zu überwachen hilft sofort zu verstehen was die Funktion tut.

5.3 Kommunikation mit der Außen Welt (Win32)
Manchmal reicht es die Ein- und Ausgaben einer Funktion zu beobachten um zu verstehen was sie tut. Auf diese Weise kann man Zeit Sparren.
Bei einfachen Netzwerk Zugriffen ist Wireshark zum analysieren ganz nützlich.
Trotzdem muss man einen Blick in die Netzwerkpakete werfen. Das erste wonach man schauen kann ist welche Funktionen die BS API’s benutzen und was für Standard libraries benutzt werden.
Wenn das Programm unterteilt ist in eine Main executable und mehrere DLL Dateien, können manchmal die Namen der Funktionen innerhalb der DLLs Helfen.
Wenn wir daran interessiert sind was genau zum Aufruf von MessageBox() mit einem spezifischen Text führt, können wir versuchen diesen Text innerhalb des Data Segments zu finden, die Referenzen auf den Text und die Punkte von denen aus die Kontrolle an den MessageBox() Aufruf an dem wir interessiert sind.
Wenn wir über Video Spiele sprechen sind wir daran interessiert welche rand() aufrufe mehr oder weniger zufällig darin vorkommen, vielleicht versuchen wir die rand() Funktion oder Ersatz Funktionen zu finden.

---

3Dynamic Binary Instrumentation
6https://yurichev.com/blog/weird_sort_KLEE/
7http://go.yurichev.com/17301
8http://go.yurichev.com/17303
9Application Programming Interface
(wie z.B. der Mersenne Twister Algorithmus) und wir versuchen die Orte zu finden von welchen aus diese Funktionen aufgerufen werden, und noch wichtiger was für Ergebnisse verwertet werden. Ein Beispiel: ?? on page ??.

Aber wenn es sich nicht um ein Spiel handelt und \( \text{rand()} \) wird trotzdem benutzt, ist es interessant zu wissen warum. Es gibt Fälle bei denen unerwartet \( \text{rand()} \) in Daten Kompressions Algorithmen benutzt wird (für die Imitation von Verschlüsslung): blog.yurichev.com.

### 5.3.1 Oft benutzte Funktionen in der Windows API

Diese Funktionen sind vielleicht unter den importierten. Es ist Sinnvoll an dieser Stelle zu erwähnen das nicht unbedingt jede Funktion benutzt wird aus dem Code den der Programmierer geschrieben hat.

Manche Funktionen haben eventuell das -a Suffix für die ASCII Version und das -w für die Unicode Version.

- Registry zugriff (advapi32.dll): RegEnumKeyEx, RegEnumValue, RegGetValue, RegOpenKeyEx, RegQueryValueEx
- Dialog boxes (user32.dll): MessageBox, MessageBoxEx, CreateDialog, SetDlgItemText, GetDlgItemText.
- Resourcen zugriff (6.5.2 on page 478): (user32.dll): LoadMenu.
- TCP/IP networking (ws2_32.dll): WSARecv, WSASend.
- Datei Zugriff (kernel32.dll): CreateFile, ReadFile, ReadFileEx, WriteFile, WriteFileEx.
- High-level Zugriff auf das Internet (wininet.dll): WinHttpOpen.
- Die Standard MSVC library (wenn sie dynamisch gelinket wurde)
  - assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strchr.

### 5.3.2 Verlängerung der Testphase

Registry Zugriffs Funktionen sind häufig ziele für Leute die versuchen die Testphase einer Software zu cracken, die eventuell die Installations Zeit und Datum in der Registery zu speichert.

Ein weiteres beliebtes Ziel sind die GetLocalTime() und GetSystemTime() Funktionen: eine Test Software, muss bei jedem Start die aktuelle Zeit und Datum überprüfen.

### 5.3.3 Entfernen nerviger DialogBoxen

Ein verbreiteter Weg raus zu finden was eine dieser nervigen Dialog boxen macht, ist den Aufruf von MessageBox(), CreateDialog() und CreateWindow() Funktionen abzufangen.

### 5.3.4 tracer: Alle Funktionen innerhalb eines bestimmten Modules abfangen

Es gibt INT3 breakpoints in German text placeholder, die nur einmal ausgelöst werden. Jedoch können diese breakpoints für alle Funktionen in einer bestimmten DLL gesetzt werden.

```bash
--one-time-INT3-bp:somedll.dll!*.
```

Oder, lasst uns einfach mal INT3 breakpoints für alle Funktionen setzen, die das xml Präfix in ihrem Namen haben:

```bash
--one-time-INT3-bp:somedll.dll!xml.*
```

Die andere Seite der Medaille ist, solche breakpoints werden nur einmal ausgelöst. Tracer zeigt den Aufruf der Funktion, wenn er passiert, aber auch nur einmal. Ein weiterer Nachteil ist—es ist unmöglich die Argumente der Funktion zu betrachten.
Dennoch, dieses Feature ist sehr nützlich wenn man weiß das das Programm eine DLL benutzt, aber man
nicht weiß welche Funktionen aufgerufen werden. Und es gibt eine ganze Menge an Funktionen.
Zum Beispiel, schauen wir uns einmal an was das uptime Kommando aus cygwin benutzt:

tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*

Dadurch sehen wir alle cygwin1.dll library Funktionen die zumindest einmal aufgerufen wurden, und von
welcher Stelle:

| One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime.exe!OEP+0x6d (0x40106d)) |
| One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from uptime.exe!OEP+0xba3 (0x401ba3)) |
| One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from uptime.exe!OEP+0xcb7 (0x401cb7)) |
| One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from uptime.exe!OEP+0xcbe (0x401cb7)) |
| One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptime.exe!OEP+0x735 (0x401735)) |
| One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptime.exe!OEP+0x994 (0x401994)) |
| One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from uptime.exe!OEP+0x7ea (0x4017ea)) |
| One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!OEP+0x809 (0x401809)) |
| One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from uptime.exe!OEP+0x994 (0x401994)) |
| One-time INT3 breakpoint: cygwin1.dll!setutent (called from uptime.exe!OEP+0x3b1 (0x4013b1)) |
| One-time INT3 breakpoint: cygwin1.dll!getutent (called from uptime.exe!OEP+0x3c5 (0x4013c5)) |
| One-time INT3 breakpoint: cygwin1.dll!endutent (called from uptime.exe!OEP+0x3e6 (0x4013e6)) |
| One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.exe!OEP+0x4c3 (0x4014c3)) |

5.4 Strings
5.4.1 Textstrings
C/C++
Die normalen C-strings sind NULL-Terminiert (ASCIIZ-strings).
Der Grund warum C Stringformatierung so ist wie sie ist (NULL-Terminiert) scheint ein Historischer zu sein. 
In [Dennis M. Ritchie, The Evolution of the Unix Time-sharing System, (1979)] kann man nach lesen:

Ein kleiner Unterschied war das die I/O Einheit ein “word” war, nicht ein Byte, weil die
PDP-7 eine word-adressierte Maschine war. In der Praxis bedeutete das lediglich das alle
Programme die mit Zeichen Streams arbeiteten, das NULL Zeichen ignorieren mussten, weil
die NULL benutzt wurde um eine Datei bis zu einer Graden Zahl an Bytes auf zu füllen.

In Hiew oder FAR Manager sehen diese Strings so aus:

```c
int main()
{
    printf ("Hello, world!\n");
}
```
Abbildung 5.1: Hiew

Borland Delphi

Dem String in Passcal und Borland Delphi hängt eine 8 oder 32-Bit Zeichenkette an.
Zum Beispiel:

Listing 5.1: Delphi

```
CODE:00518AC8       dd 19h
CODE:00518ACC       aLoading___Plea  db 'Loading... , please wait.',0
...
CODE:00518AFC       dd 10h
CODE:00518B00       aPreparingRun___ db 'Preparing run...',0
```

Unicode

Oft, ist das was Unicode genannt wird einfach eine Methode um Strings zu codieren, bei denen jedes Zeichen 2 Byte oder 16 Bits verbraucht. Das ist ein häufiger Terminologischer Fehler. Unicode ist ein Standard bei dem eine Nummer zu einem der vielen Schreibsysteme der Welt zugeordnet wird, aber es beschreibt nicht die codierungs Methode.

Die bekannteste Methode zu Codieren ist: UTF-8 ( ist weit verteilt im Internet und auf *NIX Systemen) und UTF-16LE ( wird bei Windows benutzt).

UTF-8

UTF-8 ist eine der erfolgreichsten Methoden um Zeichen zu codieren. Alle Latein Zeichen werden codiert so wie in ASCII, und alle Symbole nach der ASCII Tabelle wurden codiert mit zusätzlichen Bytes. 0 wird codiert als davor, also arbeiten alle Standard C String Funktionen mit UTF-8 Strings wie mit jedem anderen String auch.

Lasst uns anschauen wie die Symbole in verschiedenen anderen Sprachen nach UTF-8 Codiert werden und wie man sie als FAR aussehen lassen kann, durch das benutzen der codepage 437.

\[10\]

\[10\]Beispiel und Übersetzung können von hier bezogen werden: \texttt{http://go.yurichev.com/17304}

UTF-16LE

Viele win32 Funktionen in Windows haben die Suffixe -A und -W. Der erste Typ Funktionen arbeitet mit normalen Strings, der andere Typ mit UTF-16LE Strings (wide). Im zweiten Fall, wird jedes Symbol normal als 16-Bit Wert des Typs short gespeichert. Die Latein Symbole in UFT-16 Strings sehen in Hiew oder FAR aus als wären sie mit Null Bytes verschachtelt:

```
int wmain()
{
    wprintf (L"Hello, world!\n");
}
```

Abbildung 5.2: FAR: UTF-8

How much? 100€?

(English) I can eat glass and it doesn’t hurt me.
(Greek) Μπορώ να φάω πλαστικά γυαλιά χωρίς να πέσω τις τους.
(Hungarian) Meg tudom enni az üveget, nem lesz tôle bajom.
(Icelandic) Eg get etið glér án þess að meiða mig.
(Polish) Mogę jeść szkło i mi nie szkodzi.
(Russian) Я могу есть стекло, смотри мне не вредит.
(Arabic): أنا قادر على أكل الزجاج وهذا لا يجعلني
(Hebrew): אני יכול לאכול זכוכית ואני לא נפגע.
(Chinese) 我可以吃玻璃也不受伤。
(Japanese) 私はガラスを食べられます。それは私を傷つけません。
(Hindi) मैं नौली खाता हूँ और मुझे उससे कोई चोट नहीं पड़ती।
Wir können das oft auch in glsWindows NT System Dateien sehen:

Strings mit Zeichen die exakt 2 Bytes verbrauchen werden „Unicode“ in IDA genannt:

```
.data:0040E000 aHelloWorld:
data:0040E000 unicode 0, <Hello, world!>
data:0040E000 dw 0Ah, 0
```

Hier sieht man wie Russische Sprache in UTF-16LE Codiert wird:

Was man leicht sehen kann ist das die Symbole durchzogen sind von den Diamant Zeichen (das im ASCII code mit 4 codiert wird). Tatsächlich, findet man die Kyrillischen Symbole in der vierten Unicode Tabelle. Deswegen, alle Kyrillischen Symbole in UTF-16LE findet man im Bereich 0x400-0x4FF.

Hier können wir auch das BOM am Anfang sehen. Alle Latein Zeichen enthalten Null Bytes. Manche Zeichen mit unterschiedlichen Markierungen (Ungarisch und Isländisch) wurden rot unterstrichen.

**Base64**

Die Base64 Codierung ist sehr weit verbreitet für Fälle, in denen man Binärdaten als Textstring übertragen will.

Im Grunde, codiert dieser Algorithmus 3 Binär Bytes in 4 druckbare Zeichen: Alle 26 Latein Zeichen (beides klein und groß Buchstaben), Ziffern, plus Zeichen ("+"), und Slash Zeichen (/), insgesamt 64 Zeichen.

Ein charakteristisches Feature von Base64 Strings ist das sie oft (aber nicht immer) mit 1 oder 2 Gleichheitszeichen ("=") Enden, zum Beispiel:

AVjbbVSfCUMu1xvjaMgjNtueRwBxnyJw8dpGnLW8aKG3v4Y8icuQT+qEJAp9lA0uWs=

Das Gleichheitszeichen Symbol (q=) wird man niemals in der Mitte eines Base64-codierten Strings sehen.

Jetzt ein Beispiel wie man per Hand Base64 codieren kann. Lasst uns 0x00, 0x11, 0x22 und 0x33 in Hexadezimalzahlen in einen Base64 String umwandeln:

$ echo -n "\x00\x11\x22\x33" | base64
ABEiMw==

Lasst uns alle 4 Bytes in Binär Form bringen und dann neu gruppieren in 6-Bit Gruppen:

<table>
<thead>
<tr>
<th>00</th>
<th>11</th>
<th>22</th>
<th>33</th>
</tr>
</thead>
</table>
| 0000000000100010001000110001110011????????????????
| A | B | E | i | M | w | = | = |
Die ersten drei Bytes (0x00, 0x11, 0x22) können in 4 Base64 Zeichen umgewandelt werden ("ABEi"), aber nicht das letzte Byte (0x33), also wird das Byte codiert indem man zwei Buchstaben benutzt ("Mw") und das German text placeholder Symbol ("=") wird zweimal hinzugefügt um die letzte Gruppe auf 4 Zeichen zu erweitern. Das bedeutet das die Länge aller korrekten Base64 Strings sich immer durch 4 Teilen lässt.

Base64 wird oft benutzt wenn es darum geht Binärdaten in XML Dateien zu speichern. "Armored" (z.B, in Text Form) PGP Cookie und Signaturen werden codiert mit Base64.


Es gibt Werkzeuge zum scannen von beliebigen Binärdateien nach Base64 Strings. Ein solch ein Scanner ist [base64scanner](https://github.com/DennisYurichev/base64scanner).

Ein weiteres Codierungssystem welches im UseNet und FidoNet sehr weit verbreitet war, ist UUencoding. Binärdateien sind in Phrack Magazine immernoch mit UUencoding codiert. Es hat eigentlich die gleichen Features, unterscheidet sich von Base64 jedoch insofern, dass der Dateiname auch im Header gespeichert wird.


### 5.4.2 Strings in Binär finden

Actually, the best form of Unix documentation is frequently running the `strings` command over a program's object code. Using `strings`, you can get a complete list of the program's hard-coded file name, environment variables, undocumented options, obscure error messages, and so forth.

The Unix-Haters Handbook

Das Standard UNIX `strings` Utility ist ein quick-n-dirty Weg um alle Strings in der Datei an zu schauen. Zum Beispiel, in der OpenSSH 7.2 sshd executable Datei gibt es einige Strings:

```
...
0123
0123456789
0123456789abcdefABCDEF.:/
%02x
...
%.100s, line %lu: Bad permitopen specification <%.100s>
%.100s, line %lu: invalid criteria
%.100s, line %lu: invalid tun device
...
%.200s/.ssh/environment
...
2886173b9c9b6fbdeda7a247cd636db38deaa.debug
$2a$06$r3.juUaHZDlIbQa02d59FuYxL1W9M81R1Tc92PoSNmzvpEqLkLGrK
...
3des-cbc
...
Bind to port %s on %s.
Bind to port %s on %s failed: %.200s.
/bin/login
/bin/sh
/bin/sh /etc/ssh/sshrc
...
D$4PQWR1
D$4PUj
D$4PV
```
Dort kann man Optionen, Fehler Meldungen, Datei Pfade, importierte dynamische Module, Funktionen und einige andere komische Strings (keys?) sehen. Es gibt auch nicht druckbare Zeichen—x86 Code enthält chunks von druckbaren ASCII Zeichen, bis ca 8 Zeichen.

Sicher, OpenSSH ist ein open-source Programm. Aber sich die lesbaren Strings eines unbekannten Programms an zuschauen ist meist der erste Schritt bei der Analyse.

grep kann genauso benutzt werden.

Hiew hat die gleichen Fähigkeiten (Alt-F6), genau wie der Sysinternals ProcessMonitor.

### 5.4.3 Error/debug Nachrichten

Debugging Messages sind auch sehr nützlich, wenn vorhanden. Auf gewisse weise, melden die debug Nachrichten was gerade im Programm vorherrscht. Oft schreiben diese printf()-ähnlichen Funktionen, in log-Dateien oder sie schreiben nirgends hin aber die calls zu den printf-ähnlichen Funktionen sind noch vorhanden, weil der build kein Debug build aber ein release ist.

Wenn lokale oder globale Variablen in Debug messages geschrieben werden, kann das auch hilfreich sein da man so an die Variablen Namen kommt. Zum Beispiel, eine solche Funktion in Oracle RDBMS ist ksdwr().

Textstrings mit Aussage sind auch hilfreich. Der IDA disassembler zeigt welche Funktion und von welchem Punkt aus ein spezifischer String benutzt wird. Manchmal passieren lustige Dinge dabei


Es ist möglich heraus zu finden welche Funktionen Fehler melden und unter welchen Bedingungen.

Übrigens, das ist für Kopierschutzsysteme oft der Grund kryptische Fehlermeldungen oder einfach nur Fehlernumber aus zu geben. Niemand ist glücklich darüber wenn der Softwarecracker den Kopierschutz besser versteht nur weil dieser durch eine Fehlermeldung ausgelöst wurde.

Ein Beispiel von verschlüsselten Fehlermeldungen gibt es hier: ?? on page ??.

### 5.4.4 Verdächtige magic strings

Manche Magic Strings die in Hintertüren benutzt werden sehen schon ziemlich verdächtig aus.


Tatsächlich, kann man den Magic String „userRpmNatDebugRpm26525557“ in der Firmware finden.

Der String war nicht googlebar bis die Information öffentlich über die Hintertür öffentlich verbreitet wurde.

Man würde solche Informationen natürlich auch nicht in irgendeinem RFC finden.

---

16 blog.yurichev.com
17 http://sekurak.pl/tp-link-httptftp-backdoor/
18 Request for Comments
Man würde auch keinen Algorithmus finden der solch seltsame Byte Sequenzen benutzt. 
Und es sieht auch nicht nach einer Fehler- oder Debugnachricht aus. 
Also es ist immer eine gute Idee so seltsamen Dinge genauer zu betrachten. 
Manchmal, sind solche Strings auch mit base64 codiert. 
Es ist also immer eine gute Idee diese Stings zu Decodieren und sie visuell zu durchsuchen, ein Blick kann schon genügen. 
Präziser gesagt, diese Methode Hintertüren zu verstecken nennt man „security through obscurity“.

5.5 assert() Aufrufe

Manchmal ist die Präsenz des assert() macro’s ebenfalls nützlich: allgemein erlaubt dieses Makro Rückschlüsse auf source code Dateinamen, Zeilen nummern und die Bedienung für das Makro im Code.


Listing 5.2: Example of informative assert() calls

| text:107D4B29 mov  dx, [ecx+42h] |
| text:107D4B20 cmp  edx, 1 |
| text:107D4B30 jz  short loc_107D4B4A |
| text:107D4B32 push  1ECh |
| text:107D4B37 push offset aWrite_c ; "write.c" |
| text:107D4B3C push offset aTdTd_planarcon ; "td->td_planarconfig == PLANARCONFIG_CON"... |
| text:107D4B41 call ds: __assert |
...
| text:107D52CA mov  edx, [ebp-4] |
| text:107D52CD and  edx, 3 |
| text:107D52D0 test edx, edx |
| text:107D52D2 jz  short loc_107D52E9 |
| text:107D52D4 push  58h |
| text:107D52D6 push offset aDumpmode_c ; "dumpmode.c" |
| text:107D52DB push offset aN30 ; "(n & 3) == 0" |
| text:107D52E0 call ds: __assert |
...
| text:107D6759 mov  cx, [eax+6] |
| text:107D675D cmp  ecx, 0Ch |
| text:107D6760 jle  short loc_107D677A |
| text:107D6762 push  208h |
| text:107D6767 push offset aLzw_c ; "lzw.c" |
| text:107D676C push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= BITS_MAX" |
| text:107D6771 call ds: __assert |

Es ist Empfehlenswert beides die Konditionen und die Datei Namen in „google“ zu suchen, was zu einer open-source library führen kann. Zum Beispiel, wenn wir „sp->lzw_nbits <= BITS_MAX“ in „google“ suchen, ist es absehbar das wir als Ergebnis Code aus der Open-Source library für die LZW Kompression bekommen.

5.6 Konstanten

Menschen, Programmierer eingeschlossen, neigen dazu Zahlen zu runden wie z.B 10, 100, 1000, im realen Leben so wie in ihrem Code.

Der angehende Reverse Engineer kennt diese Werte und ihre hexadezimale Repräsentation sehr gut: 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.
Die Konstanten 0xAAAAAAAA (0b10101010101010101010101010101010) und 0x55555555 (0b01010101010101010101010101010101) sind auch sehr populär— sie sind zusammengesetzt aus verändernden Bits.

Dies hilft Signale voneinander zu unterscheiden bei denen alle Bits eingeschaltet (0b1111 ...) oder ausgeschaltet (0b0000 ...) werden. Zum Beispiel wird die Konstante 0x55AA beim Boot Sektor, MBR\(^{19}\), und im ROM! von IBM-Kompatiblen Erweiterung Karten benutzt.

Manche Algorithmen, speziell die Kryptografischen benutzen eindeutige Konstanten, die mit der Hilfe von IDA einfach im Code zu finden sind.

Zum Beispiel, der MD5 Algorithmus initialisiert seine Internen Variablen wie folgt:

```c
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

Wenn man diese vier Konstanten im Code hintereinander benutzt findet, dann ist die Wahrscheinlichkeit das diese Funktion sich auf MD5 bezieht.

Ein weiteres Beispiel sind die CRC16/CRC32 Algorithmen, ihre Berechnungs Algorithmen benutzen oft vorberechnete Tabellen wie diese:

```c
Listing 5.3: linux/lib/crc16.c

/* CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + x^2 + 1) */
ulong const crc16_table[256] = {
    0x0000, 0x0c01, 0x0201, 0x0a01, 0x0801, 0x0e01, 0x0601, 0x0001,
    0x0c02, 0x0202, 0x0a02, 0x0802, 0x0e02, 0x0602, 0x0002, 0x0c03,
    0x0203, 0x0a03, 0x0803, 0x0e03, 0x0603, 0x0003, 0x0c04, 0x0204,
    0x0a04, 0x0804, 0x0e04, 0x0604, 0x0004, 0x0c05, 0x0205, 0x0a05,
    0x0805, 0x0e05, 0x0605, 0x0005, 0x0c06, 0x0206, 0x0a06, 0x0806,
    0x0e06, 0x0606, 0x0006, 0x0c07, 0x0207, 0x0a07, 0x0807, 0x0e07,
    0x0607, 0x0007, 0x0c08, 0x0208, 0x0a08, 0x0808, 0x0e08, 0x0608,
    0x0008, 0x0c09, 0x0209, 0x0a09, 0x0809, 0x0e09, 0x0609, 0x0009,
    0x0c0a, 0x020a, 0x0a0a, 0x080a, 0x0e0a, 0x060a, 0x000a, 0x0c0b,
    0x020b, 0x0a0b, 0x080b, 0x0e0b, 0x060b, 0x000b, 0x0c0c, 0x020c,
    0x0a0c, 0x080c, 0x0e0c, 0x060c, 0x000c, 0x0c0d, 0x020d, 0x0a0d,
    0x080d, 0x0e0d, 0x060d, 0x000d, 0x0c0e, 0x020e, 0x0a0e, 0x080e,
    0x0e0e, 0x060e, 0x000e, 0x0c0f, 0x020f, 0x0a0f, 0x080f, 0x0e0f,
    0x060f, 0x000f,
};
```

Man beachte auch die vorberechnete Tabelle für CRC32: ?? on page ??.

In tabellenlosen CRC-Algorithmen werden bekannte Polynome benutzt, zum Beispiel, 0xEDB88320 für CRC32.

### 5.6.1 Magic numbers

Viele Datei-Formate definieren einen Standard-Dateiheader in dem eine *magic number(s)* benutzt wird, einzelne oder sogar mehrere.

Zum Beispiel, alle Win32 und MS-DOS executable starten mit zwei Zeichen „MZ“.

Am Anfang einer MIDI Datei muss die „MThd“ Signatur vorhanden sein. Wenn wir ein Programm haben das auf MIDI Dateien zugreift um sonst was zu machen, ist es sehr wahrscheinlich das das Programm die Datei validieren muss in dem es mindestens das ersten 4 Bytes prüft.

Das kann man wie folgt realisieren: (buf Zeigt auf den Anfang der geladenen Datei im Speicher)

```c
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

...oder durch das Aufrufen der Funktion für das vergleichen von Speicherblöcken wie z.B memcmp() oder beliebigen anderen Code bis hin zu einer CMPSB (.1.6 on page 536) Instruktion.

Wenn man so einen Punkt findet kann man bereits sagen das eine MIDI Datei geladen wird, wir können auch sehen wo der Puffer mit den Inhalten der MIDI Datei liegt und was/wie aus diesem Puffer verwendet wird.

**Daten**

Oft findet man auch nur eine Zahl wie 0x19870116, was ganz klar nach einem Jahres Datum aussieht (Tag 16, 1 Monat (Januar), Jahr 1987). Das ist vielleicht das Geburtsdatum von jemandem (ein Programmierer. ihre/seine bekannte, Kind), oder ein anderes wichtiges Datum. Das Datum kann auch in umgekehrter

\(^{19}\)Master Boot Record
folge auftreten, wie z.B. 0x16011987. Datumsangaben im Amerikanischen-Stil sind auch weit verbreitet wie 0x01161987.

Ein ziemlich bekanntes Beispiel ist 0x19540119 (magic number wird in der UFS2 Superblock Struktur benutzt), das Geburtsdatum von Marschall Kirk McKusick ist, einem Prominenten FreeBSD Entwickler.

Stuxnet benutzt die Zahl “19790509” (nicht als 32-Bit Zahl, aber als String), was zu Spekulationen geführt hat, ob die Malware Verbindungen nach Israel aufzeigt²⁰.

Solche Zahlen sind auch sehr beliebt in Amateur Kryptografie, zum Beispiel, ein Ausschnitt aus dem secret function Interna aus dem HASP3 Dongle ²¹:

```c
void xor_pwd(void)
{
    int i;
    pwd ^= 0x09071966;
    for (i = 0; i < 8; i++)
    {
        al_buf[i] = pwd & 7; pwd = pwd >> 3;
    }
}

void emulate_func2(unsigned short seed)
{
    int i, j;
    for (i = 0; i < 8; i++)
    {
        ch[i] = 0;
        for (j = 0; j < 8; j++)
        {
            seed *= 0x1989;
            seed += 5;
            ch[i] |= (tab[seed >> 9] & 0x3f) << (7 - j);
        }
    }
}
```

DHCP


Zum Beispiel, lasst uns die dhcpcore.dll Datei aus Windows 7 x64 analysieren die nach der Konstante suchen. Wir können die Konstante zweimal finden: Es sieht danach aus als wäre die Konstante in zwei Funktionen benutzt mit dem selbst redenden Namen DhcpExtractOptionsForValidation() und DhcpExtractFullOptions():

```
Listing 5.4: dhcpcore.dll (Windows 7 x64)
.rdata:000007FF6483CBE8 dword .7FF6483CBE8 dd 63538263h ; DATA XREF:
    DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBE8 dword .7FF6483CBE8 dd 63538263h ; DATA XREF:
    DhcpExtractFullOptions+97
```

Und hier die (Speicher) Orte an denen auf die Konstante zugegriffen wird:

```
Listing 5.5: dhcpcore.dll (Windows 7 x64)
```

²⁰ Das ist das Datum der Hinrichtung von Habib Elghanian, persischer Jude.
5.6.2 Spezifische Konstanten


Das trifft auch auf Amateur Kryptografie zu (Lizenz Schlüssel, etc). Bei einem verschlüsselten Block von $n$ Bytes, will man versuchen die vorkommen dieser Nummer im Code zu suchen, auch, wenn man ein Stück Code sieht der sich $n$ mal während einer Schleifen Ausführung wiederholt, ist das vielleicht eine ver-/Entschlüsselung Routine.

5.6.3 Nach Konstanten suchen

Das ist einfach mit IDA: Alt-B oder Alt-I. Und für das suchen von Konstanten in einem Haufen großer Dateien, oder für das suchen in nicht ausführbaren Dateien, gibt es ein kleines Utility genannt binary grep\textsuperscript{22}.

5.7 Die richtigen Instruktionen finden

Wenn ein Programm auf die FPU Instruktionen zugreift und der Code selber enthält nur sehr wenige dieser Instruktionen, kann man diese einzeln mit einem Debugger überprüfen.

Zum Beispiel, eventuell haben wir Interesse daran wie Microsoft Excel die Formel berechnet die vom Benutzer eingegeben wurde. Zum Beispiel die Division Operation.

Wenn wir excel.exe (von Office 2010) in Version 14.0.4756.1000 in IDA laden, ein komplettes Listig erstellen und jede FDIV Instruktion anschauen (ausgenommen die Instruktionen die eine Konstante als zweiten Parameter haben—diese Instruktionen interessieren uns nicht)

```bash
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

...dann sehen wir das es 144 FPU Instruktionen gibt.

Wir können einen String wie z.B. =((1/3) in Excel eingeben und dann die Instruktionen überprüfen.

Beim prüfen jeder dieser Instruktionen in einem Debugger oder German text placeholder (manche Prüfen 4 Instruktionen auf einmal), haben wir Glück und die gesuchte Instruktion ist die Nummer 14:

```assembly
.text:3011E919 DC 33 fdiv qword ptr [ebx]
```

\textsuperscript{22}GitHub
Die Instruktion nach FDIV (FSTP) schreibt jedes Ergebnis in den Speicher:

```
.text:3011E91B DD 1E fstp qword ptr [esi]
```

Wenn wir einen Breakpoint auf diese Instruktion setzen können wir das Ergebnis betrachten:

```
tracer -L:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

Excel zeigt nun 666 in unserer Zelle, was uns letztendlich auch bestätigt das wir das richtige Ergebnis gefunden haben.
Abbildung 5.7: Der Scherz hat funktioniert

Wenn wir das gleiche mit der selben Excel Version versuchen, jedoch in 64-Bit Umgebungen. Dann finden wir nur noch 12 FDIV Instruktionen und die Instruktion nach der wir suchen ist die dritte.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC, set(st0,666)
```

Es sieht danach aus als wären viele der Division Operationen der float und double Typen, vom Compiler mit SSE Instruktionen ersetzt wurden. Wie z.B DIVSD (DIVSD kommt insgesamt 268 mal vor).

### 5.8 Verdächtige Code muster

#### 5.8.1 XOR Instruktionen

Instruktionen wie XOR op, op (zum Beispiel, XOR EAX, EAX) werden normal dafür benutzt Register Werte auf Null zu setzen, wenn jedoch einer der Operanden sich unterscheidet wird die „exclusive or“ Operation ausgeführt.


Das könnte ein Hinweis sein das das etwas ver-/entschlüsselt wird oder Checksumme berechnet werden, etc. Eine Ausnahme dieser Beobachtung ist der „canary“ (1.18.3 on page 258). Die Generierung und das prüfen des „canary“ werden oft mit Hilfe der XOR Instruktion gemacht.

Dieses AWK Skript kann benutzt werden um IDA listing (.lst) Dateien zu parsen:

```
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$4) if($4!="esp") if ($4!="ebp") \n { print $1, $2, tmp, ",", $4 } }' filename.lst
```

Es sollte auch noch erwähnt werden das diese Art von Skript in der Lage ist inkorrekt disassemblierten Code zu erkennen (?? on page ??).

#### 5.8.2 Hand geschriebener Assembler code

Moderne Compiler benutzen keine LOOP und RCL Instruktionen. Auf der anderen Seite sind diese Instruktionen sehr beliebt bei Programmieren die Code direkt in Assembler schreiben. Wenn man diese Instruktionen sieht, kann man mit hoher Sicherheit sagen das dieses Code Fragment händig geschrieben wurde., Diese Instruktionen sind in der Instruktionsliste im Anhang mit (M) markiert: 1.6 on page 533.

Die Funktions Prolog und Epilog sind allgemein nicht vorhanden bei handgeschriebenen Assembler Code.
Tatsächlich gibt es kein bestimmtes System um Argumente an Funktionen zu übergeben wenn der Code handgeschrieben wurde.

Beispiel aus dem Windows 2003 Kernel (ntoskrnl.exe file):

```asm
MultiplyTest proc near
    ; CODE XREF: Get386Stepping
loc_620555:
    xor cx, cx
    ; CODE XREF: MultiplyTest+E
    push cx
    call Multiply
    pop cx
    jb short locret_620563
    loop loc_620555
    clc
locret_620563:
    ; CODE XREF: MultiplyTest+C
    retn

MultiplyTest endp

Multiply proc near
    ; CODE XREF: MultiplyTest+5
    mov ecx, 81
    mov eax, 417A000h
    mul ecx
    cmp edx, 2
    stc
    jnz short locret_62057F
    cmp eax, 0FE7A000h
    stc
    jnz short locret_62057F
    clc
locret_62057F:
    ; CODE XREF: MultiplyTest+10
    ; Multiply+18
    retn

Multiply endp
```

Tatsächlich, wenn wir in den WRK\textsuperscript{23} v1.2 source code schauen, kann dieser Code einfach in der Datei WRK-v1.2\textbackslash base\ntos\ke\i386\cpu.asm gefunden werden.

### 5.9 Using magic numbers while tracing

Oft ist unser Hauptziel zu verstehen wie ein Programm einen Wert behandelt der entweder über eine Datei oder über das Netzwerk erhalten wurde. Das manuelle tracen eines Wertes ist meistens ein ziemlich arbeits-intensiver Task. Eine der einfachsten Techniken um Werte zu Tracen (auch wenn nicht 100% verlässlich) ist eigene magic number's zu benutzen.

Das ähnelt ein wenig dem Vorgang beim Röntgen auf gewisser weise: ein radioaktives Kontrastmittel wird dem Patienten injiziert, welches dann benutzt wird um die Gefässe des Patienten besser zu erkennen durch die Röntgenstahlung. Wie das blut bei gesunden Menschen in den Nieren gereinigt wird wenn das Kontrastmittel im Blut ist, man kann dann sehr einfach auf dem Bild der Tomografie erkennen ob sich Nierensteine oder Tumore in den Nieren befinden.

Wir können einfach eine 32-Bit Zahl nehmen z.B 0xbadf00d, oder ein Geburtsdatum wie 0x11101979 und diese 4-Byte Zahl wird an einem bestimmten Punkt in eine Datei geschrieben welche von dem Programm das wir untersuchen genutzt wird.

Dann während das programm getraced wird mit German text placeholder im code coverage modus, mit der Hilfe von grep oder durch einfaches durchsuchen der Textdatei (der trace Ergebnisse), können wir ganz einfach sehen wo der Wert benutzt wurde und wie er benutzt wurde.

Beispiel der grepable German text placeholder Ergebnissen im cc mode:

\begin{verbatim}
0x150bf66 (kziaia+0x14), e= 1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934
0x150bf69 (kziaia+0x17), e= 1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (kziaia+0x1d), e= 1 [FS: MOV EAX, [2Ch]]
0x150bf75 (kziaia+0x23), e= 1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0xf1ac360
0x150bf78 (kziaia+0x26), e= 1 [MOV [EBP-4], ECX] ECX=0xf1ac360
\end{verbatim}

\textsuperscript{23}Windows Research Kernel
Das gleiche Verfahren kann man auch auf Netzwerkpakete anwenden. Für die *magic number* ist es wichtig, dass diese einzigartig ist und nicht im Programmcode vorkommt.

Neben dem *German text placeholder* Befehl, gibt es noch den DosBox (MS-DOS emulator) im heavydebug Modus, welcher in der Lage ist, alle Informationen über alle Registerzustände für jede ausgeführte Instruktion des Programms in eine einfache Textdatei zu schreiben, so kann diese Technik für DOS-Programme nützlich sein.

### 5.10 Schleifen


```assembly
0x45a6b5 e= 1 [FS: MOV [0], EAX] EAX=0x218f8b08
0x45a6bb e= 1 [MOV [EBP-254h], ECX] ECX=0x218f8bd8
0x45a6c1 e= 1 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218f8bd8
0x45a6c7 e= 1 [CMP [EAX+14h], 0] [EAX+14h]=0x102
0x45a6cb e= 1 [JZ 45A9F2h] ZF=false
0x45a6d1 e= 1 [MOV [EBP-0Dh], 1]
0x45a6d5 e= 1 [XOR ECX, ECX] ECX=0x218f8bd8
0x45a6d7 e= 1 [MOV [EBP-14h], CX] CX=0
0x45a6db e= 1 [MOV [EBP-16h], 0]
0x45a6e2 e= 1 [JMP 45A6EDh]
0x45a6e4 e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0xfd..0x101
0x45a6e7 e= 258 [ADD EDX, 1] EDX=0..5 (248 items skipped) 0xfd..0x101
0x45a6ea e= 258 [MOV [EBP-18h], EDX] EDX=1..6 (248 items skipped) 0xfe..0x102
0x45a6ed e= 259 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218f8bd8
0x45a6f3 e= 259 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (249 items skipped) 0xfe..0x102
0x45a6f6 e= 259 [CMP ECX, [EAX+14h]] EAX=0..5 (249 items skipped) 0xfe..0x102 [EAX+14h]=0x102
0x45a6f9 e= 259 [JNB 45A727h] CF=false,true
0x45a6fb e= 258 [MOV EDX, [EBP-254h]] [EBP-254h]=0x218f8bd8
0x45a701 e= 258 [MOV EAX, [EDX+10h]] [EDX+10h]=0x21ee4c8
0x45a704 e= 258 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0xfd..0x101
0x45a707 e= 258 [ADD ECX, 1] ECX=0..5 (248 items skipped) 0xfd..0x101
0x45a70a e= 258 [IMUL ECX, ECX, 1Fh] ECX=1..6 (248 items skipped) 0xfe..0x102
0x45a70d e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0xfd..0x101
0x45a710 e= 258 [MOVZX EAX, [EAX+EDX]] [EAX+EDX]=1..6 (156 items skipped) 0xf3, 0xf8, 0xf9, 0xfc, 0xfd
0x45a714 e= 258 [XOR EAX, ECX] EAX=1..6 (156 items skipped) 0xf3, 0xf8, 0xf9, 0xfc, 0xfd ECX=0
0x45a716 e= 258 [JMP 45A6EDh]
0x45a717 e= 258 [XOR ECX, ECX] ECX=0
0x45a718 e= 258 [MOV [EBP-254h], EAX] [EBP-254h]=0x218f8bd8
0x45a719 e= 258 [MOV [EBP-18h], ECX] [EBP-18h]=0x21ee4c8
0x45a71a e= 258 [MOV [EBP-10h], EAX] [EBP-10h]=0x218f8bd8
0x45a71b e= 258 [JMP 45A727h] CF=false,true
```

Wie sich herausstellte war dies auch eine Decodier-Schleife.

---

24 See also my blog post about this DosBox feature: blog.yurichev.com
5.10.1 Muster in Binärdateien finden

Arrays

Manchmal kann man klar ein Array von 16/32/64-Bit Werten mit bloßem Auge im hex Editor erkennen. Hier ist ein Beispiel eines 16-Bit Wertes. Wir sehen das das erste Byte ein paar aus 7 oder 8 ist und das zweite sieht zufällig aus:

Abbildung 5.8: FAR: array von 16-Bit Werten

Ich habe eine Datei benutzt die ein 12 Kanal Signal digitalisiert mit 16-Bit nutzt ADC\textsuperscript{25}.
Und hier ist ein Beispiel von einem Typischen MIPS Code.

Wie wir uns vielleicht erinnern, jede MIPS (also auch ARM in ARM Mode oder ARM64) Instruktion hat eine Größe von 32 Bits (oder 4 Bytes), also ist solcher Code ein Array von 32-Bit Werten.

Wenn man den Screenshot anschaut, sehen wir eine Art Muster. Vertikale und rote Linien wurden zur besseren Lesbarkeit eingefügt:

Abbildung 5.9: Hiew: sehr typischer MIPS code

Ein weiteres Beispiel eines solchen Musters ist Buch: ?? on page ??.
Sparse Dateien


Abbildung 5.10: FAR: Sparse file
Komprimierte Dateien

Diese Datei ist einfach ein komprimiertes Archiv. Es hat eine relativ hohe Entropie und visuell betrachtet sieht es eher Chaotisch aus. So sehen komprimierte oder verschlüsselte Dateien aus.

Abbildung 5.11: FAR: Komprimierte Datei

Abbildung 5.12: FAR: ISO file: Ubuntu 15 Installation **CD**[28]
32-bit x86 ausführbarer Code

So sieht 32-Bit x86 ausführbarer Code aus. Der Code hat nicht wirklich viel Entropie, weil manche Bytes öfters vorkommen als andere.

Abbildung 5.13: FAR: Executable 32-bit x86 code
**BMP graphics files**

BMP Dateien sind nicht komprimiert, also ist jedes Byte (oder Gruppen von Bytes) beschrieben als ein Pixel. Diese Bild habe ich irgendwo in meiner Windows 8.1 Installation gefunden:

Abbildung 5.14: Example picture

Man kann sehen das dieses Bild Pixel hat, die nicht wirklich gut komprimiert werden könne (um das Zentrum herum), aber es sind lange ein-Farben Linien am Anfang und am Ende der Datei. Tatsächlich Linien wie diese sehen wie Linien aus wenn man sich die Datei anschaut:

Abbildung 5.15: BMP file fragment

**5.10.2 Memory „snapshots“ comparing**

Die Technik zwei Memory Snapshots zu vergleichen ist recht einfach, das hat man auch oft benutzt um 8-Bit Computerspiele und „high score“’s zu hacken.

Zum Beispiel, wenn man ein geladenes Spiel auf einem 8-Bit Computer hat (auf den Maschinen ist nicht viel Speicher vorhanden, jedoch braucht das Spiel noch weniger Speicher) und du weißt was du im Spiel
hast, sagen wir 100 Patronen, nun kann man einen „snapshot“ vom gesamten Speicher machen und
diesen Irgendwohin speichern. Dann verschiesst man eine Patrone, dann geht der Patronen Zähler auf 99,
nun erstellt man den zweiten Snapshot und Vergleich die beiden: Nun muss es irgendwo ein Byte geben
das vorher 100 war und jetzt 99 ist.

Betrachtet man den Fakt das diese 8-Bit Spiele oftmals in Assembler geschrieben wurden und diese
Variablen meist global waren, konnte man ziemlich einfach bestimmen welche Adressen im Speicher den
Kugelzähler beinhalten. Wenn man nach allen Referenzen der Adresse im disassembelten Spiel code
sucht, ist es nicht schwer den Code decrementing zu finden und dann eine German text placeholder
Instruktion an diese Stelle zu schreiben, oder gar mehrere German text placeholder-s, und dann hat
man ein Spiel bei dem man für immer 100 Kugeln hat. Spiele auf 8-Bit Computern wurden allgemein
an konstanten Adressen geladen, zusätzlich gab es nicht viele unterschiedliche Versionen des Spiels (Es war meist eine Version für lange Zeit populär ), dadurch wussten enthusiastische Gamer welche Bytes
(durch das benutzen von Basic Instruktionen wie German text placeholder) überschrieben werden mussten
um das Spiel zu hacken. Das hat wiederum zu „cheat“ listen geführt die in Magazinen für 8-Bit Games
erschienen, die dann German text placeholder Instruktionen enthielten.

Es ist auch einfach „high score“ Dateien zu modifizieren, das funktioniert nicht nur bei 8-Bit Spielen. Man
achte auf seinen Highscore Zähler, dann macht man ein Backup der Datei. Wenn sich der „high score“
Zähler ändert, vergleicht man die zwei Dateien miteinander, das kann man sogar mit dem DOS Tool FC
(„high score“ Dateien, sind oft in Binärer Form).

Es wird beim Vergleichen der Dateien einen Punkt geben wo einige Bytes sich unterscheiden und es wird
leicht sein, die Punkte zu sehen die die Bytes des Punktezähler beinhalten. Jedoch sind sich die Spiele
Entwickler solcher Tricks bewusst und bauen Wege ein um das Programm vor solchen Manipulationen zu
schützen.

Ein ähnliches Beispiel findet man auch in dem Buch ?? on page ??.

Windows registry

Es ist auch möglich die Windows Regestry zu vergleichen vor und nach der Programm Installation.

Es ist eine sehr populäre Methode Regestry Elemente zu finden die vom Programm benutzt werden.
Vielleicht ist das auch der Grund warum die „windows regestry cleaner“ Shareware so populär ist.

Blink-comparator

Der Vergleich von Datei- oder Speichersnapshots erinnert ein wenig an einen Blinkkomparator ein Gerät
das in der Vergangenheit von Astronomen benutzt wurde, um sich bewegende Astronomische Objekte zu
finden.

Ein Blinkkomperator erlaubt es schnell zwischen Photographie zu wechseln die zu unterschiedlicher Zeit
aufgenommen wurden, so kann ein Astronom Unterschiede zwischen Fotografien visuell erkennen.

Ach übrigens, Pluto wurde durch einen solchen Blink-Komparator 1930 entdeckt.

5.11 Andere Dinge

5.11.1 Die Idee

Ein Reverse Engineer sollte versuchen so oft wie möglich in den Schuhen des Programmierers zu laufen.
Um ihren/seinen Standpunkt zu betrachten uns sich selbst zu Fragen wie man einen Task in spezifischen
Fällen lösen würde.

5.11.2 Anordnung von Funktionen in Binär Code

Sämtliche Funktionen die in einer einzelnen .c oder .cpp-Datei gefunden werden, werden zu den entsprechenden
Objekt Dateien (.o) kompiliert. Später, fügt der Linker alle Objektdatein die er braucht zusammen, ohne
die Reihenfolge oder die Funktionen in Ihnen zu verändern. Als eine Konsequenz, ergibt sich daraus wenn
man zwei oder mehr aufeinander folgende Funktionen sieht, bedeutet dass das sie in der gleichen Source
Code Datei platziert waren (Außer natürlich man bewegt sich an der Grenze zwischen zwei Dateien.). Das
bedeutet das diese Funktionen etwas gemeinsam haben, das sie aus dem gleichen API-Level stammen
oder aus der gleichen Library, etc.

29MS-DOS Utility zum vergleichen von Dateien
30http://go.yurichev.com/17348
5.11.3 kleine Funktionen

Sehr kleine oder leere Funktionen (1.3 on page 5) oder Funktionen die nur “true” (1) oder “false” (0) (?? on page ??) sind weit verbreitet, und fast jeder ordentlicher Compiler tendiert dazu nur solche Funktionen in den resultierenden ausführbaren Code zu stecken, sogar wenn es mehrere gleiche Funktionen im Source Code bereits gibt. Also, wann immer man solche kleinen Funktionen sieht die z.B nur aus 

```
mov eax, 1
```

/ ret bestehen und von mehreren Orten aus referenziert werden (und aufgerufen werden können), und scheinbar keine Verbindung zu einander haben, dann ist das wahrscheinlich das Ergebnis einer Optimierung.

5.11.4 C++

RTTI³¹ (?? on page ??)-data ist vielleicht auch nützlich für die C++ Klassen Identifikation.

³¹ Run-Time Type Information
Kapitel 6

Betriebssystem-spezifische Themen

6.1 Methoden zur Argumentenübergabe (Aufrufkonventionen)

6.1.1 cdecl
Hierbei handelt es sich um die am weitesten verbreitete Methode um in C/C++-Sprachen Argumente an Funktionen zu übergeben.

Der caller muss den Wert des Stapel-Zeiger (ESP) auf den ursprünglichen Stand bringen, nachdem callee-Funktion beendet wurde.

Listing 6.1: cdecl

```assembly
push Argument3
push Argument2
push Argument1
call Funktion
add esp, 12 ; gibt ESP zurück
```

6.1.2 stdcall
Dies ist fast gleich zu der cdecl-Aufrufkonvention, mit Ausnahme, dass die callee den Wert von ESP auf den ursprünglichen Wert setzen muss. Dies geschieht durch die RET x anstatt RET, wobei gilt x = Nummer des Arguments * sizeof(int)\(^1\).

Der caller passt den Stapel-Zeiger nicht an, es sind also keine add esp, x-Anweisungen vorhanden.

Listing 6.2: stdcall

```assembly
push Argument3
push Argument2
push Argument1
call Funktion

Funktion:
... tue etwas ...
ret 12
```

Diese Methode ist in Win32-Standard-Bibliotheken allgegenwärtig, fehlt jedoch in Win64 (siehe unten). Beispielsweise kann die Funktion von 1.64 on page 80 genommen werden und durch Hinzufügen des __stdcall-Modifizierers leicht verändert werden:

```c
int __stdcall f2 (int a, int b, int c)
{
    return a*b+c;
}
```

\(^1\)Die Größe einer Variablen vom Datentyp int ist 4 in x86-Systemen und 8 in x64-Systemen
Das Kompilat ist fast das gleiche wie bei 1.65 on page 81, jedoch wird RET 12 anstatt RET genutzt. Der SP wird im caller nicht aktualisiert.

Als Konsequenz daraus kann die Anzahl der Funktionsargumente einfach von der RETN n-Anweisung abgeleitet werden, indem n durch 4 geteilt wird

Listing 6.3: MSVC 2010

```
a$ = 8 ; size = 4
b$ = 12 ; size = 4
c$ = 16 ; size = 4
_f2@12 PROC
push ebp
mov ebp, esp
mov eax, DWORD PTR _a$[ebp]
imul eax, DWORD PTR _b$[ebp]
add eax, DWORD PTR _c$[ebp]
pop ebp
ret 12
_f2@12 ENDP
```

Funktionen mit einer variablen Anzahl von Argumenten

printf()-ähnliche Funktionen sind vielleicht die einzigen Funktionen in C/C++ mit einer Variablen Anzahl von Argumenten, aber mit ihnen kann ein wichtiger Unterschied zwischen cdecl und stdcall veranschaulicht werden. Beginnen wir mit der Vorstellung, dass der Compiler die Anzahl der Argumente für jeden Aufruf von printf() kennt.


6.1.3fastcall

Dies ist der allgemeine Name für eine Methode, in der einige Argumente mittels Registern und der Rest über den Stack übergeben werden. Für ältere CPUs istfastcall schneller als cdecl und stdcall wegen der geringeren Stack-Nutzung. Auf neueren CPUs wird dieser Ansatz vermutlich keine signifikante Geschwindigkeitsverbesserung nach sich ziehen.

fastcall ist nicht standardisiert, so das verschiedene Compiler eine unterschiedliche Umsetzung machen können. Dies kann zu Problemen führen, wenn zwei DLLs genutzt werden, von denen eine die andere nutzt und durch das Nutzen verschiedener Compiler unterschiedlichefastcall-Aufrufkonventionen genutzt werden.

Sowohl MSVC als auch GCC übergeben das erste und zweite Argument über ECX und EDX und den Rest der Arguments mittels des Stacks.

Der Stapel-Zeiger muss vom callee auf den ursprünglichen Wert gesetzt werden, wie in stdcall auch.

Listing 6.4:fastcall

```
push Argument3
mov edx, Argument2
```
Beispielsweise kann die Funktion von 1.64 on page 80 genommen werden und durch Hinzufügen des __fastcall-Modifizierers leicht verändert werden:

```c
int __fastcall f3 (int a, int b, int c)
{
    return a*b+c;
};
```

Nachfolgend das Kompilat:

```
c$ = 8 ; size = 4
@f3@12 PROC 
; \a$ = ecx
; \b$ = edx
    mov eax, ecx
    imul eax, edx
    add eax, DWORD PTR _c$[esp-4]
    ret 4
@f3@12 ENDP
; ...
    mov edx, 2
    push 3
    lea ecx, DWORD PTR [edx-1]
    call @f3@12
    push eax
    push OFFSET $SG81390
    call _printf
    add esp, 8
```

Es ist erkennbar, dass der callee den SP mit der RETN-Anweisung und einem Operanden auf den ursprünglichen Wert setzt. Dies bedeutet, dass die Anzahl der Argumente ebenfalls einfach abgeleitet werden kann.

GCC regparm

Dies ist in gewisser Weise die Weiterentwicklung vonfastcall\(^2\). Mit der -mregparm-Option ist es möglich festzulegen, wieviele Argumente per Register übergeben werden (maximal 3). Aus diesem Grund werden die Register EAX, EDX und ECX genutzt.

Natürlich werden, wenn die Anzahl der Argumente kleiner als drei ist, nicht alle drei Register genutzt.


Watcom/OpenWatcom

Hier erfolgt der Aufruf mit der „Register-Aufruf-Konvention“. Die ersten vier Argumente werden in den Registern EAX, EDX, EBX und ECX übergeben, der Rest auf dem Stack.

Diese Funktionen haben einen Unterstrich an den Funktionsnamen angehängt, um sie von den anderen Aufrufkonventionen unterscheiden zu können.

\(^2\)http://go.yurichev.com/17040

450
6.1.4 thiscall
Hier wird der this-Zeiger des Objekts an die Methode in C++ übergeben.
In MSVC wird this üblicherweise im ECX-Register übergeben.
In GCC wird der this-Zeiger im ersten Argument der Methode übergeben. Es ist sehr offensichtlich dass intern alle Methoden ein zusätzliches Argument haben.
Als Beispiel siehe (?? on page ??).

6.1.5 x86-64
Windows x64
Die Art Argumente zu Übergeben ähnelt in Win64 in gewisser Weise fastcall. Die ersten vier Argumente werden in den Registern RCX, RDX, R8 und R9 übergeben und der Rest auf dem Stack. Der caller muss Platz für 32 Byte, also 4 64-Bit-Werte bereitstellen, so dass der callee dort die ersten vier Argumente speichern kann. Kurze Funktionen können die Werte der Argumente direkt aus den Registern lesen, während längere Funktionen diese für späteren Gebrauch zwischenspeichern sollten.
Der caller muss den Stapel-Zeiger auf den vorherigen Zustand zurücksetzen.
Diese Aufrufkonvention wird auch in den Windows x86-64-System-DLLs genutzt (anstatt stdcall in Win32).
Beispiel:

```c
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g) {
    printf("%d %d %d %d %d %d\n", a, b, c, d, e, f, g);
};

int main() {
    f1(1,2,3,4,5,6,7);
};
```

Listing 6.6: MSVC 2012 /0b

```assembly
$SG2937 DB '%d %d %d %d %d %d', 0aH, 00H
main PROC
    sub rsp, 72
    mov DWORD PTR [rsp+48], 7
    mov DWORD PTR [rsp+40], 6
    mov DWORD PTR [rsp+32], 5
    mov r9d, 4
    mov r8d, 3
    mov edx, 2
    mov ecx, 1
    call f1
    xor eax, eax
    add rsp, 72
    ret 0
main ENDP

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1 PROC
$LN3:  mov DWORD PTR [rsp+32], r9d
    mov DWORD PTR [rsp+24], r8d
    mov DWORD PTR [rsp+16], edx
```

451
Es ist hier klar erkennbar, wie sieben Argumente übergeben werden: vier in den Registern und die drei restlichen auf dem Stack.

Der Code des f1()-Funktionsprologs sichert die Argumente in dem „Scratch Space“, einer Stelle auf dem Stack, die genau für diese Zwecke existiert.

Dies ist so realisiert, weil der Compiler nicht sicher sein kann, dass genug Register ohne diese vier genutzt werden können und sie sonst durch die Argumente verändert werden können, bis die Funktion beendet wird.

Die Bereitstellung des „Scratch Space“ auf dem Stack ist Aufgabe der aufrufenden Funktion.

Listing 6.7: German text placeholder MSVC 2012 /0b

<table>
<thead>
<tr>
<th>$SG2777 DB</th>
<th>'%d %d %d %d %d %d %d', 0AH, 00H</th>
</tr>
</thead>
<tbody>
<tr>
<td>a$ = 80</td>
<td></td>
</tr>
<tr>
<td>b$ = 88</td>
<td></td>
</tr>
<tr>
<td>c$ = 96</td>
<td></td>
</tr>
<tr>
<td>d$ = 104</td>
<td></td>
</tr>
<tr>
<td>e$ = 112</td>
<td></td>
</tr>
<tr>
<td>f$ = 120</td>
<td></td>
</tr>
<tr>
<td>g$ = 128</td>
<td></td>
</tr>
<tr>
<td>f1 PROC</td>
<td></td>
</tr>
<tr>
<td>$LN3:</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>rsp, 72</td>
</tr>
<tr>
<td>mov</td>
<td>eax, DWORD PTR g$[rsp]</td>
</tr>
<tr>
<td>mov</td>
<td>DWORD PTR [rsp+56], eax</td>
</tr>
<tr>
<td>mov</td>
<td>eax, DWORD PTR f$[rsp]</td>
</tr>
<tr>
<td>mov</td>
<td>DWORD PTR [rsp+48], eax</td>
</tr>
<tr>
<td>mov</td>
<td>eax, DWORD PTR e$[rsp]</td>
</tr>
<tr>
<td>mov</td>
<td>DWORD PTR [rsp+40], eax</td>
</tr>
<tr>
<td>mov</td>
<td>eax, DWORD PTR d$[rsp]</td>
</tr>
<tr>
<td>mov</td>
<td>DWORD PTR [rsp+32], eax</td>
</tr>
<tr>
<td>mov</td>
<td>r9d, DWORD PTR c$[rsp]</td>
</tr>
<tr>
<td>mov</td>
<td>r8d, DWORD PTR b$[rsp]</td>
</tr>
<tr>
<td>mov</td>
<td>edx, DWORD PTR a$[rsp]</td>
</tr>
<tr>
<td>lea</td>
<td>rcx, OFFSET FLAT:$SG2937</td>
</tr>
<tr>
<td>call</td>
<td>printf</td>
</tr>
<tr>
<td>add</td>
<td>rsp, 72</td>
</tr>
<tr>
<td>ret</td>
<td>0</td>
</tr>
<tr>
<td>f1 ENDP</td>
<td></td>
</tr>
</tbody>
</table>

main PROC
| sub        | rsp, 72                          |
| mov        | edx, 2                            |
| mov        | DWORD PTR [rsp+48], 7             |
| mov        | DWORD PTR [rsp+40], 6             |
Wenn das Beispiel ohne Optimierung compiliert wird, ist das Ergebnis fast das gleiche, lediglich der „Scratch Space“ ist unnötig und wird nicht genutzt.

Beachtenswert ist auch, wie MSVC 2012 das Laden von einfachen Werten in Register durch das Nutzen von LEA (1.6 on page 534) optimiert. MOV wäre hier ein Byte länger (5 anstatt 4).

Ein weiteres Beispiel so eines Sachverhalts ist: ?? on page ??.

**Windows x64: Übergeben von this (C/C++)**

Der this-Zeiger wird in RCX übergeben, das erste Argument der Methode ist RDX, usw. Ein Beispiel ist hier zu sehen: ?? on page ??.

**Linux x64**

Die Art wie Linux für x86-64 Argumente übergibt ist fast die Gleiche wie in Windows, jedoch werden sechs anstatt vier Register genutzt (RDI, RSI, RDX, RCX, R8, R9) und es gibt keinen „Scratch Space“, auch wenn der callee die Registerwerte auf dem Stack speichern kann wenn er dies will oder muss.

**Listing 6.8: German text placeholder GCC 4.7.3**

Zur Beachtung: die Werte werden hier in die 32-Bit-Teile der Register (z.B. EAX) geschrieben, aber nicht in die kompletten 64-Bit-Register (RAX). Dies wird gemacht, weil jeder Schreibzugang auf die niederwertigen 32-Bit-Teile eines Registers automatisch den höherwertigen Teil zurücksetzt.

Vermutlich wurde dies bei AMD so eingeführt um die Portierung des Codes zu x86-64 zu vereinfachen.
6.1.6 Rückgabewerte von float- und double-Typen

In allen Konventionen außer in Win64, werden die Werte vom Typ float oder double in dem FPU-Register ST(0) zurückgegeben.

In Win64 werden die Werte vom Typ float oder double in den niederwertigen 32 oder 64 Bit des XMM0-Registers zurückgegeben.

6.1.7 Verändern von Argumenten

Manchmal fragen C/C++-Programmierer (obwohl nicht auf diese PS beschränkt), was passieren kann wenn die Funktionsargumente verändert werden.

Die Antwort ist einfach: die Argumente sind auf dem Stack gespeichert und hier werden auch die Veränderungen vorgenommen.

Die aufzurufende Funktion wird diese nicht nach dem Verlassen des callee nutzen (der Autor dieser Linien hat so einen Fall in der Praxis noch nie gesehen).

```c
#include <stdio.h>

void f(int a, int b)
{
    a = a + b;
    printf("%d\n", a);
}
```

Listing 6.9: MSVC 2012

Der Code kann wie folgt neu geschrieben werden:

```c
push 456 ; wird gleich b sein
push 123 ; wird gleich a sein
pop eax
add esp, 4
```

454
Es ist schier vorzustellen, warum jemand dies tun sollte, aber in der Praxis ist es möglich. Nichtsdestotrotz bietet der C/C++-Standard keinen Möglichkeit dies zu tun.

### 6.1.8 Einen Zeiger auf ein Argument verarbeiten

...mehr als das ist es sogar möglich, einen Zeiger auf ein Funktionsargument zu nehmen und an eine weitere Funktion zu übergeben:

```c
#include <stdio.h>

// located in some other file
void modify_a (int *a);

void f (int a)
{
    modify_a (&a);
    printf ("%d\n", a);
}
```

Es ist schwierig die Funktionsweise zu verstehen, aber der folgende Code bringt Klarheit:

Listing 6.10: German text placeholder MSVC 2010

```
SG2796  DB   \%d, 0AH, 00H
_a$ = 8
PROC
    lea    eax, DWORD PTR _a$[esp-4]; just get the address of value in local stack
    push   eax; and pass it to modify_a()
    call   _modify_a
    mov    ecx, DWORD PTR _a$[esp]; reload it from the local stack
    push   ecx; and pass it to printf()
    push   OFFSET $SG2796; \%d
    call   _printf
    add    esp, 12
    ret    0
ENDP
```

Die Adresse der Stelle im Stack an der \(a\) übergeben wird, wird lediglich an eine weitere Funktion übergeben. Diese verändert den Wert der mit dem Zeiger übergeben wird und printf() gibt anschließend den veränderten Wert aus.

Der aufmerksame Leser mag sich fragen, was mit der Aufrufkonvention ist, in der Funktionsargumente in Registern übergeben werden.

Das ist eine Situation, in der Shadow Space genutzt wird.

Der Eingangswert wird vom Register in den Shadow Space des lokalen Stacks kopiert und dann diese Adresse an die andere Funktion übergeben:

Listing 6.11: German text placeholder MSVC 2012 x64

```
SG2994  DB   \%d, 0AH, 00H
a$ = 48
PROC
    mov    DWORD PTR [rsp+8], ecx; save input value in Shadow Space
    sub    rsp, 40
    lea    rcx, DWORD PTR a$[rsp]; get address of value and pass it to modify_a()
    call   modify_a
    mov    edx, DWORD PTR a$[rsp]; reload value from Shadow Space and pass it to printf()
    printf()
        lea    rcx, OFFSET FLAT:$SG2994; \%d
    call   printf
    add    rsp, 40
    ret    0
ENDP
```

GCC sichert den Eingangswert ebenfalls auf dem lokalen Stack:
Listing 6.12: German text placeholder GCC 4.9.1 x64

```assembly
.LC0:
    .string "%d
"
f:
    sub    rsp, 24
    mov    DWORD PTR [rsp+12], edi ; store input value to the local stack
    lea    rdi, [rsp+12] ; take an address of the value and pass it to modify_a()
    call   modify_a
    mov    edx, DWORD PTR [rsp+12] ; reload value from the local stack and pass it to printf()

    mov    esi, OFFSET FLAT:.LC0 ; '%d'
    mov    edi, 1
    xor    eax, eax
    call   __printf_chk
    add    rsp, 24
    ret
```

GCC für ARM64 tut genau das gleiche, allerdings wird hier der Platz Register Save Area genannt:

Listing 6.13: German text placeholder GCC 4.9.1 ARM64

```assembly
f:
    stp    x29, x30, [sp, -32]!
    add    x29, sp, 0 ; setup FP
    add    x1, x29, 32 ; calculate address of variable in Register Save Area
    str    w0, [x1, -4]! ; store input value there
    mov    x0, x1 ; pass address of variable to the modify_a()
    bl     modify_a
    ldr    w1, [x29, 28] ; load value from the variable and pass it to printf()
    adrp   x0, .LC0 ; '%'d'
    add    x0, x0, :lo12:.LC0
    bl     printf ; call printf()
    ldp    x29, x30, [sp], 32
    ret
.LC0:
    .string "%d"
```

Übrigens eine gleiche Nutzung des Shadow Space wird auch hier beschrieben: ?? on page ??.

### 6.2 lokaler Thread-Speicher

TLS ist ein Datenbereich der für jeden einzelnen Thread spezifisch ist. Jeder Thread kann hier alles speichern was er möchte. Eine bekanntes Beispiel ist die globale C-Standardvariable `errno`.

Mehrere Threads können Funktionen simultan aufrufen, die einen Fehlercode in `errno` zurückgeben. Eine globale Variable würde hier nicht korrekt für Multithread-Programme funktionieren, aus diesem Grund muss `errno` im TLS gesichert.

Im C++11-Standard wurde ein neues Schlüsselwort `thread_local` eingeführt, um anzuzeigen, dass jeder Thread eine eigene Kopie dieser Variable, die initialisiert werde kann und sich auf dem TLS befindet:

Listing 6.14: C++11

```cpp
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
    std::cout << tmp << std::endl;
};
```

Kompiliert mit MinGW GCC 4.8.1 jedoch nicht MSVC 2012.

---

3 C11 hat ebenfalls einen (optionalen) Thread-Support
Wenn es um PE-Dateien geht, also die ausführbaren Dateien, wird die tmp-Variable in der Sektion mit dem Namen TLS gesichert.

### 6.2.1 Nochmals Linearer Kongruenzgenerator

Der Pseudozufallszahlen-Generator der in 1.20 on page 313 bereits erwähnt wurde hat einen Nachteil: Er ist nicht Thread-sicher, weil er eine interne Zustandsvariable hat, die von verschiedenen Threads gleichzeitig gelesen und verändert werden kann.

### Win32

**Uninitialisierte TLS-Daten**

Eine mögliche Lösung ist es den __declspec( thread )-Modifizierer zu der globalen Variable hinzuzufügen, so dass diese im TLS alloziert wird (Zeile 9):

```c
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223
__declspec(thread) uint32_t rand_state;

void my_srand(uint32_t init)
{
    rand_state=init;
}

int my_rand()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    my_srand(0x12345678);
    printf("%d\n", my_rand());
}
```

Hiew zeigt, dass eine neue PE-Sektion in der ausführbaren Datei existiert: .tls.

Listing 6.15: German text placeholder MSVC 2013 x86

```
_TLS SEGMENT
_rand_state DD 01H DUP (?)
_TLS ENDS

_DATA SEGMENT
$SG84851 DB '%d', 0aH, 00H
_DATA ENDS
.TEXT SEGMENT

_init$ = 8 ; size = 4
_my_srand PROC
; FS:0=address of TIB
    mov eax, DWORD PTR fs:_tls_array ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
    mov ecx, DWORD PTR __tls_index
    mov ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
    mov eax, DWORD PTR _init$[esp-4]
    mov DWORD PTR _rand_state[ecx], eax
    ret 0
```
rand_state befindet sich nun im TLS-Segment und jeder Thread hat seine eigene Kopie dieser Variable. Auf diese Variable wird wie folgt zugegriffen: lade die Adresse von TIB von FS:2Ch, anschließend addiere einen zusätzlichen Index (falls notwendig), zuletzt berechne die Adresse des TLS-Segments.

Nun ist es möglich auf die rand_state-Variable über des ECX-Register zuzugreifen, welches auf eine spezifische Stelle in jedem Thread zeigt.

Der FS:-Selektor ist bei jedem Reverse Engineer gut bekannt und zeigt immer auf TIB, so dass es schnell möglich ist thread-spezifische Daten zu laden.

Der GS:-Selektor wird unter Win64 genutzt, die Adresse von TLS ist 0x58:

Listing 6.16: German text placeholder MSVC 2013 x64

Initialisierte TLS-Daten
Angenommen es soll ein fester Wert für rand_state gesetzt werden und der Programmierer vergisst dies, wird dies automatisch in Zeile 9 gemacht:

```c
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223
__declspec(thread) uint32_t rand_state = 1234;

void my_srand(uint32_t init)
{
    rand_state = init;
}

int my_rand()
{
    rand_state = rand_state * RNG_a;
    rand_state = rand_state + RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    printf("%d\n", my_rand());
}
```

Der Code ist nicht anders als zuvor, aber in IDA sieht man folgendes:

Jedes mal wenn ein neuer Thread gestartet wird, wird ein neuer TLS alloziert und alle Daten, inklusive der 1234 dorthin kopiert.

Dies ist eine typische Situation:

- Thread A wird gestartet und ein TLS wird dafür erstellt. 1234 wird in rand_state kopiert.
- Die Funktion my_rand() wird einige Male in Thread A aufgerufen. rand_state unterscheidet sich 1234.
- Thread B wird gestartet und ein TLS wird dafür erstellt. 1234 wird in rand_state kopiert, während Thread A einen anderen Wert in der gleichen Variable hat.

### TLS-Callback-Funktionen

Was passiert wenn die Variablen in TLS mit Daten gefüllt werden, die in einer bestimmten Weise verändert werden sollen?

Angenommen es existiert folgende Aufgabe: Der Programmierer hat vergessen die my_srand()-Funktion aufzurufen um den PRNG zu initialisieren, dieser muss jedoch initialisiert werden um „echte“ Zufallszahlen anstatt den Wert 1234 zu erzeugen. In diesem Fall kann die TLS-Callback-Funktion genutzt werden.

Der folgende Code ist nicht sehr portabl, nichtsdestotrotz ist die Idee dahinter erkennbar.

Was hier passiert ist eine Funktion zu definieren (tls_callback()) welche vor dem Starten eines Prozesses oder Threads aufgerufen wird.
Die Funktion initialisiert den PRNG mit dem Wert der von GetTickCount() zurückgegeben wird.

```c
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec(thread) uint32_t rand_state;

void my_srand(uint32_t init)
{
    rand_state = init;
}

void NTAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
{
    my_srand(GetTickCount());
}

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data_seg()

int my_rand()
{
    rand_state = rand_state * RNG_a;
    rand_state = rand_state + RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    // rand_state is already initialized at the moment (using GetTickCount())
    printf("%d\n", my_rand());
}
```

Nachfolgend das Ergebnis in IDA:

<table>
<thead>
<tr>
<th>Listing 6.17: German text placeholder</th>
<th>MSVC 2013</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:00401020 TlsCallback_0 proc near</td>
<td>DATA XREF: .rdata:TlsCallbacks</td>
</tr>
<tr>
<td>.text:00401020</td>
<td>call ds:GetTickCount</td>
</tr>
<tr>
<td>.text:00401026</td>
<td>push eax</td>
</tr>
<tr>
<td>.text:00401027</td>
<td>call my_srand</td>
</tr>
<tr>
<td>.text:0040102C</td>
<td>pop ecx</td>
</tr>
<tr>
<td>.text:0040102D</td>
<td>retn 0Ch</td>
</tr>
<tr>
<td>.text:0040102D TlsCallback_0 endp</td>
<td></td>
</tr>
</tbody>
</table>

... 

.rdata:004020C0 TlsCallbacks dd offset TlsCallback_0 ; DATA XREF: .rdata:TlsCallbacks_ptr |

... 

.rdata:00402118 TlsDirectory dd offset TlsStart |
|rdata:0040211C TlsEnd_ptr dd offset TlsEnd |
|rdata:00402120 TlsIndex_ptr dd offset TlsIndex |
|rdata:00402124 TlsCallbacks_ptr dd offset TlsCallbacks |
|rdata:00402128 TlsSizeOfZeroFill dd 0 |
|rdata:0040212C TlsCharacteristics dd 300000h |

TLS-Callback-Funktionen werden manchmal genutzt um Routinen zu entpacken und deren Funktion zu verschleieren.

Manche Menschen sind verwirrt darüber, dass einiger Code genau vor dem OEP ausgeführt wird.

---

^Original Entry Point
Nachfolgend ein Beispiel wie eine thread-lokale, globale Variable in GCC deklariert wird:

```c
__thread uint32_t rand_state=1234;
``` Dies is ein Standard-C/C++-Modifizierer sondern eine GCC-spezifische Erweiterung. Der GS:-Selektor wird ebenso genutzt um auf den TLS-Bereich zuzugreifen, jedoch in einer etwas anderen Art:

Listing 6.18: German text placeholder GCC 4.8.1 x86

```assembly
.text:08048460 my_srand proc near
.text:08048460 .text:08048460 arg_0 = dword ptr 4
.text:08048460 .text:08048460 mov eax, [esp+arg_0]
.text:08048464 .text:08048464 mov gs:0FFFFFFFCh, eax
.text:0804846A .text:0804846A retn
.text:0804846A my_srand endp

.text:08048470 my_rand proc near
.text:08048470 .text:08048470 imul eax, gs:0FFFFFFFCh, 19660Dh
.text:0804847B .text:0804847B add eax, 3C6EF35Fh
.text:08048480 .text:08048480 mov gs:0FFFFFFFCh, eax
.text:08048486 .text:08048486 and eax, 7FFh
.text:0804848B .text:0804848B retn
.text:0804848B my_rand endp
```

Mehr darüber in [Ulrich Drepper, ELF Handling For Thread-Local Storage, (2013)]

### 6.3 Systemaufrufe

Wie bekannt werden alle laufende Prozesse in einem BS in zwei Kategorien unterteilt: diejenigen, die vollen Zugriff auf die Hardware haben („kernel space“) und diejenigen die dies nicht haben („user space“).

Der Kernel des Betriebssystems sowie die Treiber gehören in der Regel zur ersten Kategorie. Alle Anwendungen gehören in der Regel der zweiten Kategorie an.

Beispielsweise gehört der Linux-Kernel in den kernel space während Glibc im user space ausgeführt wird.

Diese Unterscheidung ist von großer Bedeutung für die Sicherheit eines BS: es ist sehr wichtig nicht jedem Prozess die Möglichkeit etwas in einem anderen Prozess oder sogar im BS-Kernel zum Absturz zu bringen. Auf der anderen Seite führt ein Fehler im Treiber oder innerhalb des BS-Kernels in der Regel zu einem Kernel-Panic oder BSOD.

Der Schutz im x86-Prozessor erlaubt die Unterscheidung in vier unterschiedliche Schutzlevel (Ringe). Sowohl von Linux als auch von Windows werden jedoch nur zwei genutzt: Ring 0 („kernel space“) und Ring 3 („user space“).

Systemaufrufe (syscalls) sind die Stelle an der diese beiden Bereiche miteinander verbunden werden. In diesem Sinne sind die Systemaufrufe die Haupt-API für die Anwendungen.

Unter German text placeholder, befindet sich die Tabelle mit Systemaufrufen in der SSDT.

Die Nutzung von Systemaufrufen ist sehr verbreitet bei Shellcode- und Viren-Programmierern, weil es schwieriger ist die Adresse einer benötigten Funktion herauszufinden als einen Systemaufruf zu nutzen. Die Kehrseite ist, das viel mehr Code geschrieben werden muss, aufgrund dem geringeren Grad an Abstraktion der API.

Erwähnenswert ist es das die Anzahl der Systemaufrufe bei den unterschiedlichen Betriebssystemen variieren kann.

---

5 [http://go.yurichev.com/17062](http://go.yurichev.com/17062)  
6 German text placeholder [http://go.yurichev.com/17272](http://go.yurichev.com/17272)  
7 Blue Screen of Death  
8 System Service Dispatch Table
6.3.1 Linux

Unter Linux wird ein Systemaufruf in der Regel mit int 0x80 aufgerufen. Die Nummer des Aufrufs wird im EAX-Register übergeben und die restlichen Parameter in anderen Registern.

Listing 6.19: Ein einfaches Beispiel zur Nutzung zweier Systemaufrufe

<table>
<thead>
<tr>
<th>Section .text</th>
<th>Global _start</th>
</tr>
</thead>
<tbody>
<tr>
<td>_start:</td>
<td></td>
</tr>
<tr>
<td>mov edx,len</td>
<td>buffer len</td>
</tr>
<tr>
<td>mov ecx,msg</td>
<td>buffer</td>
</tr>
<tr>
<td>mov ebx,1</td>
<td>file descriptor. 1 is for stdout</td>
</tr>
<tr>
<td>mov eax,4</td>
<td>syscall number. 4 is for sys_write</td>
</tr>
<tr>
<td>int 0x80</td>
<td></td>
</tr>
<tr>
<td>mov eax,1</td>
<td>syscall number. 1 is for sys_exit</td>
</tr>
<tr>
<td>int 0x80</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section .data</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg</td>
</tr>
<tr>
<td>len</td>
</tr>
</tbody>
</table>

Kompilation:

nasm -f elf32 1.s
ld 1.o

Um Systemaufrufe unter Linux zu unterbrechen und nachverfolgen zu können, kann strace(7.2.3 on page 505) genutzt werden.

6.3.2 Windows

Hier werden die Systemaufrufe via int 0x2e aufgerufen oder über die spezielle x86-Anweisung SYSENTER.

Weitere Informationen:

6.4 Linux

6.4.1 Positionsabhängiger Code

Wenn der Code von Shared Libraries (.so) unter Linux analysiert wird, findet man häufig das folgende Code-Muster:

Listing 6.20: libc-2.17.so x86

.text:001205E3 __x86_get_pc_thunk_bx proc near ; CODE XREF: sub_17350+3
.text:001205E3 ; sub_173CC+4 ...
.text:001205E3 mov ebx, [esp+0]
.text:001205E6 retn
.text:001205E6 __x86_get_pc_thunk_bx endp

...  
.text:000576C0 sub_576C0 proc near ; CODE XREF: tmpfile+73  
...  
.text:000576C0 push ebp  
.text:000576C1 mov ecx, large gs:0
Alle Zeiger auf Zeichenketten sind durch Konstanten und den Wert in EBX korrigiert, welcher zu Beginn jeder Funktion berechnet wird.

Dies ist sogenannter PIC\(^9\), und hat den Zweck an jeder beliebigen Stelle im Speicher ausführbar zu sein. Aus diesem Grund können keine absoluten Speicheradressen verwendet werden.

PIC war entscheidend in früheren Computer-Systemen und ist es immer noch in Eingebetteten Systemen ohne virtuelle Speicherverwaltung in denen sich alle Prozesse in einem einzigen durchgängigen Speicherbereich befinden.


Machen wir ein sehr einfaches Experiment:

```c
#include <stdio.h>

int global_variable=123;

int f1(int var)
{
   int rt=global_variable+var;
   printf ("returning %d\n", rt);
   return rt;
}
```

Nachfolgende die kompilierte .so-Datei von GCC 4.7.3. in IDA:

```bash
gcc -fPIC -shared -O3 -o 1.so 1.c
```

---

\(^9\)Position Independent Code
Das ist es: die Zeiger auf «returning %d\n» und global_variable werden bei jedem Funktionsaufruf korrigiert.

Die _x86_get_pc_thunk_bx()-Funktion gibt in EBX die Adresse auf eine Stelle nach einem Aufruf von sich selbst zurück (hier 0x57C).

Dies ist eine einfache Möglichkeit um den Wert des Programmzählers (EIP) an einer beliebigen Stelle zu erhalten. Die Konstante 0x1A84 gehört zu dem Unterschied des Funktionsanfangs und der sogenannten Global Offset Table Procedure Linkage Table (GOT PLT), die Sektion direkt hinter der Global Offset Table (GOT), an der der Zeiger auf global_variable.IDA zeigt diese Offsets aus Gründen des einfacheren Verständnisses in einer verarbeiteten Form an, der Code ist aber wie folgt:

Hier zeigt EBX auf die GOT PLT-Sektion und um den Zeiger auf global_variable zu berechnen (welcher in der GOT gesichert ist) muss 0x8C subtrahiert werden.

Um den Zeiger auf die Zeichenkette «returning %d\n» zu berechnen muss 0x1A30 abgezogen werden.

Übrigens ist dies der Grund warum die AMD64-Anweisungen RIP-relativen Adressierung unterstützen: sie vereinfachen den PIC-Code.

Nachfolgend der selbe C-Code mit der gleichen GCC-Version, jedoch für x64 kompiliert.

IDA würde den resultierenden Code vereinfachen aber auch die Details zur RIP-relativen Adressierung unterdrücken. Um alles sehen zu können wird hier objdump anstatt IDA genutzt.

---

10Programmzähler in AMD64
Die Adresse einer Anweisung an 0x2008b9 ist der Unterschied zwischen der Adresse der Anweisung an 0x720 und global_variable, und 0x20 ist der Unterschied zwischen der Adresse der Anweisung an 0x72A und der Zeichenkette «returning %d
».

Wie zu sehen ist, die Notwendigkeit die Adressen regelmäßig neu zu berechnen macht die Ausführung etwas langsamer (auch wenn dies in x64 etwas besser ist).

Es mag also von Vorteil sein, statisch zu linken wenn Geschwindigkeit eine Rolle spielt [siehe: Agner Fog, Optimizing software in C++ (2015)].

Windows

Der PIC-Mechanismus wird in Windows-DLLs nicht genutzt. Wenn der Windows-Loader eine DLL an eine andere Basisadresse laden muss, „patched“ er diese im Speicher (und den FIXUP-Platz) um alle Adressen zu korrigieren.

Dies bedeutet, dass mehrere Windows-Prozesse eine einmal geladene DLL nicht teilen können wenn diese an verschiedenen Adressen in verschiedenen Prozess-Speichern sein muss, da jede Instanz im Speicher die Funktionen an einer festen Adresse erwartet.

6.4.2 LD_PRELOAD-Hack in Linux

Diese Technik erlaubt es eigene, dynamische Bibliotheken vor anderen zu laden... sogar vor denen des Systems, wie libc.so.6.

Dies wiederum erlaubt es die eigenen Funktionen für die des Systems zu „ersetzen“. Es ist zum Beispiel einfach alle Aufrufe zu time(), read(), write(), usw. abzufangen.

Sehen wir uns einmal an, wie das Tool uptime ausgetrickst werden kann. Wie bekannt ist, zeigt dieses Programm an, wie lange der Computer schon arbeitet. Mithilfe von strace(7.2.3 on page 505), ist es möglich zu sehen, dass das Tool die Informationen aus der Datei /proc/uptime ausliest:

```
$ strace uptime
...
open("/proc/uptime", O_RDONLY) = 3
lseek(3, 0, SEEK_SET) = 0
read(3, "416166.86 414629.38\n", 2047) = 20
...
```

Es handelt sich dabei nicht um eine reale Datei auf der Festplatte sondern um eine virtuelle, bei der die Dateien on-the-fly im Linux Kernel erstellt werden. Es gibt zwei Zahlen:

```
$ cat /proc/uptime
416690.91 415152.03
```


The first number is the total number of seconds the system has been up. The second number is how much of that time the machine has spent idle, in seconds.

Versuchen wir eine eigene dynamische Bibliothek mit den Funktionen open(), read() und close() zu schreiben die so funktionieren wie wir es gerne hätten.

Zunächst wird unsere open()-Funktion den Namen der zu öffnenden Datei mit dem was wir brauchen vergleichen. Ist dies der Fall, soll der Deskriptor der geöffnete Datei geschrieben werden.

Als zweites read(): Wenn diese Funktion für den Datei-Deskriptor aufgerufen wird, soll die Ausgaben ersetzt werden und der Rest dem original read() aus libc.so.6 entsprechen. close() wird eine Meldung geben wenn die Datei der zur Zeit gefolgt wird geschlossen wurde.

Wir werden die dlopen()- und dlsym()-Funktionen nutzen, um die Adressen der Original-Funktionen in libc.so.6 herauszufinden.

Diese werden benötigt, weil die Ausführkontrolle wieder an die „realen“ Funktionen übergeben werden müssen.

Auf der anderen Seite: wenn wir strcmp() unterbrechen und jeden einzelnen Vergleich von Zeichenketten im Programm untersuchen, müssten wir eine eigene strcmp()-Variante schreiben und nicht die Original-Funktion nutzen\(^{12}\), was einfacher wäre.

\begin{verbatim}
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool inited = false;

_Noreturn void die (const char * fmt, ...)
{
    va_list va;
    va_start (va, fmt);
    vprintf (fmt, va);
    exit(0);
};

static void find_original_functions ()
{
    if (inited)
        return;

    libc_handle = dlopen("libc.so.6", RTLD_LAZY);
    if (libc_handle==NULL)
        die("can't open libc.so.6\n");

    open_ptr = dlsym (libc_handle, "open");
    if (open_ptr==NULL)
        die("can't find open()\n");

    close_ptr = dlsym (libc_handle, "close");
    if (close_ptr==NULL)
        die("can't find close()\n");

    read_ptr = dlsym (libc_handle, "read");
    if (read_ptr==NULL)
        die("can't find read()\n");

    inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
    find_original_functions();

    int fd=(*open_ptr)(pathname, flags);
    if (strcmp(pathname, "/proc/uptime")==0)
        opened_fd=fd; // that's our file! record its file descriptor
    else
        opened_fd=0;

\end{verbatim}

\(^{12}\)Als Beispiel, wie einfach strcmp()-Unterbrechung funktioniert\(^{13}\) von Yong Huang

466
```c
return fd;
}

int close(int fd)
{
    find_original_functions();
    if (fd==opened_fd)
        opened_fd=0; // the file is not opened anymore
    return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
    find_original_functions();
    if (opened_fd!=0 && fd==opened_fd)
    {
        // that's our file!
        return snprintf(buf, count, "%d %d", 0x7fffffff, 0x7fffffff)+1;
    }
    // not our file, go to real read() function
    return (*read_ptr)(fd, buf, count);
};
```

(Quellcode)
Kompilieren wir den Code als gemeinsame, dynamische Bibliothek:

```bash
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

Jetzt starten wir `uptime` während unsere Bibliothek vor den anderen geladen wird:

```bash
LD_PRELOAD=`pwd`/fool_uptime.so uptime
```

Und wir sehen:

```
01:23:02 up 24855 days, 3:14, 3 users, load average: 0.00, 0.01, 0.05
```

Wenn die `LD_PRELOAD`-Umgebungsvariable immer auf den Dateinamen und -pfad unserer Bibliothek zeigt, wird diese vor allen anderen gestarteten Programmen geladen.

Weitere Beispiele:

- Sehr einfache Unterbrechung von `strcmp()` (Yong Huang) [http://go.yurichev.com/17143](http://go.yurichev.com/17143)
- Kevin Pulo—Fun with `LD_PRELOAD`. Viele Beispiele und Ideen. [yurichev.com](http://yurichev.com)

### 6.5 Windows NT

#### 6.5.1 CRT (win32)

Startet ein Programm genau bei der `main()`-Funktion? Nein, tut es nicht!

Wir würden jede ausführbare Datei in IDA oder HIEW öffnen können, würde wir sehen, dass OEP auf einen anderen Code-Block zeigt.


Die `main()`-Funktion nimmt ein Array mit den Argumenten entgegen, die auf der Kommandozeile übergeben wurde, sowie eins mit den Umgebungsvariablen. Genaugenommen wird eine normale Zeichenkette an das
Programm übergeben und der CRT-Code unterteilt diesen anhand der Leerzeichen in seine Bestandteile. Der CRT-Code bereitet auch das Array `envp` vor, welches die Umgebungsvariablen enthält.

Für GUI\textsuperscript{14}-Programme unter Win32 wird `WinMain` anstatt `main()` genutzt, welches eigene Argumente hat:

```c
int CALLBACK WinMain(
    _In_  HINSTANCE hInstance,
    _In_  HINSTANCE hPrevInstance,
    _In_  LPSTR lpCmdLine,
    _In_  int nCmdShow
);
```

Der CRT-Code bereitet diese ebenfalls vor.

Die Zahl die von `main()` zurückgegeben wird ist der Exit-Code.

Diese kann in der CRT für die Funktion `ExitProcess()` werden, die diesen Exit-Code als Argument entgegennimmt.

In der Regel hat jeder Compiler seinen eigenen CRT-Code.

Nachfolgend eine typischer CRT-Code für MSVC 2008.

```
___tmainCRTStartup proc near

var_24 = dword ptr -24h
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
ms_exc = CPPEH_RECORD ptr -18h

    push 14h
    call __SEH_prolog4
    mov eax, 5A40h
    cmp ds:400000h, ax
    jnz short loc_401096
    mov eax, ds:40003Ch
    cmp dword ptr [eax+4000000h], 4550h
    jnz short loc_401096
    mov ecx, 10Bh
    cmp [eax+400018h], cx
    jnz short loc_401096
    cmp dword ptr [eax+400074h], 0Eh
    jnz short loc_401096
    jbe short loc_401096
    xor ecx, ecx
    cmp [eax+4000EBh], ecx
    setnz cl
    mov [ebp+var_1C], ecx
    jmp short loc_40109A

loc_401096: ; CODE XREF: ___tmainCRTStartup+18
    and [ebp+var_1C], 0

loc_40109A: ; CODE XREF: ___tmainCRTStartup+50
    push 1
    call __heap_init
    pop ecx
    test eax, eax
    jnz short loc_4010AE
    push 1Ch
    call __fast_error_exit
    pop ecx

loc_4010AE: ; CODE XREF: ___tmainCRTStartup+60
    call __mtinit
    test eax, eax
    jnz short loc_4010BF
    push 10h

```

\textsuperscript{14}Graphical User Interface
call _fast_error_exit
pop ecx

loc_4010BF: ; CODE XREF: ___tmainCRTStartup+71
    call sub_401F2B
    and [ebp+ms_exc.disabled], 0
    call __ioinit
    test eax, eax
    jge short loc_4010D9
    push 1Bh
    call __amsg_exit
    pop ecx

loc_4010D9: ; CODE XREF: ___tmainCRTStartup+8B
    call ds:GetCommandLineA
    mov dword_40B7F8, eax
    call __crtGetEnvironmentStringsA
    mov dword_40AC60, eax
    call __setargv
    test eax, eax
    jge short loc_4010FF
    push 8
    call __amsg_exit
    pop ecx

loc_4010FF: ; CODE XREF: ___tmainCRTStartup+B1
    call __setenvp
    test eax, eax
    jge short loc_401110
    push 9
    call __amsg_exit
    pop ecx

loc_401110: ; CODE XREF: ___tmainCRTStartup+C2
    push 1
    call __cinit
    pop ecx
    test eax, eax
    jz short loc_401123
    push eax
    call __amsg_exit
    pop ecx

loc_401123: ; CODE XREF: ___tmainCRTStartup+D6
    mov eax, envp
    mov dword_40AC80, eax
    push eax
    push argv
    push argc
    call _main
    add esp, 0Ch
    mov [ebp+var_28], eax
    cmp [ebp+var_1C], 0
    jnz short $LN28
    push eax
    ; uExitCode
    call $LN32

$LN28: ; CODE XREF: ___tmainCRTStartup+105
    call __exit
    jmp short loc_401186

$LN27: ; DATA XREF: .rdata:stru_4092D0
    mov eax, [ebp+ms_exc.exc_ptr]; Exception filter 0 for function 401044
    mov ecx, [eax]
    mov ecx, [ecx]
    mov [ebp+var_24], ecx
    push eax
    push ecx
    call __XcptFilter
Wir sehen hier die Aufrufe zu GetCommandLineA() (Zeile 62), anschließend zu setargv() (Zeile 66) und setenvp() (Zeile 74), was offensichtlich die globalen Variablen argc, argv und envp initialisiert.

Zum Schluss wird main() mit diesen Argumenten aufgerufen (Zeile 97).

Es sind ebenfalls Aufrufe zu Funktionen mit selbsterklärenden Namen zu finden, wie heap_init() (Zeile 35) und ioinit() (Zeile 54).

Der heap wird jedoch vom CRT initialisiert. Wenn man versucht malloc() in einem Programm ohne CRT zu nutzen, wird dieses mit dem folgenden Fehler abstürzen:

```
runtime error R6030
- CRT not initialized
```

Die Initialisierung von globalen Objekten in C++ passiert ebenfalls in der CRT vor der Ausführung von main(): ?? on page ??.

Ist es möglich die CRT loszuwerden? Ja, wenn man genau weiß, was man tut.

Der Linker von MSVC hat die /ENTRY-Option um den Einsprungpunkt festzulegen.

```c
#include <windows.h>

int main()
{
    MessageBox(NULL, "hello, world", "caption", MB_OK);
}
```

Kompilieren wir dies in MSVC 2008.

```
cl no_crt.c user32.lib /link /entry:main
```

Wir bekommen eine lauffähige .exe-Datei mit der Größe 2560 Byte mit einem PE-Header, Anweisungen die MessageBox aufrufen, zwei Zeichenketten im Datensegment, die aus user32.dll importierte Funktion MessageBox und sonst nichts.

Dies funktioniert, jedoch kann nicht WinMain mit den 4 Argumenten anstatt main() genutzt werden.

Um genau zu sein, wäre dies zwar möglich, allerdings wurden die Argumente nicht vorbereitet um sie zu nutzen.
Es ist übrigens auch möglich die .exe-Datei kleiner machen indem die PE-Sektion an weniger als den standardmäßigen 4096 Byte auszurichten.

```
cl no_crt.c user32.lib /link /entry:main /align:16
```

Der Linker gibt aus:

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image may not run
```

Die Ausgabe ist eine .exe-Datei mit 720 Byte. Sie kann unter Windows 7 x86, jedoch nicht x64 ausgeführt werden (beim Versuch wird eine Fehlermeldung erscheinen).

Mit mehr Aufwand ist es möglich die Datei noch weiter zu verkleinern, aber wie zu sehen ist, bekommt man schnell Kompatibilitätsprobleme.

### 6.5.2 Win32 PE

**PE** ist ein Dateiformat für ausführbare Dateien unter Windows. Der Unterschied zwischen .exe, .dll und .sys ist, dass .exe und .sys in der Regel nur Imports und keine Exports haben.

Eine **DLL** hat wie jede andere PE-Datei einen Eintrittspunkt (OEP) (die Funktion DllMain() befindet sich hier), allerdings macht diese Funktion in der Regel nichts. .sys ist normalerweise ein Gerätetreiber. Wie auch bei Treibern, erwartet Windows eine Prüfsumme in der PE-Datei, die korrekt sein muss.

Ab Windows Vista, muss eine Treiberdatei auch mit einer digitalen Signatur versehen sein. Anderseits wird das Laden des Treibers fehlgeschlagen.

Jede PE-Datei beginnt mit einem kleinen DOS-Programm, welches eine Nachricht in der Art wie folgt ausgibt: „Dieses Programm kann nicht im MS-DOS-Modus gestartet werden.”—wenn versucht wird das Programm unter DOS oder Windows 3.1 zu starten (BS-E die das PE-Format nicht kennen).

**Terminologie**

- **Modul**: eine separate Datei, .exe oder .dll
- **Prozessspeicher**: der Speicher, mit dem der Prozess arbeitet. Jeder Prozess hat seinen eigenen Bereich. Hier befinden sich in der Regel Module, der Stack, German text placeholder(s) usw.
- **VA**: Eine Adresse welche zur Laufzeit in einem Programm verwendet wird.
- **Basisadresse (eines Moduls)**: die Adresse im Prozessspeicher an der das Modul geladen wird. Der BS-Lader kann diese ändern wenn die Basisadresse bereits von einem vorher geladenen Modul verwendet wird.
- **RVA**: die VA-Adresse minus der Basisadresse.
  
  Viele Adressen in PE-Dateien-Tabellen nutzen RVA-Adressen.
  
  
  **INT**: Ein Array von Namen zu importierenden Symbole.

---

15 Dynamic-Link Library  
16 [Hiew (7.1 on page 504)](Hiew_7.1_on_page_504) kann diese berechnen  
17 Virtual Address  
18 Relative Virtual Address  
19 Import Address Table  
21 Import Name Table  
Basisadresse
Das Problem ist, dass mehrere Modulprogrammierer DLL-Dateien für die Nutzung für andere vorbereiten können, es jedoch nicht möglich ist festzulegen welche Adressen für diese Module verwendet werden.

Das ist der Grund, warum in dem Fall wenn zwei für einen Prozess notwendige DLLs dieselbe Basisadresse haben, eine davon an die Basisadresse geladen wird und die andere an eine andere freie Stelle im Prozessspeicher. Jede virtuelle Adresse der zweiten DLL wird korrigiert.

Mit MSVC generiert der Linker oft .exe-Dateien mit der Basisadresse 0x400000\(^{23}\), und mit der Code-Sektion die bei 0x401000 beginnt. Dies bedeutet, dass die RVA des Beginns der Code-Sektion 0x1000 ist.

DLLs werden vom MSVC-Linker oft mit der Basisadresse 0x10000000\(^{24}\). Es gibt einen weiteren Grund warum Module an verschiedenen Basisadressen, in diesem Fall an zufälligen Adressen, geladen werden: ASLR\(^{25}\).

Ein Shellcode der auf einem kompromittierten System ausgeführt werden soll, muss Systemfunktionen aufrufen und dementsprechend deren Adressen kennen.

In älteren BS (in der German text placeholder-Reihe: bevor Windows Vista) wurden System-DLL (wie kernel32.dll, user32.dll) immer an bekannte Adressen geladen und mit dem Wissen, dass deren Versionen selten wechseln, waren die Adressen der Funktionen fest und konnten direkt aufgerufen werden.

Um dies zu verhindern lädt ASLR das Programm und alle benötigten Module jedes Mal an zufällige Basisadressen. ASLR-Unterstützung ist in der PE-Datei mit dem Flag IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE markiert [siehe Mark Russinovich, Microsoft Windows Internals].

Subsystem
Es existiert auch ein Subsystem-Feld, dass normalerweise wie folgt ist:

- native\(^{26}\) (.sys-Treiber),
- console (Konsolenanwendung) oder
- GUI (Anwendung mit grafischer Oberfläche).

Betriebssystem-Version
Eine PE-Datei spezifiziert auch die minimale Windows-Version die sie benötigt um ladbar zu sein.


MSVC 2012 erzeugt standardmäßig .exe-Dateien mit der Version 6.00, die mindestens Windows Vista benötigen. Mit Änderungen an den Compiler-Option\(^{27}\) ist es jedoch möglich eine Kompilierung für Windows XP zu erzwingen.

Sektionen
Abschnitte in Sektionen sind in allen Formaten für ausführbare Dateien vorhanden.

Dies wurde entwickelt um Code von Daten und Daten von Konstanten zu trennen.

- In der Daten-Sektion sind die Flags IMAGE_SCN_CNT_INITIALIZED_DATA, IMAGE_SCN_MEM_READ und IMAGE_SCN_MEM_WRITE gesetzt.
- In einer leeren Sektion mit uninitialisierten Daten sind die Flags IMAGE_SCN_CNT_UNINITIALIZED_DATA, IMAGE_SCN_MEM_READ und IMAGE_SCN_MEM_WRITE gesetzt.

\(^{23}\)der Ursprung dieser Adress-Auswahl ist hier: MSDN
\(^{24}\)Dies kann mit der Linker-Option /BASE geändert werden
\(^{25}\)Address Space Layout Randomization
\(^{26}\)bedeutet, dass das Modul eine eigene API statt Win32 nutzt
\(^{27}\)MSDN
• In der Sektion mit konstanten Daten (die schreibgeschützt ist), sind die Flags `IMAGE_SCN_CNT_INITIALIZED_DATA` und `IMAGE_SCN_MEM_READ` gesetzt, nicht jedoch `IMAGE_SCN_MEM_WRITE`. Ein Prozess wird abstürzen, wenn er versucht hier schreibend zuzugreifen.

Jede Sektion in einer PE-Datei kann einen Namen haben, auch wenn dieser nicht unbedingt wichtig ist. Oft (aber nicht immer) ist die Code-Sektion mit `.text` benannt, die Datensektion mit `.data`, und die Sektion mit konstanten Daten `.rdata` für (readable data).

Andere verbreitete Sektionsnamen sind:

• `.idata`: Import-Sektion IDA kann eine Pseudosektion mit dem Namen 6.5.2 on page 471 erzeugen.
• `.edata`: Export-Sektion (selten)
• `.pdata`: Sektion, die alle Informationen über Ausnahmen in Windows NT für MIPS, IA64 28 und x64 enthält: 6.5.3 on page 498
• `.reloc`: Reloc-Sektion
• `.bss`: uninitialisierte Daten (BSS)
• `.tls`: Thread-lokaler Speicher (TLS)
• `.rsrcc`: Ressourcen
• `.CRT`: kann in Binärdateien vorhanden sein die mit alten MSVC-Compilern erzeugt wurden

Pack- und Verschlüsselungsprogramme für PE-Dateien verändern häufig die Sektionsnamen oder ersetzen sie durch eigene Namen.

MSVC ermöglicht es Daten in beliebigen benannte Sektion zu deklarieren 29.


Hier gibt es separate German text placeholder-Dateien für diesen Zweck.

Nachfolgend der Aufbau der PE-Sektion in dieser Datei:

```c
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Ein Wort zur Terminologie: `PointerToRawData` wird in Hiew „Offset“ und `VirtualAddress` „RVA“ genannt.

### Datensektion


IDA kann die Grenze der initialisierten und nicht initialisierten Teile wie folgt anzeigen:

---

28 Intel Architecture 64 (Itanium)
29 MSDN
30 MSDN
Relocations (relocs)

AKA FIXUPs (zumindest in Hiew). Diese sind ebenfalls in den meisten Formaten für ausführbare Dateien vorhanden\[^{31}\]. Ausnahmen sind gemeinsam genutzte (dynamische) Bibliotheken, die PIC enthalten

Wofür dienen die relocs?

Offensichtlich können Module an verschiedene Basisadressen geladen werden. Wie wird jedoch zum Beispiel mit globalen Variablen umgegangen? Auf diese muss anhand der Adresse zugegriffen werden. Eine Möglichkeit dazu ist positionsabhängiger Code (\[^{6.4.1\text{ on page 462}}\]), was aber nicht immer komfortabel ist

Aus diesem Grund existieren Relocation-Tabellen. Hier sind die Adressen die korrigiert werden müssen aufgelistet, falls an eine andere Basisadresse geladen wird.

Beispielsweise ist eine globale Variable an Adresse 0x410000. Das folgende Listing zeigt wie auf diese zugreifen wird:

```
A1 00 00 41 00 mov eax,[000410000]
```

Die Basisadresse des Moduls ist 0x400000, die RVA der globalen Variablen ist 0x10000.

Falls das Modul an die Basisadresse 0x500000 geladen wird, muss die reale Adresse der globalen Variablen auf 0x510000 geändert werden.

Wie man sieht ist die Adresse der Variablen in der Anweisung MOV, nach dem 0xA1 kodiert.

Aus diesem Grund wird die Adresse der vier Byte nach 0xA1 in die Relocation-Tabelle geschrieben.

Wenn das Modul an einer anderen Basisadresse geladen wird, listet der BS-Lader alle Adressen in der Tabelle auf, findet jedes 32-Bit-Word auf das die Adresse zeigt, subtrahiert die Basisadresse davon (das ergibt hier die RVA) und addiert die neue Basisadresse hinzu.

Wird das Modul an der originalen Basisadresse geladen passiert nichts.

Alle globalen Variablen können auf diese Weise behandelt werden.

Relocs können verschiedene Typen haben. In Windows für x86-Prozessoren ist dieser üblicherweise \texttt{IMAGE\_REL\_BASED\_HIGHLOW}.

Übrigens sind Relocs in Hiew abgedunkelt, wie hier zu sehen: Abb. 1.22.

OllyDbg unterstreicht die Orte im Speicher auf die Relocs angewendet wurden, beispielsweise: Abb. 1.53.

Exports und Imports

Wie bereits bekannt ist, muss jedes ausführbare Programm in irgendeiner Weise die Dienste des BS oder anderer DLL-Bibliotheken nutzen.

Die Funktionen eines Moduls (in der Regel eine DLL) muss irgendwie mit den Aufrufen in anderen Modulen (.exe-Dateien oder eine andere DLL) verbunden werden.

Aus diesem Grund hat jede DLL eine „Export“-Tabelle die aus Funktionen und deren Adressen in einem Modul besteht.

\[^{31}\text{Sogar in .exe-Dateien für MS-DOS.}\]
Außerdem hat jede .exe-Datei oder DLL „Imports“, eine Tabelle von Funktionen und Liste von DLL-Dateinamen, die sie für die Ausführung benötigt.

Nach dem laden der Haupt-exe-Datei verarbeitet der BS-Lader die Import-Tabelle: er lädt die zusätzlichen DLL-Dateien, findet die Funktionsnamen in den DLL-Exports und schreibt deren Adressen in die IAT der .exe-Datei.


Auf diese Weise können sie beim Laden einer DLL schneller gefunden werden.

Die „ordinals“ also Zahlworte, sind in der Export-Tabelle immer vorhanden.

Auf diese Weise laden Programme welche die MFC-Bibliothek nutzen die mfc*.dll und es existieren keine MFC-Namen wie in INT.

Wenn solche Programme in IDA geladen werden, wird nach dem Pfad zu den mfc*.dll-Dateien gefragt um die Funktionsnamen herausfinden zu können.

Wenn der Pfad zu diesen DLLs in IDA nicht angegeben wird, erscheint mfc80_123 statt der Funktionsnamen.

Import-Sektion

Häufig wird eine separate Sektion mit dem Namen .idata für die Import-Tabelle und alle dafür relevanten Dinge angelegt. Dies ist aber keine strikte Regel.

Imports sind manchmal etwas verwirrend wegen der uneinheitlichen Terminologie. Versuchen wir alle Informationen an einer Stelle zu sammeln.

32Microsoft Foundation Classes
Abbildung 6.1: Ein Schema dass alle PE-Datei-Strukturen im Zusammenhang mit Imports vereint


Während des Ladens und falls es möglich ist die Funktion anhand der Zahl zu finden, wird der Vergleich der Zeichenketten nicht auftauchen. Das Array ist mit Null terminiert.

Es gibt auch einen Zeiger zur IAT-Tabelle mit dem Namen `FirstThunk`. Dies entspricht der RVA-Adresse der Stelle an der der Lader die Adressen der aufgelösten Funktionen schreibt.

Die Punkte an denen der Lader die Adressen schreibt werden in IDA mit `__imp_CreateFileA` und so weiter gekennzeichnet.

Es gibt zumindest zwei Arten die vom Lader geschriebenen Adressen zu nutzen.
Der Code enthält eine Anweisung wie `call __imp_CreateFileA`. Da das Feld mit der Adresse der importierten Funktion in gewisser Weise eine globale Variable ist, wird die Adresse der `call`-Anweisung (plus 1 oder 2) zur Relocation-Tabelle hinzugefügt. Dies gilt für den Fall falls das Modul an eine andere Basisadresse geladen wird.

Aber offensichtlich kann dies die Relocation-Tabelle erheblich vergrößern, da möglicherweise in dem Modul viele Aufrufe von importierten Funktionen enthalten sind.

Des weiteren verlangsamen die großen Tabellen das Laden der Module.

Für jede importierte Funktion wird nur ein Sprung mittels der `JMP`-Anweisung und einem Reloc darauf alloziert. Solche Punkte werden auch „thunk“s genannt.

Alle Aufrufe zu den importierten Funktionen sind lediglich `CALL`-Anweisungen auf die entsprechenden „thunks“. In diesem Fall sind keine zusätzlichen Relocs notwendig, da diese CALLs eine relative Adresse haben und nicht korrigiert werden müssen.

Diese beiden Methoden können miteinander kombiniert werden.

Der Linker kann mehrere einzelne „thunk“s erstellen, falls zu viele Aufrufe der Funktion vorliegen. Dies ist jedoch nicht das Standardverhalten.

Übrigens muss das Array der Funktionsadressen auf das FirstThunk zeigt nicht unbedingt in der IAT-Sektion sein. Beispielsweise hat der Autor dieser Zeilen einmal das PE_add_import\(^{33}\)-Tool geschrieben um Imports zu einer existierenden .exe-Datei hinzufügen zu können.

In einer früheren Version des Tools hat dieses an die Stelle der Funktion an der ein Aufruf zu einer anderen DLL geschrieben werden sollte, folgenden Code erzeugt:

```assembly
MOV EAX, [yourdll.dll!function]
JMP EAX
```

FirstThunk zeigt auf die erste Anweisung. Mit anderen Worten, wenn yourdll.dll geladen wird, schreibt der Lader die Adresse der Funktion `function` direkt in den Code.


Man mag sich fragen: was ist wenn ein Programm mit ein paar DLLs kommt, die sich nicht ändern (inklusive der Adressen aller DLL-Funktionen). Ist es möglich den Lade-Vorgang zu beschleunigen?

Dies ist in der Tat möglich, wenn die Adressen der Funktionen bereits im Voraus in das FirstThunk-Array geschrieben werden. Das `Timestamp`-Feld ist in der `IMAGE_IMPORT_DESCRIPTOR`-Struktur vorhanden.

Wenn der Wert dort verfügbar ist, vergleicht der Lader diesen Wert mit dem Zeitstempel der DLL-Datei.

Wenn der Wert gleich ist, macht der Lader nichts und der Vorgang kann schneller sein. Dies wird „old-style binding“\(^{34}\) genannt.

Das Tool `BIND.EXE` ist für diesen Vorgang gedacht.

Um das Laden eigener Programme zu beschleunigen empfiehlt Matt Pietrek in Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)\(^{35}\) das Binden kurz nach der Installation des Programms auf dem Rechner des Endanwenders durchzuführen.

Komprimierungs- und Verschlüsselungsprogramme für PE-Dateien komprimieren bzw. Verschlüsseln auch die Import-Tabellen.

In diesem Fall wird der Windows-Lader natürlich nicht alle notwendigen DLL-Dateien lasen.


---

\(^{33}\) [yurichev.com](http://go.yurichev.com/17318)

\(^{34}\) MSDN. Dort auch „new-style binding“.

\(^{35}\) German text placeholder [http://go.yurichev.com/17318](http://go.yurichev.com/17318)
Während die .exe-Datei geladen wird, befindet sich diese nicht als Ganzes im Speicher (man denke an riesige Installationsprogramme welche verdächtig schnell geladen werden), sondern ist „gemapped“ und wird in Teilen geladen, wenn auf diese zugegriffen wird.

Vielleicht waren die Microsoft-Entwickler der Meinung, dass dies schneller ist.

Ressourcen

Ressourcen in einer PE-Datei sind lediglich Sammlungen von Icons, Bildern, Zeichenketten und Dialog-Beschreibungen.

Möglicherweise wurden Sie vom Hauptcode getrennt um mehrere Sprachen unterstützen zu können und es einfacher ist einen Text oder ein Bild in der Sprache auszuwählen, die zur Zeit im BS eingestellt ist.

Ein Seiten effekt ist, dass diese einfach editiert und in der ausführbaren Datei zurück gespeichert werden können. Mit speziellen Editoren wie beispielsweise (6.5.2) ist dies auch ohne spezielles Wissen möglich.

.NET

.NET-Programme werden nicht in Maschinencode übersetzt sondern in einen speziellen Bytecode. Streng genommen befindet sich anstatt des gewöhnlichen x86-Code der Bytecode in der .exe-Datei. Der Einsprungpunkt (OEP) jedoch zeigt auf ein kleines Fragment x86-Code:

```
jmp mscorere.dll!_CorExeMain
```

Der .NET-Lader befindet sich in mscorere.dll, welche die PE-Datei verarbeitet.
Dies war in allen Windows-Versionen vor Windows XP der Fall. Seit Windows XP ist der BS-Lader in der Lage .NET-Dateien zu erkennen und diese ohne eine JMP-Anweisung auszuführen

TLS


Abgesehen davon beinhaltet die Spezifikation für PE-Dateien auch die Möglichkeit der Initialisierung der TLS-Sektion, sogenannte TLS-Callbacks.

Falls diese vorhanden sind, werden sie aufgerufen bevor die Ausführungskontrolle an den Haupteinsprungpunkt (OEP) übergeben wird.

Dies ist sehr verbreitet bei Packern und Verschlüsselungsprogrammen für PE-Dateien.

Tools

- objdump (in Cygwin enthalten) um alle PE-Dateistrukturen auszugeben.
- Hiew (7.1 on page 504) als Editor.
- pefile: Python-Bibliothek für die Verarbeitung von PE-Dateien
- ResHack AKA Resource Hacker: Ressourcen-Editor
- PE_add_import: einfaches Tool um Symbole zur PE Importtabelle hinzuzufügen.
- PE_patcher: einfaches Tool um ausführbare PE-Dateien zu patchen.
- PE_search_str_refs: einfaches Tool zum Suchen von Funktionen in ausführbaren PE-Dateien die bestimmte Zeichenketten nutzen.

---

36 MSDN
37 http://go.yurichev.com/17052
38 http://go.yurichev.com/17052
39 http://go.yurichev.com/17049
40 yurichev.com
41 yurichev.com
6.5.3 Windows SEH

Vergessen wir MSVC


Wir betrachten SEH hier in einer isolierten Form und nicht im Zusammenhang mit C++ oder MSVC-Erweiterungen.

Jeder laufende Prozess hat eine Kette von SEH-Handles. TIB beinhaltet die Adresse des letzten Handlers.

Wenn eine Ausnahme auftritt (Division durch Null, Zugriff auf fehlerhafte Adresse, Benutzer-Ausnahme durch Aufruf RaiseException()-Funktion), findet das BS den letzten Handler in der TIB und ruft ihn auf. Dabei werden alle Informationen über den Zustand der CPU (Register-Werte usw.) im Moment der Ausnahme übergeben.

Wenn der Ausnahme-Handler eine bekannte Ausnahme sieht wird sie von ihm behandelt. Ist dies nicht der Fall, wird das BS darüber informiert, dass keine Ausnahmebehandlung stattfand und das BS ruft den nächsten Handler in der Kette, bis ein Handler gefunden wird, der die Ausnahme behandeln kann.


Abbildung 6.2: Windows XP

42http://go.yurichev.com/17056
43Structured Exception Handling
44Objektorientierte Programmierung
Abbildung 6.3: Windows XP

Abbildung 6.4: Windows 7
Abbildung 6.5: Windows 8.1

Früher wurde dieser Handler Dr. Watson genannt.

Einige Entwickler erstellen ihren eigenen Handler, der Informationen über den Absturz des Programms zu ihnen selbst schickt. Dieser wird mit der Funktion SetUnhandledExceptionFilter() registriert und aufgerufen, wenn das BS keine andere Möglichkeit hat die Ausnahme zu behandeln. Ein Beispiel ist Oracle RDBMS die eine riesige Menge an möglichen Informationen über die CPU und den Zustand des Speichers sammelt.


```c
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
    unsigned i;

    printf ("%s\n", __FUNCTION__);
    printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRecord->ExceptionCode);
    printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionRecord->ExceptionFlags);
    printf ("ExceptionRecord->ExceptionAddress=0x%p\n", ExceptionRecord->ExceptionAddress);

    if (ExceptionRecord->ExceptionCode==0xE1223344)
    {
        printf ("That's for us\n");
        // yes, we "handled" the exception
        return ExceptionContinueExecution;
    }
    else if (ExceptionRecord->ExceptionCode==EXCEPTION_ACCESS_VIOLATION)
    {
        printf ("ContextRecord->Eax=0x%08X\n", ContextRecord->Eax);
        // will it be possible to 'fix' it?
        printf ("Trying to fix wrong pointer address\n");
        ContextRecord->Eax=(DWORD)&new_value;
        // yes, we "handled" the exception
        return ExceptionContinueExecution;
    }
    else
    {
        printf ("We do not handle this\n");
        // someone else's problem
    }
```

45 German text placeholder [http://go.yurichev.com/17293](http://go.yurichev.com/17293)
return ExceptionContinueSearch;
}

int main()
{
    DWORD handler = (DWORD)except_handler; // take a pointer to our handler

    // install exception handler
    __asm {
        push handler           // make EXCEPTION_REGISTRATION record:
        push FS:[0]           // address of handler function
        mov FS:[0], ESP       // address of previous handler
        // add new EXCEPTION_REGISTRATION
    }

    RaiseException (0xE1223344, 0, 0, NULL);

    // now do something very bad
    int* ptr=NULL;
    int val=0;
    val=*ptr;
    printf ("val=%d\n", val);

    // deinstall exception handler
    __asm {
        mov eax,[ESP]             // remove our EXCEPTION_REGISTRATION record
        mov FS:[0], EAX           // get pointer to previous record
        add esp, 8                // install previous record
        // clean our EXCEPTION_REGISTRATION off stack
    }

    return 0;
}

Das FS:Segment-Register zeigt unter Win32 auf die TIB.


Listing 6.22: MSVC/VC/crt/src/exsup.inc

_EXCEPTION_REGISTRATION struc
    prev dd ?
    handler dd ?
_EXCEPTION_REGISTRATION ends

Jedes „handler“-Feld zeigt auf einen Handler und jedes „prev“-Feld zeigt auf den vorherigen Eintrag auf dem Stack. Der letzte Eintrag hat 0xFFFFFFFF (-1) im „prev“-Feld.

Was ist der Unterschied zwischen einer System-Ausnahme und einer User-Ausnahme? Hier sind die Ausnahmen des Systems:

<table>
<thead>
<tr>
<th>win in WinBase.h definiert</th>
<th>wie in ntstatus.h definiert</th>
<th>Wert</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCEPTION_ACCESS_VIOLATION</td>
<td>STATUS_ACCESS_VIOLATION</td>
<td>0xC0000005</td>
</tr>
<tr>
<td>EXCEPTION_DATATYPE_MISALIGNMENT</td>
<td>STATUS_DATATYPE_MISALIGNMENT</td>
<td>0x80000002</td>
</tr>
<tr>
<td>EXCEPTION_BREAKPOINT</td>
<td>STATUS_BREAKPOINT</td>
<td>0x80000003</td>
</tr>
<tr>
<td>EXCEPTION_SINGLE_STEP</td>
<td>STATUS_SINGLE_STEP</td>
<td>0x80000004</td>
</tr>
<tr>
<td>EXCEPTION_ARRAY_BOUNDS_EXCEEDED</td>
<td>STATUS_ARRAY_BOUNDS_EXCEEDED</td>
<td>0xC0000008C</td>
</tr>
<tr>
<td>EXCEPTION_FLT_DENORMAL_OPERAND</td>
<td>STATUS_FLOAT_DENORMAL_OPERAND</td>
<td>0xC0000008D</td>
</tr>
<tr>
<td>EXCEPTION_FLT_DIVIDE_BY_ZERO</td>
<td>STATUS_FLOAT_DIVIDE_BY_ZERO</td>
<td>0xC0000008E</td>
</tr>
<tr>
<td>EXCEPTION_FLT_INEXACT_RESULT</td>
<td>STATUS_FLOAT_INEXACT_RESULT</td>
<td>0xC0000008F</td>
</tr>
<tr>
<td>EXCEPTION_FLT_INVALID_OPERATION</td>
<td>STATUS_FLOAT_INVALID_OPERATION</td>
<td>0xC00000090</td>
</tr>
<tr>
<td>EXCEPTION_FLT_OVERFLOW</td>
<td>STATUS_FLOAT_OVERFLOW</td>
<td>0xC00000091</td>
</tr>
<tr>
<td>EXCEPTION_FLT_STACK_CHECK</td>
<td>STATUS_FLOAT_STACK_CHECK</td>
<td>0xC00000092</td>
</tr>
<tr>
<td>EXCEPTION_FLT_UNDERFLOW</td>
<td>STATUS_FLOAT_UNDERFLOW</td>
<td>0xC00000093</td>
</tr>
<tr>
<td>EXCEPTION_INT_DIVIDE_BY_ZERO</td>
<td>STATUS_INTEGER_DIVIDE_BY_ZERO</td>
<td>0xC00000094</td>
</tr>
<tr>
<td>EXCEPTION_INT_OVERFLOW</td>
<td>STATUS_INTEGER_OVERFLOW</td>
<td>0xC00000095</td>
</tr>
<tr>
<td>EXCEPTION_PRIV_INSTRUCTION</td>
<td>STATUS_PRIVILEGED_INSTRUCTION</td>
<td>0xC00000096</td>
</tr>
<tr>
<td>EXCEPTION_IN_PAGE_ERROR</td>
<td>STATUS_IN_PAGE_ERROR</td>
<td>0xC00000006</td>
</tr>
<tr>
<td>EXCEPTION_ILLEGAL_INSTRUCTION</td>
<td>STATUS_ILLEGAL_INSTRUCTION</td>
<td>0xC0000001D</td>
</tr>
<tr>
<td>EXCEPTION_NONCONTINUABLE_EXCEPTION</td>
<td>STATUS_NONCONTINUABLE_EXCEPTION</td>
<td>0xC00000025</td>
</tr>
<tr>
<td>EXCEPTION_STACK_OVERFLOW</td>
<td>STATUS_STACK_OVERFLOW</td>
<td>0xC000000FD</td>
</tr>
<tr>
<td>EXCEPTION_INVALID_DISPOSITION</td>
<td>STATUS_INVALID_DISPOSITION</td>
<td>0xC00000026</td>
</tr>
<tr>
<td>EXCEPTION_GUARD_PAGE</td>
<td>STATUS_GUARD_PAGE_VIOLATION</td>
<td>0x80000001</td>
</tr>
<tr>
<td>EXCEPTION_INVALID_HANDLE</td>
<td>STATUS_INVALID_HANDLE</td>
<td>0xC00000008</td>
</tr>
<tr>
<td>EXCEPTION_POSSIBLE_DEADLOCK</td>
<td>STATUS_POSSIBLE_DEADLOCK</td>
<td>0xC000000194</td>
</tr>
<tr>
<td>CONTROL_C_EXIT</td>
<td>STATUS_CONTROL_C_EXIT</td>
<td>0xC0000013A</td>
</tr>
</tbody>
</table>

Nachfolgend wie der Code aufgebaut ist:

```
<table>
<thead>
<tr>
<th>31</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>U</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*S* ist ein einfacher Status-Code: 11—Fehler; 10—Warnung; 01—Information; 00—Erfolgreich. *U*—Kennzeichnet ob es sich um User-Code handelt.

---

46 MSDN
Dies erklärt, warum wir oben den Wert 0xE12344 gewählt haben—E16 (11102) 0xE (1110b) bedeutet, dass es sich um eine User-Ausnahme handelt; 2) ein Fehler vorliegt.

Genau genommen funktioniert dieses Beispiel jedoch auch gut ohne diese höherwertigen Bits.

Anschließend versuchen wir einen Wert von der Speicheradresse 0 zu lesen.

Natürlich befindet sich hier nichts unter Win32, womit eine Ausnahme geworfen wird.

Der allererste Handler wird aufgerufen (der von oben) und prüft ob der Code der Konstante EXCEPTION_ACCESS_VIOLATION entspricht.

Der Code der von der Adresse an der Speicherstelle 0 liest, sieht wie folgt aus:

```
Listing 6.23: MSVC 2010
...
xor eax, eax
mov eax, DWORD PTR [eax] ; exception will occur here
push eax
push OFFSET msg
call _printf
add esp, 8
...
```

Ist es möglich diese Fehler „on the fly“ zu beheben und mit der Programmausführung fortfahren?

In der Tat ist dies möglich, da unser Ausnahme-Handler den EAX-Wert beheben und das BS diese Anweisung ein weiteres Mal ausführen kann. Das ist was wir tun. printf() gibt 1234 aus, weil EAX nach der Ausnahmebehandlung nicht mehr 0 ist sondern die Adresse der globalen Variable new_value enthält. Die Ausführung wird fortgesetzt.

Das ist was passiert: die Speicherverwaltung in der CPU signalisiert einen Fehler und die CPU stoppt den Thread, findet die passende Ausnahmebehandlung im Windows-Kernel welche wiederum nacheinander alle Handler in der SEH-Kette aufruft.

Hier wird MSVC 2010 genutzt, aber es gibt natürlich keine Garantie, dass EAX für diesen Zeiger genutzt wird.


Warum sind die SEH-Einträge direkt auf dem Stack gespeichert und nicht irgendwo anders?

Vermutlich ist der Grund, weil das BS sich dann nicht um das Freigeben der Informationen kümmern muss. Diese Einträge werden nach dem Ende der Funktion automatisch gesäubert. Dies entspricht in gewisser Weise alloca(): (1.6.3 on page 34).

Zurück zu MSVC

Offensichtlich benötigten die Microsoft-Entwickler Ausnahmen in C aber nicht in C++ und führten eine nicht-standardisierte C-Erweiterung ein47. Diese hat aber keinen Zusammenhang zu C++ PS-Ausnahmen.

```c
__try
{
    ...
}
__except(filter code)
{
    handler code
}
```

Der „Finally“-Block kann anstelle des Handler-Codes stehen:

```c
__try
{
    ...

47MSDN
```

Im Windows-Kernel existieren eine Reihe solcher Konstrukte. Nachfolgend einige Beispiel von dort (WRK):

Listing 6.24: WRK-v1.2/base/ntos/ob/obwait.c

```c
try {
    KeReleaseMutant((PKMUTANT)SignalObject,
        MUTANT_INCREMENT,
        FALSE,
        TRUE);
} except((GetExceptionCode () == STATUS_ABANDONED ||
        GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
            EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();
    goto WaitExit;
}
```

Listing 6.25: WRK-v1.2/base/ntos/cache/cachesub.c

```c
try {
    RtlCopyBytes((PVOID)((PCHAR)CacheBuffer + PageOffset),
        UserBuffer,
        MorePages ?
            (PAGE_SIZE - PageOffset) :
            (ReceivedLength - PageOffset) );
} except( CcCopyReadExceptionFilter( GetExceptionInformation(),
               &Status ) ) { 

Hier ist ein Filter-Code-Beispiel:

Listing 6.26: WRK-v1.2/base/ntos/cache/copysup.c

```c
LONG CcCopyReadExceptionFilter(
      IN PEXCEPTION_POINTERS ExceptionPointer,
      IN PNTSTATUS ExceptionCode)

/**+
Routine Description:
This routine serves as an exception filter and has the special job of
extracting the "real" I/O error when Mm raises STATUS_IN_PAGE_ERROR
beneath us.

Arguments:
    ExceptionPointer - A pointer to the exception record that contains
        the real Io Status.
    ExceptionCode - A pointer to an NTSTATUS that is to receive the real
        status.

Return Value:
```
# SEH

SEH ist eine Erweiterung der vom BS-unterstützten Ausnahmen, aber die Handler-Funktion ist _except_handler3 (für SEH3) oder _except_handler4 (für SEH4).


## SEH3

SEH3 hat _except_handler3 als Handler-Funktion und erweitert die _EXCEPTION_REGISTRATION-Tabelle indem ein Zeiger zur Scope-Tabelle und der previous try level-Variablen hinzugefügt wird. SEH4 erweitert die Scope-Tabelle um vier Werte für Schutz vor Speicherüberläufen.

Die Scope-Tabelle ist eine Tabelle die aus Zeigern auf Filter und Handler-Code-Blöcken für jede verschachtelte Ebene für try/except besteht.
Auch hier ist es wieder sehr wichtig zu verstehen, dass das BS sich lediglich um die prev/handle-Felder kümmert und sonst nichts.
Es ist Aufgaben der _except_handler3-Funktion die anderen Felder und die Scope-Tabelle zu lesen und zu entscheiden, welcher Handler wann aufgerufen werden muss.
Der Quellcode der _except_handler3-Funktion ist nicht offen.
Sanos OS, welches einen Win32-Kompatibilitäts-Layer hat, hat die gleiche Funktion implementiert, welche ähnlich ist zu der unter Windows. Eine weitere Implementierung existiert in Wine und ReactOS.
Wenn der Filter-Zeiger NULL ist, ist der Handler-Zeiger ein Zeiger auf den finally-Code-Block.
Während der Ausführung verändert sich der Wert des previous try level, so dass der _except_handler3 Information über den aktuellen Verschachtelungslevel hat, um zu wissen welcher Eintrag der Scope-Tabelle zu nutzen ist.

SEH3: one try/except block example

```c
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
    __try
    {
```
printf("hello #1!\n");  
*p = 13;  // causes an access violation exception; 
printf("hello #2!\n"); 
__except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ? 
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) 
{
    printf("access violation, can't recover\n"); 
}

Listing 6.27: MSVC 2003

$SG74605 DB  'hello #1!', 0AH, 00H
$SG74606 DB  'hello #2!', 0AH, 00H
$SG74608 DB  'access violation, can't recover', 0AH, 00H
_DATA ENDS

; scope table:
CONST SEGMENT
$T74622 DD  0fffffffH ; previous try level
    DD  FLAT:$L74617 ; filter
    DD  FLAT:$L74618 ; handler
_CONST ENDS

_TEXT SEGMENT
$L74621 = -32 ; size = 4
_p$ = -28  ; size = 4
__$SEHRec$ = -24 ; size = 24

_main PROC NEAR
  push    ebp
  mov     ebp, esp
  push   -1 ; previous try level
  push OFFSET FLAT:$T74622 ; scope table
  push OFFSET FLAT:__except_handler3 ; handler
  mov     eax, DWORD PTR fs:__except_list
  push    eax
  mov     DWORD PTR fs:__except_list, esp
  add    esp, -16
  ; 3 registers to be saved:
  push    ebx
  push    esi
  push    edi
  mov     DWORD PTR __$SEHRec$[ebp], esp
  mov     DWORD PTR _p$[ebp], 0
  mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
  push OFFSET FLAT:$SG74605 ; 'hello #1!' 
  call     __printf
  add      esp, 4
  mov     eax, DWORD PTR __$SEHRec$[ebp]
  mov     DWORD PTR [eax], 13
  push OFFSET FLAT:$SG74606 ; 'hello #2!' 
  call     __printf
  add      esp, 4
  mov     DWORD PTR __$SEHRec$[ebp+20], -1 ; previous try level
  jmp SHORT $L74616

; filter code:
$L74617:
$L74627:
  mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
  mov     edx, DWORD PTR [ecx]
  mov     eax, DWORD PTR [edx]
  mov     DWORD PTR $T74621[ebp], eax
  mov     eax, DWORD PTR $T74621[ebp]
  sub      eax, -1073741819; c0000005H
  neg      eax
  sbb     eax, eax
  inc      eax
$L74619:
Hier ist zu sehen wie der SEH-Frame auf dem Stack aufgebaut ist. Die *Scope-Tabelle* befindet sich im CONST-Segment, diese Felder werden nicht verändert. Eine interessante Sache ist es, wie die *previous try level*-Variable sich geändert hat. Der Wert zu Beginn ist 0xFFFFFFFF (−1). Der Moment, in dem der Body der *try*-Anweisung betreten wird, ist mit einer Anweisung gekennzeichnet, die 0 in die Variable schreibt. In dem Moment in dem der Body der *try*-Anweisung geschlossen wird, wird der Wert −1 dorthin zurückgeschrieben. Es sind ebenso die Adressen der Filter- und Handler-Codes zu sehen.

Wir können sehr einfach die Struktur des *try/except*-Konstrukts in der Funktion erkennen.

Da der SEH-Setup-Code im Funktionsprolog von mehreren Funktionen geteilt werden kann, fügt der Compiler manchmal einen Aufruf zur SEH_prolog()-Funktion in den Prolog ein, welcher genau dieses tut.

Der SEH-Aufräumcode ist in der SEH_epilog()-Funktion.

Versuchen wir dieses Beispiel in [German text placeholder] laufen zu lassen:

```plaintext
tracer.exe -l:2.exe --dump-seh
```

Listing 6.28: tracer.exe output

```
EXCEPTION ACCESS VIOLATION at 2.exe!main+0x44 (0x401054) ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x0008e3c8
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF

* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401070 (2.exe!main+0x60) handler=0x401088
\( \langle 2.exe!main+0x78 \rangle \)

* SEH frame at 0x18ff78 prev=0x18ff8c handler=0x401204 (2.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401531 (2.exe!mainCRTStartup+0x18d) \( \langle 2.exe!mainCRTStartup+0x1a1 \rangle \)

* SEH frame at 0x18ff8c prev=0x18ff4e handler=0x771f71f5 (ntdll.dll!__safe_se_handler_table+0x20)
SEH4 frame. previous trylevel=0
SEH4 header: GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
EHCookieOffset=0xfffffffc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!__safe_se_handler_table+0x20) \( \langle \text{ntdll.dll!TppTerminateProcess@4+0x43} \rangle \)

* SEH frame at 0x18ff4e prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16)
```

Es ist zu erkennen, dass die SEH-Kette aus vier Handlern besteht.

489

Der dritte ist SEH4 in ntdll.dll und der vierte Handler ist nicht MSVC-spezifisch, befindet sich in der ntdll.dll und hat einen selbsterklärenden Funktionsnamen.

Es ist zu erkennen, dass es drei Arten von Handlern in einer Kette gibt:


**SEH3: two try/except blocks example**

```c
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int filter_user_exceptions(unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
  printf("in filter. code=0x%08X\n", code);
  if (code == 0x112233)
  {
    printf("yes, that is our exception\n");
    return EXCEPTION_EXECUTE_HANDLER;
  }
  else
  {
    printf("not our exception\n");
    return EXCEPTION_CONTINUE_SEARCH;
  }
}

int main()
{
  int* p = NULL;
  try
  {
    try
    {
      printf ("hello!\n");
      RaiseException (0x112233, 0, 0, NULL);
      printf ("0x112233 raised. now let's crash\n");
      *p = 13;  // causes an access violation exception;
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
      EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
      printf("access violation, can't recover\n");
    }
  }
  __except(filter_user_exceptions(GetExceptionCode(), GetExceptionInformation()))
  {
    // the filter_user_exceptions() function answering to the question
    // "is this exception belongs to this block?"
    // if yes, do the follow:
    printf("user exception caught\n");
  }
}
```

Es existieren jetzt zwei try-Blöcke. Die `Scope-Table` hat jetzt zwei Einträge, einen für jeden Block. *Previous try level* verändert sich wenn die Ausführung einen try-Block betritt oder verlässt.
$SG74619 DB '0x112233 raised. now let's crash', 0AH, 00H
$SG74621 DB 'access violation, can't recover', 0AH, 00H
$SG74623 DB 'user exception caught', 0AH, 00H

_code$ = 8 ; size = 4
_ep$ = 12 ; size = 4

_filter_user_exceptions PROC NEAR
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _code$[ebp]
    push eax
    call OFFSET FLAT:$SG74606 ; 'in filter. code=0x%08X'
    call _printf
    cmp DWORD PTR _code$[ebp], 1122867 ; 00112233H
    jne SHORT $L74607
    push OFFSET FLAT:$SG74608 ; 'yes, that is our exception'
    call _printf
    add esp, 4
    mov eax, 1
    jmp SHORT $L74605
$L74607:
    push OFFSET FLAT:$SG74610 ; 'not our exception'
    call _printf
    add esp, 4
    xor eax, eax
$L74605:
    pop ebp
    ret 0
_filter_user_exceptions ENDP

; scope table:
CONST SEGMENT
$T74644 DD 0ffffffffH ; previous try level for outer block
    DD FLAT:$L74634 ; outer block filter
    DD FLAT:$L74635 ; outer block handler
    DD 00H ; previous try level for inner block
    DD FLAT:$L74638 ; inner block filter
    DD FLAT:$L74639 ; inner block handler
CONST ENDS

$T74643 = -36 ; size = 4
$T74642 = -32 ; size = 4
_p$ = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24

_main PROC NEAR
    push ebp
    mov ebp, esp
    push -1 ; previous try level
    push OFFSET FLAT:$T74644
    push OFFSET FLAT:_except_handler3
    mov eax, DWORD PTR fs:_except_list
    push eax
    mov DWORD PTR fs:_except_list, esp
    add esp, -20
    push ebx
    push esi
    push edi
    mov DWORD PTR _$SEHRec$[ebp], esp
    mov DWORD PTR _p$[ebp], 0
    mov DWORDPTR _$SEHRec$[ebp+20], 0 ; outer try block entered. set previous try level to 0
    mov DWORDPTR _$SEHRec$[ebp+20], 1 ; inner try block entered. set previous try level to 1
    push OFFSET FLAT:$SG74617 ; 'hello!'
    call _printf
    add esp, 4
    push 0
    push 0
    push 0
    push 1122867 ; 00112233H

491
Wenn ein Breakpoint auf die printf()-Funktion gesetzt wird, die vom Handler aufgerufen wird, ist auch sichtbar, wie ein neuer SEH-Handler hinzugefügt wird.

Möglichwerise ist innerhalb des SEH Handling-Prozesses noch eine andere Funktion. Es sind hier in der Scope-Tabelle zwei Einträge zu sehen.

tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh

Listing 6.30: tracer.exe output

(0) 3.exe!printf
EAX=0x0000001b EEBX=0x00000000 ECX=0x0040cc58 EDX=0x0000e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18f79c handler=0x771db4ad (ntdll.dll!ExecuteHandler2@20+0x3a)
* SEH frame at 0x18f79c prev=0x18f78 handler=0x4012e0 (3.exe!_except_handler3)
* SEH frame. previous try level=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.exe!main+0xb0) handler=0x40113b
* SEH frame. previous try level=0, filter=0x401100 (3.exe!main+0x78) handler=0x401100
* SEH frame. previous try level=0
* SEH frame at 0x18f78 prev=0x18f77 handler=0x4012e0 (3.exe!_except_handler3)
* SEH frame. previous try level=0
* SEH frame at 0x18f764 prev=0x18f75 handler=0x771f71f5 (ntdll.dll!__except_handler4)
* SEH frame. previous try level=0
(SEH4 header: GSCookieOffset=0xfffffffe GSCookieXORoffset=0x0
EHCookieOffset=0xffffffff EHCookieXOROffset=0x0
* SEH frame. previous try level=-2, filter=0x771f74d0 (ntdll.dll!__safe_se_handler_table+0x20) handler=0x771f90de ntdll.dll! TppTerminateProcess@4+0x43)
* SEH frame at 0x18f744 prev=0x18f74 handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16)

SEH4


Der initiale previous try level ist –2 in SEH4 anstatt –1.
Hier sind beide Beispiele mit MSVC 2012 und SEH4 kompiliert:

Listing 6.31: MSVC 2012: one try block example

```
$SG85485 DB 'hello #1!', 0AH, 00H
$SG85486 DB 'hello #2!', 0AH, 00H
$SG85488 DB 'access violation, can''t recover', 0AH, 00H

; scope table: xdata$x
_xdata$x SEGMENT
__sehtable$main DD 0FFFFFFFFH ; GS Cookie Offset
  DD 00H ; GS Cookie XOR Offset
__sehtable$main DD 0FFFFFFfCH ; EH Cookie Offset
  DD 00H ; EH Cookie XOR Offset
__sehtable$main DD 0FFFFFFFFfH ; previous try level
  DD FLAT:$LN12@main ; filter
  DD FLAT:$LN8@main ; handler
xdata$x ENDS
$T2 = -36 ; size = 4
_p$ = -32 ; size = 4
tv68 = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24
```
_main PROC
    push ebp
    mov ebp, esp
    push -2
    push OFFSET __sehtable$ _main
    push OFFSET __except_handler4
    mov eax, DWORD PTR fs:0
    push eax
    add esp, -20
    push ebx
    push esi
    push edi
    mov eax, DWORD PTR __security_cookie
    xor DWORD PTR __$SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor eax, ebp
    lea eax, DWORD PTR __$SEHRec$[ebp+8] ; pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov DWORD PTR fs:0, eax
    mov DWORD PTR __$SEHRec$[ebp], esp
    mov DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push OFFSET __SG85485 ; 'hello #1!'
    call _printf
    add esp, 4
    mov eax, DWORD PTR __$SEHRec$[ebp]
    mov DWORD PTR [eax], 13
    push OFFSET __SG85486 ; 'hello #2!'
    call _printf
    add esp, 4
    mov DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
    jmp SHORT $LN6@main

; filter:
$LN7@main:
$LN12@main:
    mov ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov edx, DWORD PTR [ecx]
    mov eax, DWORD PTR [edx]
    mov DWORD PTR $T2[ebp], eax
cmp DWORD PTR $T2[ebp], -1073741819 ; c0000005H
    jne SHORT $LN4@main
    mov DWORD PTR tv68[ebp], 1
    jmp SHORT $LN5@main
$LN4@main:
    mov DWORD PTR tv68[ebp], 0
$LN5@main:
    mov eax, DWORD PTR tv68[ebp]
$LN9@main:
    $LN11@main:
        ret 0

; handler:
$LN8@main:
    mov esp, DWORD PTR __$SEHRec$[ebp]
    push OFFSET __SG85488 ; 'access violation, can't recover'
    call _printf
    add esp, 4
    mov DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
$LN6@main:
    xor eax, eax
    mov DWORD PTR __$SEHRec$[ebp+8]
    mov DWORD PTR fs:0, ecx
    pop ecx
    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
Listing 6.32: MSVC 2012: two try blocks example

$xdatax$ SEGMENT
__sehtable$_main$ DD 0fffffffeH ; GS Cookie Offset
    DD 00H ; GS Cookie XOR Offset
    DD 0ffffffc0H ; EH Cookie Offset
    DD 00H ; EH Cookie Offset
    DD 0fffffffeH ; previous try level for outer block
    DD FLAT:$LN19@main ; outer block filter
    DD FLAT:$LN9@main ; outer block handler
    DD 00H ; previous try level for inner block
    DD FLAT:$LN18@main ; inner block filter
    DD FLAT:$LN13@main ; inner block handler
$xdatax$ ENDS

$T2$ = -40 ; size = 4
$T3$ = -36 ; size = 4
$p$ = -32 ; size = 4
tv72 = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24

_main$ PROC
    push ebp
    mov ebp, esp
    push -2 ; initial previous try level
    push OFFSET __sehtable$_main
    push OFFSET __except_handler4
    mov eax, DWORD PTR fs:0
    push eax ; prev
    add esp, -24
    push ebx
    push esi
    push edi
    mov eax, DWORD PTR __security_cookie
    xor DWORD_PTR __$SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor eax, ebp ; ebp ^ security_cookie
    push eax
    lea eax, DWORD PTR __$SEHRec$[ebp+8] ; pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov DWORD PTR fs:0, eax
    mov DWORD PTR __$SEHRec$[ebp], esp
    mov DWORD PTR p$[ebp], 0
    mov DWORD PTR __$SEHRec$[ebp+20], 0 ; entering outer try block, setting previous try
    level=0
    mov DWORD PTR __$SEHRec$[ebp+20], 1 ; entering inner try block, setting previous try
    level=1
    push OFFSET $SG85497 ; 'hello!'
call _printf
    add esp, 4
    push 0
    push 0
    push 0
    push 1122867 ; 00112233H
    call DWORD PTR __imp__RaiseException@16
    push OFFSET $SG85499 ; '0x112233 raised. now let''s crash'
call _printf
    add esp, 4
    mov eax, DWORD PTR p$[ebp]
    mov DWORD PTR [eax], 13
    mov DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, set previous try level
    back to 0
    jmp SHORT $LN2@main

; inner block filter:
$LN12@main:
$LN18@main:
mov ecx, DWORD PTR __SEHRec$[ebp+4]
mov edx, DWORD PTR [ecx]
mov eax, DWORD PTR [edx]
mov DWORD PTR $T3[ebp], eax
cmp DWORD PTR $T3[ebp], -1073741819 ; c000005H
jne SHORT $LN5@main
mov DWORD PTR tv72[ebp], 1
jmp SHORT $LN6@main
$LN5@main:
    mov DWORD PTR tv72[ebp], 0
$LN6@main:
    mov eax, DWORD PTR tv72[ebp]
$LN14@main:
    ret 0

; inner block handler:
$LN13@main:
    mov esp, DWORD PTR __SEHRec$[ebp]
push OFFSET $SG85501 ; 'access violation, can't recover'
call _printf
add esp, 4
mov DWORD PTR __SEHRec$[ebp+20], 0 ; exiting inner try block, setting previous try level back to 0
$LN2@main:
    mov DWORD PTR __SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level back to -2
jmp SHORT $LN7@main

; outer block filter:
$LN8@main:
$LN9@main:
    mov ecx, DWORD PTR __SEHRec$[ebp+4]
mov edx, DWORD PTR [ecx]
mov eax, DWORD PTR [edx]
mov DWORD PTR $T2[ebp], eax
mov ecx, DWORD PTR __SEHRec$[ebp+4]
push ecx
mov edx, DWORD PTR $T2[ebp]
push edx
call _filter_user_exceptions
add esp, 8
$LN10@main:
$LN17@main:
    ret 0

; outer block handler:
$LN9@main:
    mov esp, DWORD PTR __SEHRec$[ebp]
push OFFSET $SG85503 ; 'user exception caught'
call _printf
add esp, 4
mov DWORD PTR __SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level back to -2
$LN7@main:
xor eax, eax
mov ecx, DWORD PTR __SEHRec$[ebp+8]
mov DWORD PTR fs:0, ecx
pop ecx
pop edi
pop esi
pop ebx
mov esp, ebp
pop ebp
ret 0

_main ENDP

_code$ = 8 ; size = 4
_ep$ = 12 ; size = 4
_filter_user_exceptions PROC
    push ebp

Die Bedeutung des Cookies ist wie folgt: Der Cookie Offset ist die Differenz zwischen der Adresse des gespeicherten EBP-Wertes auf dem Stack und des EBP@security_cookie-Werts auf dem Stack. Der Cookie XOR Offset ist eine zusätzliche Differenz zwischen dem EBP@security_cookie-Wert und was auf dem Stack gespeichert ist.

Wenn diese Gleichung nicht richtig ist, wird der Prozess aufgrund eines korrupten Stack angehalten.

Die Cookie-Überprüfung ist auch in dem German text placeholder implementiert. Siehe GitHub für Details.

Es ist immer noch möglich, SEH3 im Compiler zu nutzen wenn eine neuere Version als MSVC 2005 genutzt wird, durch setzen der /GS--Option. Der CRT-Code nutzt SEH4 auf jeden Fall.

Windows x64

Wie man sich vielleicht denken kann, ist es nicht sehr schnell bei jedem Funktionsprolog einen SEH-Frame aufzubauen. Ein weiteres Geschwindigkeitsproblem ist das häufige Ändern des previous try level-Werts während der Ausführung einer Funktion.

Also haben sich die Dinge in x64 komplett geändert: alle Zeiger auf einen try-Block, Filter und Handler-Funktionen sind im einem PE-Segment .pdata gesichert. Von hier nimmt die BS-Ausnahmebehandlung alle Informationen.

Hier sind zwei Beispiele aus dem letzten Abschnitt, für x64 kompiliert:

Listing 6.33: MSVC 2012

```
SG86276 DB 'hello #1!', 0AH, 00H
SG86277 DB 'hello #2!', 0AH, 00H
SG86279 DB 'access violation, can't recover', 0AH, 00H
pdata SEGMENT
pdata$main DD imagerel $LN9
     DD imagerel $LN9+61
     DD imagerel $unwind$main
pdata ENDS
pdata SEGMENT
pdata$main$filt$0 DD imagerel main$filt$0
     DD imagerel main$filt$0+32
     DD imagerel $unwind$main$filt$0
pdata ENDS
xdata SEGMENT
$unwind$main DD 020609H
     DD 030023206H
     DD imagerel __C_specific_handler
```
$SG86277 DB 'in filter. code=0x%08X', 0AH, 00H
$SG86279 DB 'yes, that is our exception', 0AH, 00H
$SG86281 DB 'not our exception', 0AH, 00H
$SG86288 DB 'hello!!', 0AH, 00H
$SG86290 DB '0x112233 raised. now let''s crash', 0AH, 00H
$SG86292 DB 'access violation, can''t recover', 0AH, 00H
$SG86294 DB 'user exception caught', 0AH, 00H

pdata SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
DD imagerel $LN6+73
DD imagerel $unwind$filter_user_exceptions
$pdata$main DD imagerel $LN14
DD imagerel $LN14+95
DD imagerel $unwind$main
pdata ENDS

Listing 6.34: MSVC 2012
pdata SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
   DD imagerel main$filt$0+32
   DD imagerel $unwind$main$filt$0
$pdata$main$filt$1 DD imagerel main$filt$1
   DD imagerel main$filt$1+30
   DD imagerel $unwind$main$filt$1
pdata ENDS

xdata SEGMENT
$unwind$filter_user_exceptions DD 020601H
   DD 030023206H
$unwind$main DD 020609H
   DD 030023206H
   DD imagerel __C_specific_handler
   DD 02H
   DD imagerel $LN14+8
   DD imagerel $LN14+59
   DD imagerel main$filt$0
   DD imagerel $LN14+59
   DD imagerel $LN14+8
   DD imagerel $LN14+74
   DD imagerel main$filt$1
   DD imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
   DD 050023206H
$unwind$main$filt$1 DD 020601H
   DD 050023206H
xdata ENDS

_TEXT SEGMENT
main PROC
$LN14:
   push  rbx
   sub   rsp, 32
   xor   ebx, ebx
   lea   rcx, OFFSET FLAT:$SG86288 ; 'hello'
   call  printf
   xor   r9d, r9d
   xor   r8d, r8d
   xor   edx, edx
   mov   ecx, 1122867 ; 00112233H
   call  QWORD PTR __imp_RaiseException
   lea   rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised. now let''s crash'
   call  printf
   mov   DWORD PTR [rbx], 13
   jmp   SHORT $LN13$main
$LN11$main:
   lea   rcx, OFFSET FLAT:$SG86292 ; 'access violation, can''t recover'
   call  printf
   npad1 ; align next label
$LN13$main:
   jmp   SHORT $LN9$main
$LN7$main:
   lea   rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
   call  printf
   npad1 ; align next label
$LN9$main:
   xor   eax, eax
   add   esp, 32
   pop   rbx
   ret   0
main ENDP

text$x SEGMENT
main$filt$0 PROC
   push  rbp
   sub   rsp, 32
   mov   rbp, rdx
$LN10$main$filt$:

Neben den Ausnahme-Informationen, beinhaltet .pdata die Adressen von fast allen Funktionsbeginn- und enden, da dies für Tools die für automatische Analysen nützlich sein kann.

---

51German text placeholder http://go.yurichev.com/17294
6.5.4 Windows NT: Kritischer Abschnitt

Kritische Abschnitte sind in jedem BS sehr wichtig bei Multithread-Umgebungen. Der Zweck besteht darin, einen exklusiven Zugriff auf eine Ressource zu garantieren, während andere Threads oder Interrupts blockiert sind.

Nachfolgend, wie eine CRITICAL_SECTION-Struktur unter German text placeholder deklariert wird:

```
Listing 6.35: (Windows Research Kernel v1.2) public/sdk/inc/nturtl.h

typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;
    // // The following three fields control entering and exiting the critical
    // section for the resource
    //
    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread; // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount; // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

Nachfolgend wird gezeigt, wie die Funktion EnterCriticalSection() funktioniert:

```
Listing 6.36: Windows 2008/ntdll.dll/x86 (begin)

_RtlEnterCriticalSection@4

var_C = dword ptr -0Ch
var_B = dword ptr -8
var_A = dword ptr -4
arg_0 = dword ptr 8

    mov   edi, edi
    push  ebp
    mov   ebp, esp
    sub   esp, 0Ch
    push  esi
    push  edi
    mov   edi, [ebp+arg_0]
    lea   esi, [edi+4] ; LockCount
    mov   eax, esi
    lock btr dword ptr [eax], 0
    jnb   wait ; jump if CF=0

loc_7DE922DD:

    mov   eax, large fs:18h
    mov   ecx, [eax+24h]
    mov   [edi+0Ch], ecx
    mov   dword ptr [edi+8], 1
    pop   edi
    xor   eax, eax
    pop   esi
    mov   esp, ebp
    pop   ebp
    retn 4
```

[52] German text placeholder http://go.yurichev.com/17293
[53] German text placeholder http://go.yurichev.com/17294
Die wichtigste Funktion in diesem Code-Fragment ist BTR (nach dem vorangehenden LOCK):

Das nullte Bit wird im CF-Flag gesichert und im Speicher zurückgesetzt. Dies ist eine German text placeholder und blockiert alle Zugriffe der CPU auf diesen Teil des Speichers (siehe LOCK vor der BTR-Anweisung). Wenn das Bit in LockCount 1 ist, wird es zurückgesetzt und von der Funktion zurückgekehrt: die CPU befindet sich nun um Kritischen Abschnitt.

Wenn nicht, wurde der Kritische Abschnitt bereits von einem anderen Thread betreten, also muss gewartet werden. Das Warten wird durch die Funktion WaitForSingleObject() realisiert.

Hier nun, wie die Funktion LeaveCriticalSection() funktioniert:

Listing 6.37: Windows 2008/ntdll.dll/x86 (begin)

XADD bedeutet „exchange and add“.

In diesem Fall wird 1 zu LockCount addiert, während der ursprüngliche Wert von LockCount im EBX-Register gesichert wird. Der Wert in EBX wird durch aufeinander folgende INC EBX-Anweisungen inkrementiert und wird damit gleich dem aktualisierten Wert von LockCount.

Diese Operation ist atomar, da sie ebenfalls mit LOCK eingeleitet wird und so alle anderen CPUs oder CPU-Kerne des Systems für den Zugriff auf diesen Speicherbereich blockiert werden.

Das vorangehende LOCK ist sehr wichtig:

ohne diese Anweisung können zwei Threads die auf unterschiedlichen CPUs oder CPU-Kernen laufen, versuchen den Kritischen Abschnitt zu betreten und den Wert im Speicher zu verändern. Diese kann zu einem nicht-deterministischen Verhalten führen.
Kapitel 7

Tools

Now that Dennis Yurichev has made this book free (libre), it is a contribution to the world of free knowledge and free education. However, for our freedom’s sake, we need free (libre) reverse engineering tools to replace the proprietary tools described in this book.

Richard M. Stallman

7.1 Binäre Analyse

Tools die genutzt werden können, wenn kein Prozess gestartet wurde.

- (kostenlos, Open Source) ent^1: Entropie-Analyse-Tool. Mehr über Entropie: ?? on page ??.
- (kostenlos, Open Source) GHex^3: Einfacher Hex-Editor für Linux.
- (kostenlos, Open Source) xxd und od: Standard UNIX-Tools für Dumping.

7.1.1 Disassembler

- IDA. Eine ältere Freeware-Version ist online erhältlich^6. German text placeholder: .4.1 on page 544
- Binary Ninja^7
- (kostenlos, Open Source) zynamics BinNavi^8
- (kostenlos, Open Source) objdump: Einfaches Kommandozeilen-Tool für Dumping und zum disassemblieren.
- (kostenlos, Open Source) readelf^9: Gibt Informationen über ELF-Dateien aus.

---

^1 http://www.fourmilab.ch/random/
^2 hiew.ru
^3 https://wiki.gnome.org/Apps/Ghex
^5 http://binwalk.org/
^6 hex-rays.com/products/ida/support/download_freeware.shtml
^7 http://binary.ninja/
^8 https://www.zynamics.com/binnavi.html
^9 https://sourceware.org/binutils/docs/binutils/readelf.html
7.1.2 Decompiler
Es gibt lediglich einen bekannten, öffentlich verfügbaren Decompiler für C-Code in hoher Qualität: Hex-Rays:
hex-rays.com/products/decompiler/
Mehr darüber: ?? on page ??.

7.1.3 Vergleichen von Patches
Diese Tools können genutzt werden wenn die Original-Version einer ausführbaren Datei mit einer veränderten Version verglichen werden soll, oder um herauszufinden was verändert wurde und warum.

- (kostenlos) zynamics BinDiff
- (kostenlos, Open Source) Diaphora

7.2 Live-Analyse
Tools die im Live-System oder auf laufende Prozesse angewandt werden können.

7.2.1 Debugger

- (kostenlos) OllyDbg. Sehr populärer user-mode Debugger für die Win32-Architektur. German text placeholder: .4.2 on page 545
- (kostenlos, Open Source) GDB. Nicht sehr populärer Debugger unter Reverse Engineers, da eher für Programmierer gemacht. Einige Kommandos: .4.5 on page 545. Es gibt eine grafische Oberfläche für GDB, “GDB dashboard”.
- (kostenlos, Open Source) LLDB
- WinDbg: Kernel-Debugger für Windows.
- IDA hat einen internen Debugger.
- (kostenlos, Open Source) Radare AKA rada.re AKA r2. Es existiert auch eine GUI: ragui.
- (kostenlos, Open Source) tracer. Der Autor benutzt oft tracer anstatt Debugger.

Der Autor dieses Buchs hat irgendwann aufgehört Debugger zu nutzen, da alles was er von diesen brauchte, die Funktionsargumente während der Ausführung oder die Zustände der Register an einem bestimmten Punkt, waren. Jedes Mal den Debugger zu starten ist zu aufwändig, deswegen entstand das kleine Tool tracer. Es funktioniert in der Kommandozeile und erlaubt es Funktionsausführungen abzufangen, Breakpoints an beliebigen Stellen zu setzen und Register-Zustände zu lesen und ändern. tracer wird nicht weiterentwickelt, weil es als Demonstrationstool für dieses Buch entstand und nicht als Tool für den Alltag.

7.2.2 Tracenvon Bibliotheksaufrufen
ltrace

7.2.3 Tracenvon Systemaufrufe
strace / dtruss
Dies zeigt welche Systemaufrufe (syscalls( 6.3 on page 461)) vom aktuellen Prozess aufgerufen werden.

10https://www.zynamics.com/software.html
11https://github.com/joxeankoret/diaphora
12ollydbg.de
13https://github.com/cyrus-and/gdb-dashboard
14http://lldb.llvm.org/
16http://rada.re/r/
17http://radare.org/ragui/
18yurichev.com
19http://www.ltrace.org/
Zum Beispiel:

```
# strace df -h
...
access("/etc/ld.so.nohwcap", F_OK)       = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, ":ELF\1\1\1\0\0\0\0\0\0\0\0\0\0\0\0\0\0\1\0\0\0\0\0\0\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb75b3000
```

Mac OS X hat dtruss für den selben Verwendungszweck.

Cygwin beinhaltet ebenso strace, funktioniert aber soweit bekannt nur mit .exe-Dateien die für die Cygwin-Umgebung kompiliert wurden.

### 7.2.4 Netzwerk-Analyse (Sniffing)


Wireshark hat einen jüngeren (oder älteren) Bruder **tcpdump**[^22], bei dem es sich um ein simples Kommandozeilen-Tool handelt.

### 7.2.5 Sysinternals


### 7.2.6 Valgrind


### 7.2.7 Emulatoren

- (kostenlos, Open Source) **QEMU**[^24]: Emulator für verschiedene CPUs und Architekturen.
- (kostenlos, Open Source) **DosBox**[^25]: MS-DOS-Emulator, meist genutzt für Retro-Gaming.
- (kostenlos, Open Source) **SimH**[^26]: Emulator für ältere Computer, Mainframes, etc.

### 7.3 Andere Tools

**Microsoft Visual Studio Express**[^27]: Abgespeckte, freie Variante von Visual Studio, praktisch für einfache Experimente.

Einige nützliche Optionen: .4.3 on page 545.


[^20]: https://www.wireshark.org/
[^21]: https://wiki.wireshark.org/CaptureSetup/USB
[^22]: http://www.tcpdump.org/
[^24]: http://qemu.org
[^25]: https://www.dosbox.com/
[^26]: http://simh.trailing-edge.com/
[^27]: visualstudio.com/en-US/products/visual-studio-express-vs
7.3.1 Rechner
Gute Rechner für Reverse Engineering sollten zumindest Unterstützung für Dezimal, Hexadezimal und binär-Basen, sowie wichtige Operationen wie XOR oder Schiebeoperationen haben.

- IDA hat einen eingebauten Rechner ("?").
- rada.re hat rax2.
- https://yurichev.com/progcalc/

7.4 Fehlt etwas?
Wenn Sie ein gutes Tool kennen, was hier nicht aufgelistet ist, schreiben Sie mir:
<first_name @ last_name . com> / <first_name . last_name @ gmail . com>.
Kapitel 8

Beispiele für das Reverse Engineering proprietärer Dateiformate

8.1 German text placeholder

8.1.1 Einfachste XOR-Verschlüsselung überhaupt

Ich habe einmal eine Software gesehen, bei der alle Debugging-Ausgaben mit XOR mit dem Wert 3 verschlüsselt wurden. Mit anderen Worten, die beiden niedrigsten Bits aller Buchstaben wurden invertiert. “Hello, world” wurde zu “Kfool/#tlqog”:

```python
#!/usr/bin/python
msg="Hello, world!"
print "".join(map(lambda x: chr(ord(x)^3), msg))
```

Das ist eine ziemlich interessante Verschlüsselung (oder besser eine Verschleierung), weil sie zwei wichtige Eigenschaften hat: 1) es ist eine einzige Funktion zum Verschlüsseln und entschlüsseln, sie muss nur wiederholt angewendet werden 2) die entstehenden Buchstaben befinden sich im druckbaren Bereich, also die ganze Zeichenkette kann ohne Escape-Symbole im Code verwendet werden.

Die zweite Eigenschaft nutzt die Tatsache, dass alle druckbaren Zeichen in Reihen organisiert sind: 0x2x-0x7x, und wenn die beiden niedrigwertigsten Bits invertiert werden, wird der Buchstabe um eine oder drei Stellen nach links oder rechts verschoben, aber niemals in eine andere Reihe:

![7-Bit ASCII Table in Emacs](image)

Abbildung 8.1: 7-Bit ASCII Tabelle in Emacs

...mit dem Zeichen 0x7F als einziger Ausnahme.

Im Folgenden werden also beispielsweise die Zeichen A-Z verschlüsselt:
Es sieht so aus als würden die Zeichen “@” und “C” sowie “B” und “A” vertauscht werden.

8.2 German text placeholder

Pierre Capillon – Black-box cryptanalysis of home-made encryption algorithms: a practical case study.
How to Hack an Expensive Camera and Not Get Killed by Your Wife.
Kapitel 9

Dynamic Binary Instrumentation (DBI)

DBI-Werkzeuge können als sehr fortschrittliche und schnelle Debugger angesehen werden.
Kapitel 10

Weitere Themen

10.1 Patchen von ausführbaren Dateien

10.1.1 x86-Code

Häufige Aufgaben beim Patchen sind:

• Eine der häufigsten Aufgaben ist das Deaktivieren bestimmter Anweisungen. Oft wird dies durch Austauschen des Bytes durch 0x90 (**NOP**).

• Bedingte Sprünge, die den Opcode wie 74 xx (JZ) haben, können durch **NOP**s ersetzt werden. Es ist möglich alle bedingten Sprünge zu deaktivieren, in dem eine 0 in das zweite Byte geschrieben wird (**Sprung-Offset**).

• Eine weitere häufige Aufgabe ist es einen bedingten Sprung immer ausführen zu lassen: dies kann durch Schreiben von 0xEB, was für **JMP** steht, anstatt des Opcodes erreicht werden.

• Die Ausführung einer Funktion kann deaktiviert werden, wenn RETN (0xC3) an den Anfang geschrieben wird. Dies gilt für alle Funktionen außer stdcall (**6.1.2 on page 448**). Um stdcall-Funktionen zu patchen muss die Anzahl der Argumente bekannt sein (zum Beispiel durch Finden der RETN-Anweisung in der Funktion) und die RETN-Anweisung mit einem 16-Bit-Argument (0xC2) angewendet werden.

• Manchmal muss eine deaktivierte Funktion den Wert 0 oder 1 zurückgeben. Dies kann durch MOV EAX, 0 oder MOV EAX, 1 erreicht werden, was aber relativ ausführlich ist. Ein besserer Weg ist XOR EAX, EAX (2 Byte 0x31 0xC0) oder XOR EAX, EAX / INC EAX (3 Byte 0x31 0xC0 0x40).

Eine Software kann gegen Manipulation geschützt sein.


**German text placeholder** hat für diesen Zweck die BPM-Option.


Eine Möglichkeit ist es Sprünge zu schreiben, welche die Relocs umgehen oder die Reloc-Tabelle muss editiert werden.

10.2 Statistik von Funktionsargumenten

Ich war immer sehr daran interessiert welches die durchschnittliche Anzahl von Argumenten der einzelnen Funktionen ist.

Dazu wurden viele Windows 7 32-Bit-DLLs analysiert (crypt32.dll, mfc71.dll, msvcr100.dll, shell32.dll, user32.dll, d3d11.dll, mshtml.dll, msxml6.dll, sqlncli11.dll, wininet.dll, mfc120.dll, msvbvm60.dll, ole32.dll, themeui.dll, wmp.dll), da diese die stdcall-Konvention nutzen, was es einfach macht das Ergebnis des Disassemblers mit grep nach RETN X zu durchsuchen.

• keine Argumente: ≈ 29%
• 1 Argument: ≈ 23%
• 2 Argumente: ≈ 20%
• 3 Argumente: ≈ 11%
• 4 Argumente: ≈ 7%
• 5 Argumente: ≈ 3%
• 6 Argumente: ≈ 2%
• 7 Argumente: ≈ 1%

Abbildung 10.1: Statistiken von Funktionsargumenten

Das Ergebnis ist stark vom Programmierstil abhängig und kann bei anderen Programmen deutlich anders ausfallen.

10.3 Intrinsische Compiler-Funktionen


Beispielsweise gibt es keine zyklische Schiebe-Anweisungen in C/C++-Sprachen, in den meisten CPUs sind sie jedoch vorhanden. Um dem Programmierer das Leben einfacher zu machen hat zumindest MSVC die Pseudofunktionen \_rotl() und \_rotr()\(^1\) welche vom Compiler direkt in die ROL/ROR x86-Anweisungen übersetzt werden.

Ein anderes Beispiel sind Funktionen die SSE-Anweisungen direkt im Code umwandeln.

Eine vollständige Liste von intrinsischen Funktionen in MSVC ist hier zu finden: MSDN.

10.4 Compiler Anomalien

10.4.1 Oracle RDBMS 11.2 und Intel C++ 10.1

Der Intel C++ 10.1-Compiler, der für Oracle RDBMS 11.2 für Linux 86 genutzt wurde, kann zwei JZ in einer Reihe ausgeben. Es gibt keine Referenz zum zweiten JZ. Das zweite ist also ohne Bedeutung.

\(^1\)MSDN
Listing 10.1: kdli.o from libserver11.a

Listing 10.2: from the same code

Dies ist vermutlich ein Fehler im Codegenerator der während der Tests nicht gefunden wurde. Der resultierende Code funktioniert trotzdem.

10.4.2 MSVC 6.0
Gerade in einem altem Code gefunden:

Die erste FXCH-Anweisung tauscht ST (0) und ST (1), die zweite tu das gleiche, also haben beide zusammen keine Wirkung. Das Programm nutzt MFC42.dll, also könnte es sich bei dem Compiler im MSVC 6.0, 5.0 oder eventuell MSVC 4.2 aus den 1990ern handeln.

10.4.3 Zusammenfassung

10.5 Itanium

Auch wenn fast gescheitert, ist der Intel Itanium (IA64) eine sehr interessante Architektur.

Während OOE²-CPU entscheiden wie die Anweisungen neu organisiert werden und diese parallel ausführen, war EPIC³ ein Versuch diese Entscheidung dem Compiler zu überlassen: das Gruppieren der Anweisungen soll während des Compilierens erfolgen.

Dies führte zu einer berüchtigten Komplexität der Compiler.

Hier ist ein Beispiel von IA64-Code, ein einfacher kryptografischer Algorithmus aus dem Linux-Kernel:

Listing 10.3: Linux kernel 3.2.0.4

```c
#define TEA_ROUNDS 32
#define TEA_DELTA 0x9e3779b9

static void tea_encrypt(struct crypto_tfm *tfm, u8 *dst, const u8 *src)
{
    u32 y, z, n, sum = 0;
    u32 k0, k1, k2, k3;
    struct tea_ctx *ctx = crypto_tfm_ctx(tfm);
    const __le32 *in = (const __le32*)src;
    __le32 *out = (__le32*)dst;

    y = le32_to_cpu(in[0]);
    z = le32_to_cpu(in[1]);

    k0 = ctx->KEY[0];
    k1 = ctx->KEY[1];
    k2 = ctx->KEY[2];
    k3 = ctx->KEY[3];

    n = TEA_ROUNDS;

    while (n-- > 0) {
        sum += TEA_DELTA;
        y += (((z << 4) ^ k0) + (z + sum) ^ ((z >> 5) + k1));
        z += (((y << 4) ^ k2) + (y + sum) ^ ((y >> 5) + k3));
    }

    out[0] = cpu_to_le32(y);
    out[1] = cpu_to_le32(z);
}
```

Nachfolgend das Ergebnis des Compilierens:

Listing 10.4: Linux Kernel 3.2.0.4 für Itanium 2 (McKinley)

```
0090| tea_encrypt:
0090|08 80 80 41 00 21 adds r16 = 96, r32 // ptr to ctx->KEY[2]
0096|80 C0 82 00 42 00 adds r8 = 88, r32 // ptr to ctx->KEY[0]
009C|00 00 04 00 nop.i 0
00A0|09 18 70 41 00 21 adds r3 = 92, r32 // ptr to ctx->KEY[1]
00A6|F0 20 88 20 20 00 ld4 r15 = [r34], 4 // load z
00AC|44 06 01 84 adds r32 = 100, r32;; // ptr to ctx->KEY[3]
00B0|08 98 00 20 10 10 ld4 r19 = [r16] // r19=k2
00B6|00 01 00 00 42 40 mov r16 = r0 // r0 always contain zero
00BC|00 08 CA 00 mov.i r2 = ar.lc // save lc register
00C0|05 70 00 44 10 10 ld4 r14 = [r34] // load y
00CC|92 F3 CE 6B movl r17 = 0xFFFFF9E3779B9;; // TEA_DELTA
00D0|08 00 00 00 01 00 mov.m 0
00D6|50 01 20 20 20 00 ld4 r21 = [r8] // r21=k0
00DC|F0 09 2A 00 mov.i ar.lc = 31 // TEA_ROUNDS is 32
00E0|0A A0 00 06 10 10 ld4 r20 = [r3];; // r20=k1
00E6|20 01 80 20 20 00 ld4 r18 = [r32] // r18=k3
00EC|00 00 04 00 nop.i 0
00F0|
```

²Out-of-Order Execution
³Explicitly Parallel Instruction Computing
Zunächst sind alle IA64-Anweisungen in Pakete von 3 Anweisungen zusammengefasst.
Jedes Paket hat eine Größe von 16 Byte (128 Bit) und besteht aus Template-Code (5 Bit) und drei Anweisungen (je 41 Bit).
IDA zeigt die Pakete als 6+6+4 Byte, das Muster ist leicht zu erkennen.
Alle drei Anweisungen von jedem Paket wird in der Regel gleichzeitig ausgeführt, außer eine der Anweisungen enthält ein „Stop-Bit“.
Vermutlich haben die Intel- und HP-Ingenieure Statistiken über die am meisten verwendeten Anweisungsmuster erhoben und entschieden die Pakettypen zu erstellen (AKA „Templates“): ein Paket-Code definiert den Anweisungstyp im Paket. Es existieren 12 von ihnen.
Beispielsweise ist der nullte Pakettyp MII, was impliziert, dass die erste Anweisung Speicher (Lesen oder Schreiben) ist und die zweite und dritte jeweils eine Integer-Anweisung ist.
Ein weiteres Beispiel ist das Paket vom Typ 0x1d: MFB: die erste Anweisung betrifft wieder den Speicher (Lesen oder Schreiben), die zweite eine Fließkomma (FPU-Anweisung) und die dritte ein Springbefehl.
Wenn der Compiler keine passende Anweisung für den entsprechenden Paketplatz finden kann, ist es möglich, dass er ein NOP einfügt: man kann hier die nop.i-Anweisung (NOP anstelle einer Integer-Anweisung) oder nop.m (anstelle einer Speicheroperation) sehen.
NOPs werden automatisch eingefügt wenn mit Assembler gearbeitet wird.
Dies ist nicht alles: Pakete können ebenfalls gruppiert werden.
Jedes Paket kann ein „Stop-Bit“ enthalten, so dass alle aufeinander folgenden Pakete mit einem terminierenden Paket (mit „Stop-Bit“) gleichzeitig verarbeitet werden können.
In der Praxis kann Itanium 2 gleichzeitig zwei Pakete ausführen, was zu sechs Anweisungen führt.
Also kann keine der Anweisungen innerhalb einer Paket-Gruppe mit einer anderen interagieren (es kann also nicht zu Datenkonflikten kommen).
Falls sie auftreten können die Ergebnisse undefiniert sein.
Jedes Stop-Bit ist in Assembler mit zwei Semikolon (;;) nach der Anweisung markiert.
Die Anweisungen bei [90-ac] können also simultan ausgeführt werden: sie beeinflussen sich gegenseitig nicht. Die nächste Gruppe ist [b0-cc].
Hier ist auch das Stop-Bit bei 10c zu sehen Die nächste Anweisung bei 110 hat ebenfalls ein Stop-Bit.
Dies impliziert dass diese Anweisungen von allen anderen getrennt ausgeführt werden müssen (wie in CISC).

Außerdem ist zu sehen, dass die Anweisung nach 110 das Ergebnis der vorangehenden benutzt (Den Wert im Register r26), dementsprechend können sie nicht gleichzeitig ausgeführt werden.

Anscheinend war der Compiler nicht in der Lage einen besseren Weg zum Parallelisieren der Anweisungen zu finden, also die CPU so weit wie möglich auszulasten. Daher die vielen Stop-Bits und NOP-Anweisungen.

Manuelle Assembler-Programmierung ist ein mühsamer Job: der Programmierer muss die Anweisungen selber in Gruppen einteilen.

Der Programmierer ist immer noch in der Lage Stop-Bits zu jeder Anweisung hinzuzufügen, doch dies wird die Geschwindigkeit heruntersetzen für die Itanium gemacht wurde.

Ein interessantes Beispiel von manuellem Assembler-Code in IA64 kann im Code des Linux-Kernels gefunden werden:


Eine weitere Einführung für den Itanium-Assembler: [Mike Burrell, Writing Efficient Itanium 2 Assembly Code (2010)]\(^4\), [papasutra of haquebright, WRITING SHELLCODE FOR IA-64 (2001)]\(^5\).

Weitere sehr interessante Itanium-Features sind speculative execution und das NaT („not a thing“)-Bit, was in gewisser Weise German text placeholder-Zahlen ähnelt:

MSDN

**10.6 8086-Speichermodell**

Wenn es um 16-Bit-Programme für MS-DOS oder Win16 geht ( ?? on page ?? oder ?? on page ??), kann man sehen, dass die Zeiger aus zwei 16-Bit-Werten bestehen. Was bedeutet das? Ja, das ist wieder ein weiteres sonderbares Artefakt von MS-DOS und 8086.

8086/8088 war eine 16-Bit-CPU, war aber in der Lage 20-Bit-Adressen im RAM anzusprechen (und somit externen Speicher bis 1MB zu adressieren).

Der Adressbereich für externen Speicher ist aufgeteilt zwischen RAM (maximal 640KB), ROM!, Fenster für Videospeicher, EMS-Karten, etc.

Erinnern wir uns nochmal daran, dass der 8086/8088 der Nachfolger der 8-Bit-CPU 8080 war.

Der 8080 hat einen 16-Bit-Adressspeicher, kann also lediglich 64KB Speicher adressieren.

Möglichweise aus Gründen der Portierung alter Software\(^6\), kann der 8086 viele 64KB-Fenster gleichzeitig unterstützen, die sich im 1MB-Adressbereich befinden.

Dies ist eine Art Top-Level-Virtualisierung.

Alle 8086-Register sind 16-Bit breit. Um einen größeren Bereich adressieren zu können, wurden spezielle Segment-Register (CS, DS, ES, SS) eingeführt.

Jeder 20-Bit-Zeiger wird aus den Werten eines Segment-Registers und einem Adressregister-Paar (z.B. DS:BX) berechnet.

\[ \text{real adresse} = (\text{segment register} \ll 4) + \text{adress register} \]

Zum Beispiel: das Grafik-Video-Speicher-Fenster (EGA\(^7\), VGA\(^8\)) auf alten zu IBM PC kompatiblen Rechnern hat eine Größe von 64KB.

Um darauf zuzugreifen muss der Wert 0xA000 in eines der Segment-Register geschrieben werden, zum Beispiel in DS.

Anschließend wird DS:0 das erste Byte des Video-RAM und DS:0xFFFF das letzte Byte adressieren.

Die echte Adresse auf dem 20-Bit-Adressbus ist in dem Bereich zwischen 0xA0000 und 0xAFFFF.

Das Programm kann hart-kodierte Adressen wie 0x1234 beinhalten, das BS lädt das Programm aber bei Bedarf an eine beliebige Adresse. Dazu werden die Segment-Registerwerte derart neu berechnet, dass das Programm sich nicht darum kümmern muss an welcher Stelle im RAM es sich befindet.

---

\(^{4}\)German text placeholder http://yurichev.com/mirrors/RE/itanium.pdf

\(^{5}\)German text placeholder http://phrack.org/issues/57/5.html

\(^{6}\)Der Autor ist sich hier jedoch nicht 100% sicher.

\(^{7}\)Enhanced Graphics Adapter

\(^{8}\)Video Graphics Array
Jeder Zeiger in der alten MS-DOS-Umgebung besteht aus der Segmentadresse und der Adresse innerhalb des Segment, also zwei 16-Bit-Werten. 20 Bit sind hierfür genug, allerdings muss die Adresse recht oft neu berechnet werden. Mehr Informationen auf dem Stack zu übergeben schien eine bessere Speicher-/Komfort-Balance zu haben.

Übrigens: aufgrund all der vorherigen Überlegungen war es nicht mögliche Speicherblöcke zu allozieren die größer 64KB waren.

Die Segmentregister wurde beim 80286 als „Selektoren“ wieder genutzt, jedoch mit einer anderen Funktion. Als die 80386-CPU mit größerem RAM eingeführt wurde, war MS-DOS immer noch weit verbreitet, so das die DOS-Extender auftraten. Diese waren eigentlich ein Schritt zu einem „seriösen“ BS indem die CPU in den Protected Mode geschaltet wurde und sehr viel bessere Speicher-APIs für die Programme angeboten wurden, die noch unter MS-DOS liefen.

Sehr populäre Beispiele waren DOS/4GW (das Spiel DOOM wurde hierfür kompiliert), Phar Lap und PMODE. Übrigens wurde das gleiche Adressierungsmodell für Speicher in der 16-Bit-Reihe von Windows 3.x genutzt, bevor Win32 aufkam.

10.7 Basic Block Reordering

10.7.1 Profile-guided Optimization

Diese Optimierungsmethode kann einige German text placeholders zu anderen Sektionen der ausführbaren Datei verschieben.

Offensichtlich gibt es Teile einer Funktion die öfter ausgeführt werden als andere (zum Beispiel Schleifen-Rümpfe) und welche, die weniger oft ausgeführt werden (beispielsweise Fehlerberichte oder Ausnahmebehandlungen).

Der Compiler fügt Messcode in die ausführbare Datei ein. Anschließend führt der Programmierer diesen mit vielen Tests aus um Statistiken zu erstellen.

Der Compiler präpariert die ausführbare Datei mithilfe der erstellten Statistiken insofern, dass alle weniger häufige Codeteile in eine andere Sektion der Datei verschoben werden.

Als Ergebnis ist der häufig ausgeführte Funktionscode zusammengefasst, was sehr wichtig für die Ausführungszeit und die Cachebenutzung ist.

Ein Beispiel vom Oracle RDBMS-Code, der mit dem Intel C++-Compiler übersetzt wurde:

```assembly
Listing 10.5: orageneric11.dll (win32)

public _skgfsync
_skgfsync proc near
; address 0x6030D86A

    db 66h
    nop
    push ebp
    mov ebp, esp
    mov edx, [ebp+0Ch]
    test edx, edx
    jz short loc_6030D884
    mov eax, [edx+30h]
    test eax, 400h
    jnz __VInfreq__skgfsync ; write to log

continue:

    mov eax, [ebp+8]
    mov edx, [ebp+10h]
    mov dword ptr [eax], 0
    lea eax, [edx+0Fh]
    and eax, 0FFFFFFFCh
    mov ecx, [eax]
    cmp ecx, 45726963h
    jnz error ; exit with error

    mov esp, ebp
    pop ebp
    retn

_skgfsync endp
```

517
Der Abstand der Adressen zwischen diesen beiden Code-Fragmenten beträgt fast 9 MB.
Alle weniger oft ausgeführten Codeteile wurden an das Ende der Code-Sektion der DLL-Datei verschoben.
Dieser Teil der Funktion wurde vom Intel C++-Compiler mit dem VInfreq-Präfix markiert.
Man kann hier sehen, dass der Teil der Funktion der in die Logdatei schreibt (zum Beispiel im Falle eines Fehlers oder einer Warnung) vermutlich selten oder vielleicht gar nicht ausgeführt wurde als der Entwickler von Oracle die Statistiken erstellt hat.
Das Schreiben in log basic block gibt die Ausführkontrolle letztendlich wieder zurück an den „heißen“ Teil der Funktion.
Ein weiterer „seltener“ Teil ist der German text placeholder, welcher den Fehlercode 27050 zurück gibt.
In Linux ELF-Dateien wird der selten ausgeführte Code vom Intel C++-Compiler in die separate text.unlikely-Sektion verschoben und der „heiße“ Code in die Sektion text.hot.
Aus Sicht eines Reverse-Engineers kann diese Information helfen um die Funktion in den Hauptteil und den Fehlerbehandlungsteil zu unterteilen.
Kapitel 11

Bücher / Lesenswerte Blogs

11.1 Bücher und andere Materialien

11.1.1 Reverse Engineering

- Ebenfalls das Buch von Kris Kaspersky.

11.1.2 Windows

- Mark Russinovich, *Microsoft Windows Internals*
- Peter Ferrie – The “Ultimate” Anti-Debugging Reference

Blogs:
- Microsoft: Raymond Chen
- nynaeve.net

11.1.3 C/C++

- C++11 standard³
- Marshall Cline, *C++ FAQ*⁵
- Dennis Yurichev, *C/C++ programming language notes*⁶
- JPL Institutional Coding Standard for the C Programming Language⁷

---

2² [German text placeholder](http://go.yurichev.com/17274)
3³ [German text placeholder](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3698.pdf)
4⁴ [German text placeholder](http://agner.org/optimize/optimizing_cpp.pdf)
5⁵ [German text placeholder](http://go.yurichev.com/17291)
6⁶ [German text placeholder](http://yurichev.com/C-book.html)
7⁷ [German text placeholder](https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf)
11.1.4 x86 / x86-64

- Intel Handbücher
- AMD Handbücher

Etwas veraltet aber immer noch interessant zu lesen:


11.1.5 ARM

- ARM Handbücher
- [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)]

11.1.6 Assembler

Richard Blum — Professional Assembly Language.

11.1.7 Java


11.1.8 UNIX


11.1.9 Programmierung Allgemein


11.1.10 Kryptografie

- (Free) lvh, *Crypto 101*
• (Free) Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*\textsuperscript{18}.

\textsuperscript{18}German text placeholder https://crypto.stanford.edu/~dabo/cryptobook/
Kapitel 12

Communities


Im IRC gibt es einen ##re-Kanal auf FreeNode\(^2\).
Nachwort
12.1 Fragen?

Zögern Sie nicht dem Autor Ihre Fragen per <first_name @ last_name . com> / <first_name . last_name @ gmail . com> zu schicken. Haben Sie irgendwelche Vorschläge für neue Inhalte in diesem Buch? Gerne können Sie Korrekturen (auch grammatikaler Art, vor allem im Englischen) zuschicken.

Der Autor arbeitet sehr viel an diesem Buch, so dass sich Seitenzahlen, Nummerierungen und so weiter schnell ändern können. Bitte beziehen Sie sich also nicht auf diese Angaben. Einfacher ist es einen Screenshot von der entsprechenden Seite zu machen und den Fehler in einer Bildbearbeitung zu markieren. Der Autor kann so den Fehler sehr viel schneller korrigieren. Wenn Sie sich mit git und \LaTeX auskennen, können Sie den Fehler direkt im Quellcode ändern:

GitHub.

Auch wenn Sie sich nicht ganz sicher sind, teilen Sie bitte die kleinen oder großen Fehler mit, die Sie finden. Dieses Buch richtet sich speziell an Anfänger, so dass deren Meinungen und Kommentare einen entscheidenden Einfluss haben.
Anhang
.1 x86

.1.1 Terminologie

Geläufig für 16-Bit (8086/80286), 32-Bit (80386, etc.), 64-Bit.


**Wort** 16-Bit. DW Assembler-Direktive —""—. Bytes werden in dem 16-Bit-Teil der folgenden Register übergeben: AX/BX/CX/DX/SI/DI/R*W.

**Doppelwort** („dword“) 32-Bit. DD Assembler-Direktive —""—. Doppelwörter werden in Registern (x86) oder dem 32-Bit-Teil der Register (x64) übergeben. In 16-Bit-Code werden Doppelwörter in 16-Bit-Registerpaaren übergeben.

**zwei Doppelwörter** („qword“) 64-Bit. DQ Assembler-Direktive —""—. In 32-Bit-Umgebungen werden diese in 32-Bit-Registerpaaren übergeben.

**tbyte** (10 Byte) 80-Bit oder 10 Bytes (für IEEE 754 FPU Register).

**paragraph** (16 Byte) — Bezeichnung war in MS-DOS Umgebungen gebräuchlich.

Datentypen der selben Breite (BYTE, WORD, DWORD) entsprechen auch denen in der Windows API.

.1.2 Mehrzweckregister

Es ist möglich auf viele Register byteweise oder über 16-Bit-Wortteile zuzugreifen.

Ältere 8-Bit-CPUs (8080) hatten 16-Bit-Register geteilt durch zwei.

Programme für den 8080 geschrieben wurden, konnten auf den niederwertigen Teil von 16-Bit-Registern, den höherwertigen Teil oder auf das ganze 16-Bit-Register zugreifen.

Dieses Feature wurde vielleicht als Hilfe für eine einfachere Portierung beim 8086 beibehalten.

Register mit dem Präfix R- tauchen bei x86-64 auf; der Präfix E- —bei 80386.

Dementsprechend sind, R-Register 64-Bit und E-Register 32-Bit.

8 weitere GPR's wurden bei x86-66 hinzugefügt: R8-R15.

N.B.:

RAX/EAX/AX/AL

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>German text placeholder</td>
</tr>
<tr>
<td>RAX</td>
</tr>
<tr>
<td>EAX</td>
</tr>
<tr>
<td>AX</td>
</tr>
<tr>
<td>AH</td>
</tr>
<tr>
<td>AL</td>
</tr>
</tbody>
</table>

AKA Akkumulator. Das Ergebnis einer Funktion wird in der Regel in einem Register zurückgegeben.

RBX/EBX/BX/BL

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>German text placeholder</td>
</tr>
<tr>
<td>RBX</td>
</tr>
<tr>
<td>EBX</td>
</tr>
<tr>
<td>BX</td>
</tr>
<tr>
<td>BH</td>
</tr>
<tr>
<td>BL</td>
</tr>
</tbody>
</table>
RCX/ECX/CX/CL

AKA Zähler: in dieser Rolle in REP Befehlen und auch als Schiebeoperationen verwendet (SHL/SHR/RxL/RxR).

RDX/EDX/DX/DL

AKA „source index“. Als Quelle der Befehle verwendet REP MOVSx, REP CMPSx.

RDI/EDI/DI/DIL

AKA „destination index“. Als Zeiger auf das Ziel der Befehle verwendet REP MOVSx, REP STOSx.

R8/R8D/R8W/R8L

R9/R9D/R9W/R9L
### R10/R10D/R10W/R10L

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10</td>
</tr>
<tr>
<td>R10D</td>
</tr>
<tr>
<td>R10W</td>
</tr>
<tr>
<td>R10L</td>
</tr>
</tbody>
</table>

### R11/R11D/R11W/R11L

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R11</td>
</tr>
<tr>
<td>R11D</td>
</tr>
<tr>
<td>R11W</td>
</tr>
<tr>
<td>R11L</td>
</tr>
</tbody>
</table>

### R12/R12D/R12W/R12L

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R12</td>
</tr>
<tr>
<td>R12D</td>
</tr>
<tr>
<td>R12W</td>
</tr>
<tr>
<td>R12L</td>
</tr>
</tbody>
</table>

### R13/R13D/R13W/R13L

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R13</td>
</tr>
<tr>
<td>R13D</td>
</tr>
<tr>
<td>R13W</td>
</tr>
<tr>
<td>R13L</td>
</tr>
</tbody>
</table>

### R14/R14D/R14W/R14L

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R14</td>
</tr>
<tr>
<td>R14D</td>
</tr>
<tr>
<td>R14W</td>
</tr>
<tr>
<td>R14L</td>
</tr>
</tbody>
</table>

### R15/R15D/R15W/R15L

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R15</td>
</tr>
<tr>
<td>R15D</td>
</tr>
<tr>
<td>R15W</td>
</tr>
<tr>
<td>R15L</td>
</tr>
</tbody>
</table>

### RSP/ESP/SP/SPL

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSP</td>
</tr>
<tr>
<td>ESP</td>
</tr>
<tr>
<td>SP</td>
</tr>
<tr>
<td>SPL</td>
</tr>
</tbody>
</table>

*AKA Stapel-Zeiger.*
RBP/EBP/BP/BPL

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>German text placeholder</td>
</tr>
<tr>
<td>RBP</td>
</tr>
</tbody>
</table>

AKA frame pointer. : (1.8.1 on page 52).

RIP/EIP/IP

<table>
<thead>
<tr>
<th>Byte-Nummer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>German text placeholder</td>
</tr>
<tr>
<td>RIP</td>
</tr>
</tbody>
</table>

AKA „instruction pointer“ ³:  

MOV EAX, ... 
JMP EAX :

PUSH value 
RET

CS/DS/ES/SS/FS/GS

(CS), (DS), (SS).

FS German text placeholder win32 TLS, ... (10.6 on page 516).

AKA EFLAGS.

³„program counter“
<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (1)</td>
<td>CF (Carry)</td>
<td>CLC/STC/CMC</td>
</tr>
<tr>
<td>2 (4)</td>
<td>PF (Parity)</td>
<td>(1.17.7 on page 210).</td>
</tr>
<tr>
<td>4 (0x10)</td>
<td>AF (Adjust)</td>
<td></td>
</tr>
<tr>
<td>6 (0x40)</td>
<td>ZF (Zero)</td>
<td>0</td>
</tr>
<tr>
<td>7 (0x80)</td>
<td>SF (Sign)</td>
<td></td>
</tr>
<tr>
<td>8 (0x100)</td>
<td>TF (Trap)</td>
<td></td>
</tr>
<tr>
<td>9 (0x200)</td>
<td>IF (Interrupt enable)</td>
<td>.</td>
</tr>
<tr>
<td>10 (0x400)</td>
<td>DF (Direction)</td>
<td>REP MOVSx/CMPSx/LODSx/SCASx. CLD/STD</td>
</tr>
</tbody>
</table>
| 11 (0x800) | OF (Overflow) | ??
| 12, 13 (0x3000) | IOPL (I/O privilege level) | i286 |
| 14 (0x4000) | NT (Nested task) | i286 |
| 16 (0x10000) | RF (Resume) | i386 |
| 17 (0x20000) | VM (Virtual 8086 mode) | i386 |
| 18 (0x40000) | AC (Alignment check) | i386 |
| 19 (0x80000) | VIF (Virtual interrupt) | i386 |
| 20 (0x100000) | VIP (Virtual interrupt pending) | i386 |
| 21 (0x200000) | ID (Identification) | i386 |

### 1.3 FPU German text placeholder

8 80:- ST(0)-ST(7). N.B.: IDA ST(0) ST. IEEE 754.

\[
\begin{array}{cccc|c}
79 & 78 & 64 & 63 & 62 & 0 \\
S & I
\end{array}
\]

(S — , I — )

FPU.
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IM (Invalid operation Mask)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>DM (Denormalized operand Mask)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ZM (Zero divide Mask)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>OM (Overflow Mask)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>UM (Underflow Mask)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>PM (Precision Mask)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>iEM (Interrupt Enable Mask)</td>
<td></td>
</tr>
<tr>
<td>8, 9</td>
<td>PC (Precision Control)</td>
<td>00 — 24 (REAL4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 — 53 (REAL8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 — 64 (REAL10)</td>
</tr>
<tr>
<td>10, 11</td>
<td>RC (Rounding Control)</td>
<td>00 —</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 — +∞</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 — +∞</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 — 0</td>
</tr>
<tr>
<td>12</td>
<td>IC (Infinity Control)</td>
<td>0 — () +∞ German text placeholder −∞</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 — +∞ German text placeholder −∞</td>
</tr>
</tbody>
</table>

PM, UM, OM, ZM, DM, IM.


\[
\begin{array}{|c|c|}
\hline
15 & B (Busy) \\
14 & C3 \\
13, 12, 11 & TOP \\
10 & C2 \\
9 & C1 \\
8 & C0 \\
7 & IR (Interrupt Request) \\
6 & SF (Stack Fault) \\
5 & P (Precision) \\
4 & U (Underflow) \\
3 & O (Overflow) \\
2 & Z (Zero) \\
1 & D (Denormalized) \\
0 & I (Invalid operation) \\
\hline
\end{array}
\]

SF, P, U, O, Z, D, I.

C3, C2, C1, C0: (1.17.7 on page 209).

N.B.: .

**Tag Word**


\[
\begin{array}{|c|c|}
\hline
15, 14 & Tag(7) \\
13, 12 & Tag(6) \\
11, 10 & Tag(5) \\
9, 8 & Tag(4) \\
7, 6 & Tag(3) \\
5, 4 & Tag(2) \\
3, 2 & Tag(1) \\
1, 0 & Tag(0) \\
\hline
\end{array}
\]

• 00 —
• 01 —
• 11 —
.1.4 SIMD German text placeholder

MMX German text placeholder

8 64:- MM0..MM7.

SSE German text placeholder AVX German text placeholder

SSE: 8 128-Bit-Register: XMM0..XMM7. x86-64 : XMM8..XMM15.
AVX ist eine Erweiterung aller Register auf 256 Bit.

.1.5

hardware breakpoints.

- DR0 — #1
- DR1 — #2
- DR2 — #3
- DR3 — #4
- DR6 —
- DR7 —

DR6

<table>
<thead>
<tr>
<th>()</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (1)</td>
<td>B0 —</td>
<td></td>
</tr>
<tr>
<td>1 (2)</td>
<td>B1 —</td>
<td></td>
</tr>
<tr>
<td>2 (4)</td>
<td>B2 —</td>
<td></td>
</tr>
<tr>
<td>3 (8)</td>
<td>B3 —</td>
<td></td>
</tr>
<tr>
<td>13 (0x2000)</td>
<td>BD —</td>
<td></td>
</tr>
<tr>
<td>14 (0x4000)</td>
<td>BS — .</td>
<td></td>
</tr>
<tr>
<td>15 (0x8000)</td>
<td>BT (task switch flag)</td>
<td></td>
</tr>
</tbody>
</table>

N.B. . (.1.2 on page 529).

DR7

<table>
<thead>
<tr>
<th>()</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (1)</td>
<td>L0 — #1</td>
<td></td>
</tr>
<tr>
<td>1 (2)</td>
<td>G0 — #1</td>
<td></td>
</tr>
<tr>
<td>2 (4)</td>
<td>L1 — #2</td>
<td></td>
</tr>
<tr>
<td>3 (8)</td>
<td>G1 — #2</td>
<td></td>
</tr>
<tr>
<td>4 (0x10)</td>
<td>L2 — #3</td>
<td></td>
</tr>
<tr>
<td>5 (0x20)</td>
<td>G2 — #3</td>
<td></td>
</tr>
<tr>
<td>6 (0x40)</td>
<td>L3 — #4</td>
<td></td>
</tr>
<tr>
<td>7 (0x80)</td>
<td>G3 — #4</td>
<td></td>
</tr>
<tr>
<td>8 (0x100)</td>
<td>LE —</td>
<td></td>
</tr>
<tr>
<td>9 (0x200)</td>
<td>GE —</td>
<td></td>
</tr>
<tr>
<td>13 (0x2000)</td>
<td>GD —</td>
<td></td>
</tr>
<tr>
<td>16,17 (0x30000)</td>
<td>#1: R/W —</td>
<td></td>
</tr>
<tr>
<td>18,19 (0xC0000)</td>
<td>#1: LEN —</td>
<td></td>
</tr>
<tr>
<td>20,21 (0x300000)</td>
<td>#2: R/W —</td>
<td></td>
</tr>
<tr>
<td>22,23 (0xC00000)</td>
<td>#2: LEN —</td>
<td></td>
</tr>
<tr>
<td>24,25 (0x3000000)</td>
<td>#3: R/W —</td>
<td></td>
</tr>
<tr>
<td>26,27 (0xC000000)</td>
<td>#3: LEN —</td>
<td></td>
</tr>
<tr>
<td>28,29 (0x30000000)</td>
<td>#4: R/W —</td>
<td></td>
</tr>
<tr>
<td>30,31 (0xC0000000)</td>
<td>#4: LEN —</td>
<td></td>
</tr>
</tbody>
</table>
(R/W):
  • 00 —
  • 01 —
  • 10 —
  • 11 —
N.B.: .

(LEN):
  • 00 —
  • 01 —
  • 10 —
  • 11 —

.1.6
( 10.3 on page 512).
.
( 10.1.1 on page 511).

**LOCK** . . . ADD, AND, BTR, BTS, CMPXCHG, OR, XADD, XOR. ( 6.5.4 on page 502).
**REP** MOVsx **German text placeholder** STOSx: . MOVsx ( .1.6 on page 535) **German text placeholder**
STOSx ( .1.6 on page 536).
.
**REPE/REPNE** (AKA REPZ/REPNZ) CMPSx **German text placeholder** SCASx: . .
CMPSx ( .1.6 on page 536) **German text placeholder** SCASx ( .1.6 on page 535).
.
.
**ADC** *(add with carry)*
  : 1.23 on page 358.

**ADD**
**AND**
**CALL** :
  PUSH address_after_CALL_instruction; JMP label

**CMP**
**DEC** **German text placeholder**.
**IMUL** 2.1.1 on page 409.
**INC** **German text placeholder**.
**JCXZ, JECXZ, JRCXZ** (M)
**JMP** . **German text placeholder**.
**Jcc** *( cc — condition code)*
  . . **German text placeholder**.
  **JAE** AKA JNC: : CF=0
  **JB** AKA JC: : CF=1
JC  AKA  JB:  
JE  AKA  JZ:  ZF=1  
JGE : SF=OF  
JL : SF=OF  
JNAE  AKA  JC:  CF=1  
JNA  CF=1  German text placeholder  ZF=1  
JNBE : CF=0  German text placeholder  ZF=0  
JNB  AKA  JNC:  CF=0  
JNC  AKA  JAE:  JNB.  
JNE  AKA  JNZ:  ZF=0  
JNGE : SF≠OF  
JNLE : ZF=0  German text placeholder  SF=OF  
JNL : SF=OF  
JNO : OF=0  
JNS  
JNZ  AKA  JNE:  ZF=0  
JO : OF=1  
JPO (Jump Parity Odd)  
JP  AKA  JPE:  
JS  
JZ  AKA  JE:  ZF=1  

LAHF :  

\[
\begin{array}{cccc}
\text{SF} & \text{ZF} & \text{AF} & \text{PF} & \text{CF} \\
\hline
7 & 6 & 4 & 2 & 0
\end{array}
\]

LEAVE  
LEA (Load Effective Address)  

int f(int a, int b)  
{  
    return a*8+b;  
};

Listing 1: German text placeholder MSVC 2010  

<table>
<thead>
<tr>
<th>Proc</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_f</td>
<td>a$ = 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc</td>
<td></td>
<td>; size = 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_f</td>
<td>b$ = 12</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|      | mov  eax, DWORD PTR b$[esp-4]  
|      | mov  ecx, DWORD PTR a$[esp-4]  
|      | lea  eax, DWORD PTR [eax+ecx*8]  
|      | ret  0  
| _f   | ENDP

Intel C++ :
```c
int f1(int a)
{
    return a*13;
};
```

Listing 2: Intel C++ 2011

```assembly
_f1 PROC NEAR
mov   ecx, DWORD PTR [4+esp] ; ecx = a
lea   edx, DWORD PTR [ecx+ecx*8] ; edx = a*9
lea   eax, DWORD PTR [edx+ecx*4] ; eax = a*9 + a*4 = a*13
ret
```

**MOVSB/MOVSW/MOVSQ** / 16-/ 32-/ 64- SI/ESI/RSI DI/EDI/RDI.

`memcpy()` .

`memcpy(EDI, ESI, 15):`

**MOVSX : (1.15.1 on page 179)**

**MOVZX i: (1.15.1 on page 180)**

**MUL** : 2.1.1 on page 409.

**NEG : op = −op**

**NOP** : NOP. XCHG EAX,EAX. . 1.7.1 on page 47.

`: (.1.7 on page 542).

**NOT op1: op1 = ¬op1.**

**OR**

**RET** : POP tmp; JMP tmp.

`: RET RETN („return near“) (10.6 on page 516), RETF („return far“).

`: POP tmp; ADD ESP op1; JMP tmp.RET : 6.1.2 on page 448.

**SAHF :**

```
+---+---+---+---+
| SF | ZF | AF | PF |
+---+---+---+---+
```

**SBB (subtraction with borrow)** : 1.23 on page 358.

**SCASB/SCASW/SCASD/SCASQ (M)** / 16-/ 32-/ 64- AX/EAX/RAX DI/EDI/RDI. .

**: .:**

**SHL**

**SHR :**

```
+---+---+---+---+---+---+---+---+
| CF | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
+---+---+---+---+---+---+---+---+
```

535
The process of stretching numbers by extending the sign bit is called sign extension. The 8086 provides instructions (Fig. 3.29) to facilitate the task of sign extension. These instructions were initially named SEX (sign extend) but were later renamed to the more conservative CBW (convert byte to word) and CWD (convert word to double word).
Listing 3: base
ntos\rtl\386\movemem.asm

; ULONG
; RtlCompareMemory ( IN PVOID Source1, IN PVOID Source2, IN ULONG Length )

; Routine Description:
; This function compares two blocks of memory and returns the number
; of bytes that compared equal.

; Arguments:
; Source1 (esp+4) - Supplies a pointer to the first block of memory to
; compare.
; Source2 (esp+8) - Supplies a pointer to the second block of memory to
; compare.
; Length (esp+12) - Supplies the Length, in bytes, of the memory to be
; compared.

; Return Value:
; The number of bytes that compared equal is returned as the function
; value. If all bytes compared equal, then the length of the original
; block of memory is returned.

RcmSource1    equ  [esp+12]
RcmSource2    equ  [esp+16]
RcmLength     equ  [esp+20]

CODE_ALIGNMENT
cPublicProc  _RtlCompareMemory,3
cPublicFpo  3,0

push  esi      ; save registers
push  edi
cld          ; clear direction
mov  esi,RcmSource1  ; (esi) -> first block to compare
mov  edi,RcmSource2  ; (edi) -> second block to compare

; Compare dwords, if any.

rcm10:  mov  ecx,RcmLength          ; (ecx) = length in bytes
        shr  ecx,2                ; (ecx) = length in dwords
        jz   rcm20                ; no dwords, try bytes
        repe cmpsd               ; compare dwords
        jnz  rcm40                ; mismatch, go find byte

; Compare residual bytes, if any.

rcm20:  mov  ecx,RcmLength          ; (ecx) = length in bytes
        and  ecx,3                ; (ecx) = length mod 4
        jz   rcm30                ; 0 odd bytes, go do dwords
        repe cmpsb               ; compare odd bytes
        jnz  rcm50                ; mismatch, go report how far we got

; All bytes in the block match.
rcm30:  mov   eax,RcmLength       ; set number of matching bytes
      pop   edi                   ; restore registers
      pop   esi                   
      stdRET _RtlCompareMemory

; When we come to rcm40, esi (and edi) points to the dword after the
; one which caused the mismatch. Back up 1 dword and find the byte.
; Since we know the dword didn't match, we can assume one byte won't.

rcm40:  sub   esi,4               ; back up
      sub   edi,4               ; back up
      mov   ecx,5               ; ensure that ecx doesn't count out
      repe cmpsb                ; find mismatch byte

; When we come to rcm50, esi points to the byte after the one that
; did not match, which is TWO after the last byte that did match.

rcm50:  dec   esi                ; back up
      sub   esi,RcmSource1      ; compute bytes that matched
      mov   eax,esi
      pop   edi
      pop   esi
      stdRET _RtlCompareMemory

stdENDP _RtlCompareMemory

N.B.:  

CPUID  :( 1.21.6 on page 344).

DIV

IDIV

INT  (M): INT x PUSHF; CALL dword ptr [x*4] ....
        .: [Ralf Brown Ralf Brown’s Interrupt List], .
        .

INT 3 (M): INT, (0xCC), 0xCC.
               German text placeholder, EXCEPTION_BREAKPOINT . . MSVS5. German text placeholder, .
               MSVC  German text placeholder: __debugbreak()6.
               kernel32.dll DebugBreak()7, INT 3.

IN  (M).  ( ?? on page ??).

IRET:  . POP tmp; POPF; JMP tmp.

LOOP  (M) CX/ECX/RCX, .

      MOV      CX, nnnn
      LABEL: LOOP LABEL

OUT  (M).  ( ?? on page ??).

POPA  (M) (R|E)DI, (R|E)SI, (R|E)BP, (R|E)BX, (R|E)DX, (R|E)CX, (R|E)AX.

POPCNT population count.  

POPF  (AKA )

PUSHA  (M) (R|E)AX, (R|E)CX, (R|E)DX, (R|E)BX, (R|E)BP, (R|E)SI, (R|E)DI.

5Microsoft Visual Studio
6MSDN
7MSDN
PUSHF (AKA )

RCL (M) :

```
7 6 5 4 3 2 1 0
    CF

CF
7 6 5 4 3 2 1 0
```

RCR (M) :

```
7 6 5 4 3 2 1 0
    CF

CF
7 6 5 4 3 2 1 0
```

ROL/ROR (M)

ROL: :

```
7 6 5 4 3 2 1 0
    CF

CF
7 6 5 4 3 2 1 0
```

ROR: :

```
7 6 5 4 3 2 1 0
    CF

CF
7 6 5 4 3 2 1 0
```

(compiler intrinsics) _rotl() German text placeholder _rotr()¹³.

SAL, SHL

```
7 6 5 4 3 2 1 0
```

SAR

```
7 6 5 4 3 2 1 0
```

MSB!

SETcc op: . ( .1.6 on page 533).

STC (M)

STD (M). ntoskrnl.exe.

STI (M)

SYSCALL (AMD) ( 6.3 on page 461)

SYSENTER (Intel) ( 6.3 on page 461)

UD2 (M)

XCHG (M)

.-P

FABS

¹³MSDN
FADD  op: ST(0)=op+ST(0)
FADD  ST(0), ST(i): ST(0)=ST(0)+ST(i)
FADDP ST(1)=ST(0)+ST(1);
FCHS  ST(0)=−ST(0)
FCOM  ST(0) ST(1)
FCOM  op: ST(0) op
FCOMP ST(0) ST(1);
FCOMP ST(0) ST(1);
FDIVR  op: ST(0)=op/ST(0)
FDIVR  ST(i), ST(j): ST(i)=ST(j)/ST(i)
FDIVRP op: ST(0)=op/ST(0);
FDIVRP ST(i), ST(j): ST(i)=ST(j)/ST(i);
DIV  op: ST(0)=ST(0)/op
DIV  ST(i), ST(j): ST(i)=ST(i)/ST(j)
DIVP ST(1)=ST(0)/ST(1);
FILD  op: .
FIST  op: ST(0) op
FISTP op: ST(0) op;
FLD1
FLDCW  op: FPU control word (.1.3 on page 530) 16-bit op.
FLDZ
FLD  op: .
FMUL  op: ST(0)=ST(0)*op
FMUL  ST(i), ST(j): ST(i)=ST(i)*ST(j)
FMULP op: ST(0)=ST(0)*op;
FMULP ST(i), ST(j): ST(i)=ST(i)*ST(j);
FSINCS  : tmp=ST(0); ST(1)=sin(tmp); ST(0)=cos(tmp)
FSQRT  : ST(0)=√ST(0)
FSTCW  op: FPU control word (.1.3 on page 530) 16-bit op.
FNSTCW op: FPU control word (.1.3 on page 530) 16-bit op.
FSTSW  op: FPU status word (.1.3 on page 531) 16-bit op.
FNSTSW op: FPU status word (.1.3 on page 531) 16-bit op.
FST  op: ST(0) op
FSTP op: ST(0) op;
FSUBR  op: ST(0)=op−ST(0)
FSUBR  ST(0), ST(i): ST(0)=ST(i)−ST(0)
FSUBRP ST(1)=ST(0)−ST(1);
FSUB  op: ST(0)=ST(0)−op
FSUB  ST(0), ST(i): ST(0)=ST(0)−ST(i)
FSUBP ST(1)=ST(1)−ST(0);
FUCOM ST(i): ST(0) German text placeholder ST(i)
FUCOM ST(0) German text placeholder ST(1)
FUCOMP ST(0) German text placeholder ST(1); .
FUCOMPP ST(0) German text placeholder ST(1); .

.  
FXCH ST(i)
FXCH

().
.
: ?? on page ??.

<table>
<thead>
<tr>
<th>ASCII</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>39</td>
</tr>
<tr>
<td>:</td>
<td>3a</td>
</tr>
<tr>
<td>;</td>
<td>3b</td>
</tr>
<tr>
<td>&lt;</td>
<td>3c</td>
</tr>
<tr>
<td>=</td>
<td>3d</td>
</tr>
<tr>
<td>?</td>
<td>3f</td>
</tr>
<tr>
<td>@</td>
<td>40</td>
</tr>
<tr>
<td>À</td>
<td>41</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
</tr>
<tr>
<td>D</td>
<td>44</td>
</tr>
<tr>
<td>E</td>
<td>45</td>
</tr>
<tr>
<td>F</td>
<td>46</td>
</tr>
<tr>
<td>G</td>
<td>47</td>
</tr>
<tr>
<td>H</td>
<td>48</td>
</tr>
<tr>
<td>I</td>
<td>49</td>
</tr>
<tr>
<td>J</td>
<td>4a</td>
</tr>
<tr>
<td>K</td>
<td>4b</td>
</tr>
<tr>
<td>L</td>
<td>4c</td>
</tr>
<tr>
<td>M</td>
<td>4d</td>
</tr>
<tr>
<td>N</td>
<td>4e</td>
</tr>
<tr>
<td>O</td>
<td>4f</td>
</tr>
<tr>
<td>P</td>
<td>50</td>
</tr>
<tr>
<td>Q</td>
<td>51</td>
</tr>
<tr>
<td>R</td>
<td>52</td>
</tr>
<tr>
<td>S</td>
<td>53</td>
</tr>
<tr>
<td>T</td>
<td>54</td>
</tr>
<tr>
<td>U</td>
<td>55</td>
</tr>
<tr>
<td>V</td>
<td>56</td>
</tr>
<tr>
<td>W</td>
<td>57</td>
</tr>
<tr>
<td>X</td>
<td>58</td>
</tr>
<tr>
<td>Y</td>
<td>59</td>
</tr>
<tr>
<td>Z</td>
<td>5a</td>
</tr>
<tr>
<td>[</td>
<td>5b</td>
</tr>
<tr>
<td>\</td>
<td>5c</td>
</tr>
<tr>
<td>]</td>
<td>5d</td>
</tr>
<tr>
<td>^</td>
<td>5e</td>
</tr>
<tr>
<td>_</td>
<td>5f</td>
</tr>
<tr>
<td>`</td>
<td>60</td>
</tr>
<tr>
<td>a</td>
<td>61</td>
</tr>
<tr>
<td>h</td>
<td>68</td>
</tr>
<tr>
<td>i</td>
<td>69</td>
</tr>
<tr>
<td>j</td>
<td>6a</td>
</tr>
</tbody>
</table>
Dies ist ein Assembler-Makro um Labels an bestimmten Grenzen auszurichten.


Entnommen von listing.inc (MSVC):

Dies ist übrigens ein Beispiel für die unterschiedlichen NOP-Variationen. Keine dieser Anweisungen hat eine Auswirkung, aber alle haben eine unterschiedliche Größe.

Eine einzelne Idle-Anweisung anstatt mehrerer NOPs hat positive Auswirkungen auf die CPU-Performance.

```assembly
; Listing INC
;
; This file contains assembler macros and is included by the files created
; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
; Assembler).
;
; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

; non destructive nops
npad macro size
  if size eq 1
    nop
  else
    if size eq 2
      mov edi, edi
    else
      if size eq 3
        lea ecx, [ecx+00]
        db 80h, 49h, 00h
      else
        if size eq 4
          lea esp, [esp+00]
          db 80h, 64h, 24h, 00h
        else
          if size eq 5
```

: AAA, AAS, CMP, DEC, IMUL, INC, JA, JAE, JB, JBE, JE, JNE, JNO, JNS, JO, JP, JS, POP, POPA, PUSH, PUSHA, XOR.

## 1.7 npad

Dies ist ein Assembler-Makro um Labels an bestimmten Grenzen auszurichten.


Entnommen von listing.inc (MSVC):

Dies ist übrigens ein Beispiel für die unterschiedlichen NOP-Variationen. Keine dieser Anweisungen hat eine Auswirkung, aber alle haben eine unterschiedliche Größe.

Eine einzelne Idle-Anweisung anstatt mehrerer NOPs hat positive Auswirkungen auf die CPU-Performance.

```assembly
; Listing INC
;
; This file contains assembler macros and is included by the files created
; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
; Assembler).
;
; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

; non destructive nops
npad macro size
  if size eq 1
    nop
  else
    if size eq 2
      mov edi, edi
    else
      if size eq 3
        lea ecx, [ecx+00]
        db 80h, 49h, 00h
      else
        if size eq 4
          lea esp, [esp+00]
          db 80h, 64h, 24h, 00h
        else
```
add eax, DWORD PTR 0
else
  if size eq 6
    ; lea ebx, [ebx+00000000]
    DB 8DH, 9BH, 00H, 00H, 00H, 00H
  else
    if size eq 7
      ; lea esp, [esp+00000000]
      DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
    else
      if size eq 8
        ; jmp .+8; .npad 6
        DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H, 00H
      else
        if size eq 9
          ; jmp .+9; .npad 7
          DB 0EBH, 07H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
        else
          if size eq 10
            ; jmp .+A; .npad 7; .npad 1
            DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 90H
          else
            if size eq 11
              ; jmp .+C; .npad 7; .npad 2
              DB 0EBH, 09H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8BH, 0FFH
            else
              if size eq 12
                ; jmp .+E; .npad 7; .npad 3
                DB 0EBH, 0AH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 49H, 00H
              else
                if size eq 13
                  ; jmp .+D; .npad 7; .npad 4
                  DB 0EBH, 0BH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 64H, 24H, 00H
                else
                  if size eq 14
                    ; jmp .+F; .npad 7; .npad 5
                    DB 0EBH, 0CH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 05H, 00H, 00H, 00H, 00H
                  else
                    if size eq 15
                      ; jmp .+F; .npad 7; .npad 6
                      DB 0EBH, 0DH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH, 9BH, 00H, 00H, 00H, 00H
                    else
                      %out error: unsupported npad size
                      .err
                      endif
                    endif
                  endif
                endif
              endif
            endif
          endif
        endif
      endif
    endif
  endif
endif
endm
.2 Einige GCC-Bibliotheks-Funktionen

<table>
<thead>
<tr>
<th>Name</th>
<th>Bedeutung</th>
</tr>
</thead>
<tbody>
<tr>
<td>__divdi3</td>
<td>vorzeichenbehaftete Division</td>
</tr>
<tr>
<td>__moddi3</td>
<td>Rest (Modulo) einer vorzeichenbehafteten Division</td>
</tr>
<tr>
<td>__udivdi3</td>
<td>vorzeichenlose Division</td>
</tr>
<tr>
<td>__umoddi3</td>
<td>Rest (Modulo) einer vorzeichenlosen Division</td>
</tr>
</tbody>
</table>

.3 Einige MSVC-Bibliotheks-Funktionen

ll in Funktionsnamen steht für „long long“, z.B. einen 64-Bit-Datentyp.

<table>
<thead>
<tr>
<th>Name</th>
<th>Bedeutung</th>
</tr>
</thead>
<tbody>
<tr>
<td>__alldiv</td>
<td>vorzeichenbehaftete Division</td>
</tr>
<tr>
<td>__allmul</td>
<td>Multiplikation</td>
</tr>
<tr>
<td>__allrem</td>
<td>Rest einer vorzeichenbehafteten Division</td>
</tr>
<tr>
<td>__allshr</td>
<td>Schiebe links, vorzeichenbehaftet</td>
</tr>
<tr>
<td>__aulldiv</td>
<td>vorzeichenlose Division</td>
</tr>
<tr>
<td>__aullrem</td>
<td>Rest (Modulo) einer vorzeichenlosen Division</td>
</tr>
<tr>
<td>__aullshr</td>
<td>Schiebe rechts, vorzeichenlos</td>
</tr>
</tbody>
</table>

Multiplikation und Links-Schiebebefehle sind sowohl für vorzeichenbehaftete als auch vorzeichenlose Zahlen, da hier für jede Operation nur ein Befehl existiert.

Der Quellcode dieser Funktionen kann im Pfad des installierten MSVS, gefunden werden: VC/crt/src/intel/*.asm.

.4 Cheatsheets

.4.1 IDA

German text placeholder:

<table>
<thead>
<tr>
<th>Taste</th>
<th>Bedeutung</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>Zwischen Quellcode und grafischer Ansicht wechseln</td>
</tr>
<tr>
<td>C</td>
<td>zu Code konvertieren</td>
</tr>
<tr>
<td>D</td>
<td>zu Daten konvertieren</td>
</tr>
<tr>
<td>A</td>
<td>zu Zeichenkette konvertieren</td>
</tr>
<tr>
<td>*</td>
<td>zu Array konvertieren</td>
</tr>
<tr>
<td>U</td>
<td>undefinieren</td>
</tr>
<tr>
<td>O</td>
<td>Offset von Operanden</td>
</tr>
<tr>
<td>H</td>
<td>Dezimalzahl erstellen</td>
</tr>
<tr>
<td>R</td>
<td>Zeichen erstellen</td>
</tr>
<tr>
<td>B</td>
<td>Binärzahl erstellen</td>
</tr>
<tr>
<td>Q</td>
<td>Hexadezimalzahl erstellen</td>
</tr>
<tr>
<td>N</td>
<td>Identifikator umbenennen</td>
</tr>
<tr>
<td>?</td>
<td>Rechner</td>
</tr>
<tr>
<td>G</td>
<td>zu Adresse springen</td>
</tr>
<tr>
<td>:</td>
<td>Kommentar einfügen</td>
</tr>
<tr>
<td>Ctrl-X</td>
<td>Referenz zu aktueller Funktion, Variable, ... zeigen (inkl. lokalem Stack)</td>
</tr>
<tr>
<td>X</td>
<td>Referenz zu Funktion, Variable, ... zeigen</td>
</tr>
<tr>
<td>Alt-I</td>
<td>Konstante suchen</td>
</tr>
<tr>
<td>Ctrl-I</td>
<td>nächsten Auftreten der Konstante suchen</td>
</tr>
<tr>
<td>Alt-B</td>
<td>Byte-Sequenz suchen</td>
</tr>
<tr>
<td>Ctrl-B</td>
<td>nächsten Auftreten der Byte-Sequenz suchen</td>
</tr>
<tr>
<td>Alt-T</td>
<td>Text suchen (inkl. Anweisungen, usw.)</td>
</tr>
<tr>
<td>Ctrl-T</td>
<td>nächstes Auftreten des Textes suchen</td>
</tr>
<tr>
<td>Alt-P</td>
<td>aktuelle Funktion editieren</td>
</tr>
<tr>
<td>Enter</td>
<td>zu Funktion, Variable, ... springen</td>
</tr>
<tr>
<td>Esc</td>
<td>zurückgehen</td>
</tr>
<tr>
<td>Num -</td>
<td>Funktion oder markierten Bereich einklappen</td>
</tr>
<tr>
<td>Num +</td>
<td>Funktion oder Bereich anzeigen</td>
</tr>
</tbody>
</table>
Das Einklappen ist nützlich um Teile von Funktionen zu verstecken, wenn bekannt ist was sie tun. dies wird genutzt imScript um häufig genutzte Inline-Code-Stellen zu verstecken.

### 4.2 OllyDbg

German text placeholder:

<table>
<thead>
<tr>
<th>Tastenkürzel</th>
<th>Bedeutung</th>
</tr>
</thead>
<tbody>
<tr>
<td>F7</td>
<td>Schritt</td>
</tr>
<tr>
<td>F8</td>
<td>German text placeholder</td>
</tr>
<tr>
<td>F9</td>
<td>starten</td>
</tr>
<tr>
<td>Ctrl-F2</td>
<td>Neustart</td>
</tr>
</tbody>
</table>

### 4.3 MSVC

Einige nützliche Optionen die in diesem Buch genutzt werden.

<table>
<thead>
<tr>
<th>Option</th>
<th>Bedeutung</th>
</tr>
</thead>
<tbody>
<tr>
<td>/O1</td>
<td>Speicherplatz minimieren</td>
</tr>
<tr>
<td>/Ob0</td>
<td>Keine Inline-Erweiterung</td>
</tr>
<tr>
<td>/Ox</td>
<td>maximale Optimierung</td>
</tr>
<tr>
<td>/GS-</td>
<td>Sicherheitsüberprüfungen deaktivieren (Buffer Overflows)</td>
</tr>
<tr>
<td>/Fa(file)</td>
<td>Assembler-Quelltext erstellen</td>
</tr>
<tr>
<td>/Zi</td>
<td>Debugging-Informationen erstellen</td>
</tr>
<tr>
<td>/Zp(n)</td>
<td>Strukturen an n-Byte-Grenze ausrichten</td>
</tr>
<tr>
<td>/MD</td>
<td>ausführbare Daten nutzt MSVCR*.DLL</td>
</tr>
</tbody>
</table>

Informationen zu MSVC-Versionen: 5.1.1 on page 419.

### 4.4 GCC

Einige nützliche Optionen die in diesem Buch genutzt werden.

<table>
<thead>
<tr>
<th>Option</th>
<th>Bedeutung</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Os</td>
<td>Optimierung der Code-Größe</td>
</tr>
<tr>
<td>-O3</td>
<td>maximale Optimierung</td>
</tr>
<tr>
<td>-regparm=</td>
<td>Anzahl der in Registern übergebenen Argumente</td>
</tr>
<tr>
<td>-o file</td>
<td>Name der Ausgabedatei</td>
</tr>
<tr>
<td>-g</td>
<td>Debug-Informationen in der ausführbaren Datei erzeugen</td>
</tr>
<tr>
<td>-S</td>
<td>Assembler-Quellcode erstellen</td>
</tr>
<tr>
<td>-masm=intel</td>
<td>Quellcode im Intel-Syntax erstellen</td>
</tr>
<tr>
<td>-fno-inline</td>
<td>keine Inline-Funktionen verwenden</td>
</tr>
</tbody>
</table>

### 4.5 GDB

Einige nützliche Optionen die in diesem Buch genutzt werden:

9GitHub
<table>
<thead>
<tr>
<th>Option</th>
<th>Bedeutung</th>
</tr>
</thead>
<tbody>
<tr>
<td>break filename.c:number</td>
<td>Setzen eines Breakpoints in der angegebenen Zeile</td>
</tr>
<tr>
<td>break function</td>
<td>Setzen eines Breakpoints in der Funktion</td>
</tr>
<tr>
<td>break *address</td>
<td>Setzen eines Breakpoints auf Adresse</td>
</tr>
<tr>
<td>b</td>
<td>——</td>
</tr>
<tr>
<td>p variable</td>
<td>Ausgabe eines Variablenwerts</td>
</tr>
<tr>
<td>run</td>
<td>Starten</td>
</tr>
<tr>
<td>r</td>
<td>——</td>
</tr>
<tr>
<td>cont</td>
<td>Ausführung fortfahren</td>
</tr>
<tr>
<td>c</td>
<td>——</td>
</tr>
<tr>
<td>bt</td>
<td>Stack ausgeben</td>
</tr>
<tr>
<td>set disassembly-flavor intel</td>
<td>Intel-Syntax nutzen</td>
</tr>
<tr>
<td>disas</td>
<td>disassemble current function</td>
</tr>
<tr>
<td>disas function</td>
<td>Funktion disassemblieren</td>
</tr>
<tr>
<td>disas function,+50</td>
<td>disassemble portion</td>
</tr>
<tr>
<td>disas $eip,+0x10</td>
<td>——</td>
</tr>
<tr>
<td>disas/r</td>
<td>mit OpCodes disassemblieren</td>
</tr>
<tr>
<td>info registers</td>
<td>Ausgabe aller Register</td>
</tr>
<tr>
<td>info float</td>
<td>Ausgabe der FPU-Register</td>
</tr>
<tr>
<td>info locals</td>
<td>(bekannte) lokale Variablen ausgeben</td>
</tr>
<tr>
<td>x/w ...</td>
<td>Speicher als 32-Bit-Wort ausgeben</td>
</tr>
<tr>
<td>x/w $rdi</td>
<td>Speicher als 32-Bit-Wort ausgeben an Adresse in RDI</td>
</tr>
<tr>
<td>x/10w ...</td>
<td>10 Speicherworte ausgeben</td>
</tr>
<tr>
<td>x/s ...</td>
<td>Speicher als Zeichenkette ausgeben</td>
</tr>
<tr>
<td>x/i ...</td>
<td>Speicher als Code ausgeben</td>
</tr>
<tr>
<td>x/10c ...</td>
<td>10 Zeichen ausgeben</td>
</tr>
<tr>
<td>x/b ...</td>
<td>Bytes ausgeben</td>
</tr>
<tr>
<td>x/h ...</td>
<td>16-Bit-Halbworte ausgeben</td>
</tr>
<tr>
<td>x/g ...</td>
<td>große (64-Bit-) Worte ausgeben</td>
</tr>
<tr>
<td>finish</td>
<td>bis Funktionsende fortfahren</td>
</tr>
<tr>
<td>next</td>
<td>Nächste Anweisung (nicht in Funktion springen)</td>
</tr>
<tr>
<td>step</td>
<td>Nächste Anweisung (in Funktion springen)</td>
</tr>
<tr>
<td>set step-mode on</td>
<td>Beim schrittweisen Ausführen keine Zeilennummerninfos nutzen</td>
</tr>
<tr>
<td>frame n</td>
<td>Stack-Frame tauschen</td>
</tr>
<tr>
<td>info break</td>
<td>Breakpoints schauen</td>
</tr>
<tr>
<td>del n</td>
<td>Breakpoints löschen</td>
</tr>
<tr>
<td>set args ...</td>
<td>Aufrufparameter setzen</td>
</tr>
</tbody>
</table>
Verwendete Abkürzungen
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS</td>
<td>Betriebssystem</td>
</tr>
<tr>
<td>OOP</td>
<td>Objektorientierte Programmierung</td>
</tr>
<tr>
<td>PS</td>
<td>Programmiersprache</td>
</tr>
<tr>
<td>PRNG</td>
<td>Pseudozufallszahlen-Generator</td>
</tr>
<tr>
<td>ALE</td>
<td>Arithmetisch-logische Einheit</td>
</tr>
<tr>
<td>RA</td>
<td>Rücksprungadresse</td>
</tr>
<tr>
<td>PE</td>
<td>Portable Executable</td>
</tr>
<tr>
<td>SP</td>
<td>Stapel-Zeiger, SP/ESP/RSP (German text placeholder) x86/x64, SP (German text placeholder) ARM</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic-Link Library</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter, IP/EIP/RIP (German text placeholder) x86/64, PC (German text placeholder) ARM</td>
</tr>
<tr>
<td>LR</td>
<td>Link Register</td>
</tr>
<tr>
<td>IDA</td>
<td>Interaktiver Disassembler und Debugger entwickelt von Hex-Rays</td>
</tr>
<tr>
<td>IAT</td>
<td>Import Address Table</td>
</tr>
<tr>
<td>INT</td>
<td>Import Name Table</td>
</tr>
<tr>
<td>RVA</td>
<td>Relative Virtual Address</td>
</tr>
<tr>
<td>VA</td>
<td>Virtual Address</td>
</tr>
<tr>
<td>OEP</td>
<td>Original Entry Point</td>
</tr>
<tr>
<td>MSVC</td>
<td>Microsoft Visual C++</td>
</tr>
<tr>
<td>MSVS</td>
<td>Microsoft Visual Studio</td>
</tr>
<tr>
<td>ASLR</td>
<td>Address Space Layout Randomization</td>
</tr>
<tr>
<td>MFC</td>
<td>Microsoft Foundation Classes</td>
</tr>
<tr>
<td>TLS</td>
<td>Thread Local Storage</td>
</tr>
<tr>
<td>AKAS</td>
<td>German text placeholder</td>
</tr>
<tr>
<td>CRT</td>
<td>C Runtime library</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating-Point Unit</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computing</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>RTTI</td>
<td>Run-Time Type Information</td>
</tr>
<tr>
<td>BSS</td>
<td>Block Started by Symbol</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data</td>
</tr>
<tr>
<td>BSOD</td>
<td>Blue Screen of Death</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management Systems</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>SEH</td>
<td>Structured Exception Handling</td>
</tr>
<tr>
<td>ELF</td>
<td>German text placeholder</td>
</tr>
<tr>
<td>TIB</td>
<td>Thread Information Block</td>
</tr>
<tr>
<td>PIC</td>
<td>Position Independent Code</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>BEQ</td>
<td>(PowerPC, ARM) Branch if Equal</td>
</tr>
<tr>
<td>BNE</td>
<td>(PowerPC, ARM) Branch if Not Equal</td>
</tr>
<tr>
<td>XOR</td>
<td>eXclusive OR</td>
</tr>
<tr>
<td>RAM</td>
<td>Random-Access Memory</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>EGA</td>
<td>Enhanced Graphics Adapter</td>
</tr>
<tr>
<td>VGA</td>
<td>Video Graphics Array</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>ASCIIZ</td>
<td>ASCII Zero ( )</td>
</tr>
<tr>
<td>IA64</td>
<td>Intel Architecture 64 (Itanium)</td>
</tr>
<tr>
<td>EPIC</td>
<td>Explicitly Parallel Instruction Computing</td>
</tr>
</tbody>
</table>
**OOE** Out-of-Order Execution

**STL** (C++) Standard Template Library

**VM** Virtual Memory

**WRK** Windows Research Kernel

**GPR** General Purpose Registers

**SSDT** System Service Dispatch Table

**RE** Reverse Engineering

**BOM** Byte Order Mark

**GDB** GNU Debugger

**FP** Frame Pointer

**MBR** Master Boot Record

**JPE** Jump Parity Even

**STMFD** Store Multiple Full Descending

**LDMFD** Load Multiple Full Descending

**STMED** Store Multiple Empty Descending

**LDMED** Load Multiple Empty Descending

**STMFA** Store Multiple Full Ascending

**LDMFA** Load Multiple Full Ascending

**STMEA** Store Multiple Empty Ascending

**LDMEA** Load Multiple Empty Ascending

**APSR** (ARM) Application Program Status Register

**FPSCR** (ARM) Floating-Point Status and Control Register

**RFC** Request for Comments

**JVM** Java Virtual Machine

**JIT** Just-In-Time compilation

**CDFS** Compact Disc File System
**CD**  Compact Disc

**ADC**  Analog-to-Digital Converter ............................................................ 439

**EOF**  End of File  German text placeholder ............................................... 70

**DBI**  Dynamic Binary Instrumentation ..................................................... 421

**URL**  Uniform Resource Locator ............................................................. 428
Glossar

**German text placeholder** German text placeholder. 445

**German text placeholder** German text placeholder. 210, 232, 515

**German text placeholder** German text placeholder. 417, 472

**German text placeholder** German text placeholder. 445

**German text placeholder** German text placeholder. 268, 378, 460, 471, 501, 537

**German text placeholder** German text placeholder. 31, 61

**German text placeholder** German text placeholder. 502

**German text placeholder** German text placeholder. 516, 517

**German text placeholder** German text placeholder. 537

**German text placeholder** German text placeholder. 164, 180, 401, 445, 532

**German text placeholder** German text placeholder. 21, 63, 320

**German text placeholder** German text placeholder. 29, 469, 470

**German text placeholder** German text placeholder. 16, 164, 167, 180, 401, 532

**German text placeholder** German text placeholder. 79, 114, 532

**German text placeholder** German text placeholder. 28, 31

**German text placeholder** German text placeholder. 31

**German text placeholder** German text placeholder. 166

**German text placeholder** German text placeholder. 418, 419

**German text placeholder** German text placeholder. 426, 427

**German text placeholder** German text placeholder. 194, 197, 199, 200, 204

**German text placeholder** German text placeholder. 194

**German text placeholder** German text placeholder. 179, 281, 381

**German text placeholder** German text placeholder. 537

**German text placeholder** German text placeholder. 52, 81, 82, 492

**German text placeholder** German text placeholder. 21, 34, 135

**German text placeholder** German text placeholder. 22

**German text placeholder** German text placeholder. 168–170, 421, 432, 435, 436, 488, 497, 510

**German text placeholder** German text placeholder. 409

**German text placeholder** German text placeholder. 492

callee aufgerufene Funktion. 31, 32, 45, 80, 83, 85, 447–450, 452, 453
caller aufrufende Funktion. 5, 6, 10, 45, 80–82, 84, 135, 447–450, 453

GiB German text placeholder. 15

Produkt Ergebnis einer Multiplikation. 201, 204, 408, 409
Stapel-Zeiger Ein Register das auf eine Stelle im Stack zeigt. 9, 10, 29, 34, 41, 58, 83, 447-450, 527, 547
Index

.NET, 477
German text placeholder, 170, 239, 417, 432, 435
German text placeholder, 11, 18, 30, 36, 128, 134, 250, 257, 277, 289, 306, 492, 511
alloca(), 34, 261, 483
assert(), 267, 429
close(), 464
localtime_r(), 329
longjmp(), 135
malloc(), 322
memchr(), 534
memcmp(), 12, 51, 534
memset(), 242, 535
open(), 464
pow(), 206
puts(), 21
rand(), 312, 420
read(), 464
scanf(), 50
strcmp(), 465
strcpy(), 12
strlen(), 178, 377, 534
strstr(), 411

German text placeholder, 21
0x0BADF00D, 61
0xC0C0C0C0C0, 61

Ada, 90
Alpha AXP, 2
AMD, 452
Angry Birds, 238, 239
Apollo Guidance Computer, 188
ARM, 186

German text placeholder

ADC, 360
ADD, 21, 89, 117, 171, 295, 307
ADDAI, 117
ADDC, 154
ADDS, 87, 360
ADR, 19, 117
ADRS, 117, 143
ADRP/ADD pair, 23, 67, 265, 278, 403
APSR, 237
ASR, 310
ASRS, 289
B, 117, 118
Bcc, 80, 128
BCS, 118, 240
BEQ, 143
BGE, 118
BIC, 289, 294, 312
BL, 19–22, 24, 117, 404
BLX, 21, 22
localtime_r(), 329
longjmp(), 135
malloc(), 322
memchr(), 534
memcmp(), 12, 51, 534
memset(), 242, 535
open(), 464
pow(), 206
puts(), 21
rand(), 312, 420
read(), 464
scanf(), 50
strcmp(), 465
strcpy(), 12
strlen(), 178, 377, 534
strstr(), 411

0x0BADF00D, 61
0xC0C0C0C0C0, 61

German text placeholder

BLE, 118
BLS, 118
BLT, 171
BLX, 21, 22
BNE, 118
BX, 87, 155
CMP, 117, 143, 154, 171, 307
CSEL, 126, 130, 132, 308
EOR, 294
FCMPE, 240
FCSEL, 240
FMOV, 402
FMRS, 295
FPSCR, 237
IT, 133, 238, 261
LDMeacFD, 117
LDMEA, 29
LDMED, 29
LDMFA, 29
LDMFD, 19, 29, 117
LDP, 24
LDR, 58, 66, 247, 264, 400
LDRB, 338
LDRB.W, 186
LDRSB, 186
Link Register, 19, 31, 156
LSL, 307, 310
LSL.W, 307
LSLS, 248, 294
LSR, 310
LSRS, 294
MADD, 87
MLA, 86, 87
MOV, 7, 19, 20, 307
MOVcc, 128, 132
MOVK, 402
MOVT, 21
MOVT.W, 22
MOVW, 22
MUL, 89
MULS, 87
MVNS, 187
ORR, 289
POP, 18–20, 29, 31
PUSH, 20, 29, 31
RET, 24
RSB, 122, 274, 307
SBC, 360
scratch registers, 186
STMEAC, 29
STMEAD, 29
STMFA, 29
STMFD, 18, 29
80486, 195
FPU, 195
Intel C++, 9, 368, 511, 516, 533
iPod/iPhone/iPad, 18
Itanium, 513
Java, 412
jumptable, 147, 155
Keil, 18
kernel panic, 460
kernel space, 460
LD_PRELOAD, 464
Linker, 282, 461
lib.so.6, 280
LLDB, 504
LLVM, 18
long double, 196
Loop unwinding, 166
Mac OS X, 505
MDS, 430
MFC, 474
MIDI, 430
MinGW, 418
minifloat, 402
MIPS, 2, 439, 472

German text placeholder, 24
ADD, 90
ADDIU, 25, 69, 70
ADDU, 90
AND, 290
BC1F, 242
BC1T, 242
BEQ, 80, 119
BEQZ, 121
BLTZ, 123
BNE, 119
BNEZ, 157
C.LT.D, 242
FCCR, 241
J, 6, 8, 26
JAL, 90
JALR, 25, 90
JR, 146
LA, 28
LB, 176
LBU, 176
LI, 8, 404
LUI, 25, 69, 70, 293, 405
LW, 25, 59, 70, 146, 405
MFHI, 89
MFLO, 89
MOVE, 25, 68
MTC1, 354
MULT, 89
NEGU, 123
NOP, 28, 68
NOR, 188
NOT, 188
OR, 28
ORI, 290, 404
SB, 176

SLL, 157, 190, 309
SLLV, 309
SLT, 119
SLTIU, 157
SLTU, 119, 121, 157
SRL, 195
SUBU, 123
Branch delay slot, 8
Global Pointer, 274
Load delay slot, 146
O32, 50
MS-DOS, 13, 33, 259, 430, 436, 445, 470, 515, 525, 534, 537
DOS extenders, 516
MSVC, 543, 544

Native API, 471
Non-a-numbers (NaNs), 232
objdump, 352, 463, 477, 503
OEP, 470, 477
OpenMP, 420
OpenWatcom, 418, 449
Oracle RDBMS, 9, 367, 428, 480, 511, 516
Pascal, 423
PDP-11, 400
PGP, 427
Phrack, 427
positionsabhängiger Code, 19, 461
PowerPC, 2, 25
puts() anstatt printf(), 21
puts() anstelle von printf(), 115
puts() instead of printf(), 56
Quake III Arena, 357

rada.re, 13
Radare, 504
rafind2, 503
RAM, 66
Raspberry Pi, 18
ReactOS, 486
Register allocation, 381
Relocation, 22
Reverse Polish notation, 242
RISC pipeline, 117
ROM, 66
row-major order, 269
RSA, 5
RVA, 470
SAP, 417
Scratch space, 451
Security cookie, 258, 492
Security through obscurity, 429
Seite (Speicher), 378
Shadow space, 84, 85, 388
Shellcode, 460, 471, 540
Signed numbers, 106, 407
SIMD, 387
SSE, 387
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE2</td>
<td>387</td>
</tr>
<tr>
<td>Stack</td>
<td>29, 80, 135</td>
</tr>
<tr>
<td>German text placeholder, 30</td>
<td></td>
</tr>
<tr>
<td>Stack frame</td>
<td>52</td>
</tr>
<tr>
<td>stdcall</td>
<td>447, 510</td>
</tr>
<tr>
<td>strace</td>
<td>464, 504</td>
</tr>
<tr>
<td>Stuxnet</td>
<td>431</td>
</tr>
<tr>
<td>syscall</td>
<td>280, 460, 504</td>
</tr>
<tr>
<td>Sysinternals</td>
<td>428, 505</td>
</tr>
<tr>
<td>thiscall</td>
<td>450</td>
</tr>
<tr>
<td>thunk-functions</td>
<td>22, 476</td>
</tr>
<tr>
<td>TLS</td>
<td>259, 455, 472, 477, 528</td>
</tr>
<tr>
<td>Callbacks</td>
<td>458, 477</td>
</tr>
<tr>
<td>Tor</td>
<td>427</td>
</tr>
<tr>
<td>tracer</td>
<td>168, 421, 432, 435, 488, 497, 504, 510</td>
</tr>
<tr>
<td>UFS2</td>
<td>430</td>
</tr>
<tr>
<td>Unicode</td>
<td>423</td>
</tr>
<tr>
<td>UNIX</td>
<td></td>
</tr>
<tr>
<td>chmod</td>
<td>4</td>
</tr>
<tr>
<td>grep</td>
<td>428, 510</td>
</tr>
<tr>
<td>od</td>
<td>503</td>
</tr>
<tr>
<td>strings</td>
<td>427, 503</td>
</tr>
<tr>
<td>xxd</td>
<td>503</td>
</tr>
<tr>
<td>Unrolled loop</td>
<td>172, 261</td>
</tr>
<tr>
<td>uptime</td>
<td>464</td>
</tr>
<tr>
<td>UseNet</td>
<td>427</td>
</tr>
<tr>
<td>user space</td>
<td>460</td>
</tr>
<tr>
<td>UTF-16LE</td>
<td>423, 424</td>
</tr>
<tr>
<td>UTF-8</td>
<td>423</td>
</tr>
<tr>
<td>Unencoding</td>
<td>427</td>
</tr>
<tr>
<td>VA</td>
<td>470</td>
</tr>
<tr>
<td>Watcom</td>
<td>418</td>
</tr>
<tr>
<td>WinDbg</td>
<td>504</td>
</tr>
<tr>
<td>Windows</td>
<td>501</td>
</tr>
<tr>
<td>API</td>
<td>525</td>
</tr>
<tr>
<td>IAT</td>
<td>470</td>
</tr>
<tr>
<td>INT</td>
<td>470</td>
</tr>
<tr>
<td>KERNEL32.DLL</td>
<td>279</td>
</tr>
<tr>
<td>PDB</td>
<td>417, 472</td>
</tr>
<tr>
<td>Structured Exception Handling</td>
<td>36, 478</td>
</tr>
<tr>
<td>TIB</td>
<td>259, 478, 528</td>
</tr>
<tr>
<td>Win32</td>
<td>279, 424, 464, 470, 516</td>
</tr>
<tr>
<td>GetProcAddress</td>
<td>476</td>
</tr>
<tr>
<td>LoadLibrary</td>
<td>476</td>
</tr>
<tr>
<td>MulDiv()</td>
<td>409</td>
</tr>
<tr>
<td>Ordinal</td>
<td>474</td>
</tr>
<tr>
<td>RaiseException()</td>
<td>478</td>
</tr>
<tr>
<td>SetUnhandledExceptionFilter()</td>
<td>480</td>
</tr>
<tr>
<td>Windows 2000</td>
<td>471</td>
</tr>
<tr>
<td>Windows 3.x</td>
<td>516</td>
</tr>
<tr>
<td>Windows NT4</td>
<td>471</td>
</tr>
<tr>
<td>Windows Vista</td>
<td>470</td>
</tr>
<tr>
<td>Windows XP</td>
<td>471, 477</td>
</tr>
<tr>
<td>Wine</td>
<td>486</td>
</tr>
<tr>
<td>x86</td>
<td></td>
</tr>
<tr>
<td>German text placeholder</td>
<td></td>
</tr>
<tr>
<td>German text placeholder, 71, 108, 528</td>
<td></td>
</tr>
<tr>
<td>AAA</td>
<td>541</td>
</tr>
<tr>
<td>AAS</td>
<td>541</td>
</tr>
<tr>
<td>ADC</td>
<td>359, 532</td>
</tr>
<tr>
<td>ADD</td>
<td>9, 41, 81, 532</td>
</tr>
<tr>
<td>ADDSD</td>
<td>388</td>
</tr>
<tr>
<td>ADDSS</td>
<td>400</td>
</tr>
<tr>
<td>ADRcc</td>
<td>125</td>
</tr>
<tr>
<td>AH</td>
<td>533, 534</td>
</tr>
<tr>
<td>AND</td>
<td>10, 280, 283, 296, 310, 345, 532, 535</td>
</tr>
<tr>
<td>BFS</td>
<td>380, 535</td>
</tr>
<tr>
<td>BSR</td>
<td>535</td>
</tr>
<tr>
<td>BSWAP</td>
<td>535</td>
</tr>
<tr>
<td>BT</td>
<td>535</td>
</tr>
<tr>
<td>BTC</td>
<td>292, 535</td>
</tr>
<tr>
<td>BTR</td>
<td>292, 502, 535</td>
</tr>
<tr>
<td>BTS</td>
<td>292, 535</td>
</tr>
<tr>
<td>CALL</td>
<td>9, 30, 475, 532</td>
</tr>
<tr>
<td>CBW</td>
<td>408, 535</td>
</tr>
<tr>
<td>CDQ</td>
<td>366, 408, 535</td>
</tr>
<tr>
<td>CDQE</td>
<td>408, 535</td>
</tr>
<tr>
<td>CF</td>
<td>33, 532, 534, 535, 538</td>
</tr>
<tr>
<td>CLD</td>
<td>535</td>
</tr>
<tr>
<td>CLI</td>
<td>535</td>
</tr>
<tr>
<td>CMC</td>
<td>535</td>
</tr>
<tr>
<td>CMOVc</td>
<td>117, 125, 126, 129, 132, 535</td>
</tr>
<tr>
<td>CMP</td>
<td>71, 532, 541</td>
</tr>
<tr>
<td>CMP</td>
<td>535</td>
</tr>
<tr>
<td>CMPSB</td>
<td>430, 535</td>
</tr>
<tr>
<td>CMPSD</td>
<td>535</td>
</tr>
<tr>
<td>CMPSQ</td>
<td>535</td>
</tr>
<tr>
<td>COMSID</td>
<td>396</td>
</tr>
<tr>
<td>COMISS</td>
<td>400</td>
</tr>
<tr>
<td>CPUID</td>
<td>343, 537</td>
</tr>
<tr>
<td>CS</td>
<td>515</td>
</tr>
<tr>
<td>CWD</td>
<td>408, 535</td>
</tr>
<tr>
<td>CWDE</td>
<td>408, 535</td>
</tr>
<tr>
<td>DEC</td>
<td>180, 532, 541</td>
</tr>
<tr>
<td>DF</td>
<td>535, 538</td>
</tr>
<tr>
<td>DIV</td>
<td>408, 537</td>
</tr>
<tr>
<td>DIVSD</td>
<td>388, 434</td>
</tr>
<tr>
<td>DR6</td>
<td>531</td>
</tr>
<tr>
<td>DR7</td>
<td>531</td>
</tr>
<tr>
<td>DS</td>
<td>515</td>
</tr>
<tr>
<td>EAX</td>
<td>71</td>
</tr>
<tr>
<td>EBP</td>
<td>52, 81</td>
</tr>
<tr>
<td>ES</td>
<td>515</td>
</tr>
<tr>
<td>ESP</td>
<td>41, 52</td>
</tr>
<tr>
<td>FABS</td>
<td>538</td>
</tr>
<tr>
<td>FADD</td>
<td>538</td>
</tr>
<tr>
<td>FADDP</td>
<td>197, 203, 538</td>
</tr>
<tr>
<td>FATRET</td>
<td>305, 306</td>
</tr>
<tr>
<td>FCHS</td>
<td>539</td>
</tr>
<tr>
<td>FCMOVc</td>
<td>234</td>
</tr>
<tr>
<td>FCOM</td>
<td>221, 232, 539</td>
</tr>
<tr>
<td>FCOMP</td>
<td>209, 539</td>
</tr>
<tr>
<td>FCOMPP</td>
<td>539</td>
</tr>
<tr>
<td>FDIV</td>
<td>197, 432, 433, 539</td>
</tr>
<tr>
<td>FDIVP</td>
<td>197, 539</td>
</tr>
<tr>
<td>FDIVR</td>
<td>203, 539</td>
</tr>
<tr>
<td>FDIVRP</td>
<td>539</td>
</tr>
<tr>
<td>FILD</td>
<td>539</td>
</tr>
<tr>
<td>FIST</td>
<td>539</td>
</tr>
<tr>
<td>FISTP</td>
<td>539</td>
</tr>
<tr>
<td>FLDS</td>
<td>207, 209, 539</td>
</tr>
<tr>
<td>FLDD</td>
<td>539</td>
</tr>
<tr>
<td>FLDCW</td>
<td>539</td>
</tr>
<tr>
<td>FLDO</td>
<td>539</td>
</tr>
<tr>
<td>FLDSZ</td>
<td>539</td>
</tr>
</tbody>
</table>
FMUL, 197, 539
FMULP, 539
FNSTCW, 539
FNSTSW, 209, 232, 539
FS, 457
FSCALE, 355
FSINCOS, 539
FSQRT, 539
FST, 539
FSTCW, 539
FSTP, 207, 539
FSTSW, 539
FSUB, 539
FSUBP, 539
FSUBR, 539
FSUBRP, 539
FUCOM, 232, 539
FUCOMI, 234
FUCOMP, 539
FUCOMPP, 232, 539
FWAIT, 195
FXCH, 512, 540
GS, 259, 457, 460
IDIV, 408, 537
IF, 535, 538
IMUL, 81, 276, 408, 532, 541
IN, 537
INC, 180, 510, 532, 541
INT, 33, 537
INT3, 421
IRET, 537
JA, 106, 233, 407, 532, 541
JAE, 106, 532, 541
JB, 106, 407, 532, 541
JBE, 106, 532, 541
JC, 532
Jcc, 80, 128
JCXZ, 532
JE, 135, 532, 541
JE CXZ, 532
JG, 106, 407, 532
JGE, 106, 532
JL, 106, 407, 532
JLE, 106, 532
JMP, 30, 153, 476, 510, 532
JNA, 532
JNAE, 532
JNB, 532
JNBE, 233, 532
JNC, 532
JNE, 71, 532, 541
JNG, 532
JNGE, 532
JNL, 532
JNLE, 532
JNO, 532, 541
JNS, 532, 541
JNZ, 532
JO, 532, 541
JP, 210, 532, 541
JPO, 532
JRCXZ, 532
JS, 532, 541
JZ, 135, 511, 532
LAHF, 533
LEA, 52, 83, 325, 452, 533
LEAVE, 10, 533
LOCK, 501, 502, 532
LOOP, 163, 177, 434, 537
MAXSD, 396
MOV, 7, 10, 12, 473, 510, 534
MOVQ, 370
MOVQ, 370
MOVSB, 534
MOVS, 395, 534
MOVSDX, 395
MOVQ, 534
MOVSQ, 400
MOVSW, 534
MOV, 179, 186, 337–339, 408, 534
MOVSD, 263
MOVZX, 180, 322, 534
MUL, 408, 534
MULSD, 388
NEG, 534
NOP, 510, 534, 541
NOT, 185, 187, 534
OR, 283, 534
OUT, 537
PADD, 370
PCMPGTQ, 379
PLMULH, 367
PLMULLD, 367
PMOVMSKB, 379
POP, 9, 29, 30, 534, 541
POPA, 537, 541
POPCNT, 537
POPF, 537
PUSH, 9, 10, 29, 30, 52, 534, 541
PUSHA, 537, 541
PUSHF, 537
PXOR, 379
RCL, 434, 538
RCR, 538
REP, 532, 534, 535
REPE/REPNE, 532
REPN, 534
REPN, 534
RETP, 5, 7, 10, 30, 258, 510, 534
RIP, 463
ROL, 306, 511, 538
ROR, 511, 538
SAHF, 232, 534
SAL, 538
SAR, 310, 408, 538
SBB, 359, 534
SCASB, 534
SCASD, 534
SCASQ, 534
SCASW, 534
SETcc, 119, 180, 233, 538
SHL, 190, 244, 310, 534
SHR, 194, 310, 345, 534
SHRD, 365, 535
SS, 515
STC, 538
STD, 538
STI, 538
STOSB, 535
STOSD, 535
STOSQ, 535
STOSW, 535
SUB, 10, 71, 135, 532, 535
SYSCALL, 537, 538
SYSENTER, 461, 537, 538
TEST, 179, 280, 283, 310, 535
UD2, 538
XADD, 502
XCHG, 534, 538
XOR, 10, 71, 185, 434, 510, 535, 541
ZF, 71, 280
AVX, 367
FPU, 529
MMX, 366
SSE, 367
SSE2, 367
x86-64, 14, 51, 57, 83, 380, 387, 450, 463, 525, 531
Xcode, 18
XML, 427