Übungsblatt 10

Gesundes neues Jahr – wünscht das Übersetzerbau-Team ☺

Wichtig: Abgabe des vierten Meilensteins

Bitte denken Sie daran, Ihre Implementierung des vierten Meilensteins bis spätestens **14.01.** fertigzustellen und die entsprechend markierte Revision auf den `git`-Server des Lehrstuhls zu pushen.

Dokumentation zur `x86-64`-Architektur

Auf der Internetseite zur Übung finden Sie einen Verweis auf die Dokumentation zur `x86-64`-Architektur.

Aufgabe 10.1: `x86-64`: Fibonacci (iterativ), Assemblierer `as`, Binder `ld`, `objdump`

In dieser Aufgabe implementieren Sie ein erstes einfaches Programm in `x86-64`-Assembler und bringen dieses zur Ausführung auf einem Linux-System.

a) Auf der Internetseite zur Übung finden Sie ein Grundgerüst für das Programm, das aus zwei Dateien `fib.s` und `exit.s` besteht. Erweitern Sie das Programm so, dass es die `n`-te Fibonacci-Zahl *iterativ* berechnet.

b) Assemblieren Sie die beiden Module jeweils mittels `as`.

c) Binden Sie die erzeugten Objektmodule mittels `ld` zu einem ausführbaren Programm.

d) Führen Sie das Programm aus. Das Ergebnis der Berechnung wird als *Exit-Code* an das Betriebssystem zurückgegeben. Sie können sich diesen am Anschluss an die Ausführung mittels „`echo $?`“ in der Kommandozeile ausgeben lassen. Überprüfen Sie das Ergebnis.

e) Verwenden Sie das Werkzeug `objdump`, um das Programm vor und nach dem Binden zu disassemblieren. Worin unterscheiden sich die Programme?

Aufgabe 10.2: Debugger `gdb`

Im Rahmen dieser Aufgabe machen Sie sich mit dem Debugger `gdb` vertraut. Laden Sie sich dazu zunächst das Assembler-Programm `fib_fail.s` von der Internetseite zur Übung herunter und versuchen Sie, das Programm zu verstehen. Erzeugen Sie anschließend mittels `as` und `ld` ein ausführbares Programm.

a) Die Funktion `fib` liefert falsche Werte zurück. Wie können Sie dies feststellen, ohne das Programm zu verändern (d.h. insbesondere ohne Ausgabeanweisungen hinzuzufügen)?

b) Verwenden Sie `gdb`, um die fehlerhaften Stellen zu identifizieren. Setzen Sie dazu *Breakpoints* an „interessanten“ Stellen, um die Ausführung des Programms an diesen zu unterbrechen und um sich die zu diesem Zeitpunkt berechneten Zwischenergebnisse ansehen zu können.

c) Beheben Sie die identifizierten Fehler und übersetzen Sie das Programm erneut. Testen Sie ihre verbesserte Version erneut mittels `gdb`.

Hinweise zur Verwendung von `gdb`:

- Debugger starten:
  - `gdb <Programm>`: startet den Debugger für das angegebene `<Programm>`
  - `run`: führt das Programm bis zum ersten *Breakpoint* (s.u.) aus
  - `file`: lädt das Programm (bspw. nach einer erneuten Übersetzung) neu, behält *Breakpoints*

- Disassemblieren:
  - `disassemble`: zeigt die nachfolgenden Assembler-Instruktionen ab der aktuellen Instruktion
  - `disassemble <Funktion>`: zeigt die Assembler-Instruktionen der angegebenen `<Funktion>`
- *Breakpoints* setzen:
  - `break <Funktion>`: setzt einen *Breakpoint* am Beginn der angegebenen `<Funktion>`
  - `break *<Adresse>`: setzt einen *Breakpoint* an der `<Adresse>` (siehe Ausgabe von `disassemble`)
- *Breakpoints* verwalten:
  - `info breakpoints`: zeigt die gesetzten *Breakpoints* an
  - `delete <Nummer>`: entfernt den *Breakpoint* mit der angegebenen `<Nummer>` wieder
- Steuerung der Programmausführung nach Erreichen eines *Breakpoints*:
  - `continue`: führt das Programm bis zum nächsten *Breakpoint* aus
  - `quit`: beendet die Programmausführung
  - `si`: führt die nächste Instruktion aus und pausiert die Ausführung nach dieser erneut; steigt bei einer `call`-Instruktion in die aufgerufene Funktion ab
  - `ni`: wie `si`, aber steigt *nicht* in aufgerufene Funktionen ab
- Register/Speicher lesen und schreiben:
  - mittels `info`:
    - `info registers <Reg>`: zeigt den Inhalt des Registers `<Reg>` (z.B. `$rax`) an
    - `info registers`: zeigt den Inhalt aller Register an
  - mittels `print`:
    - `print <Ziel>`: gibt Wert an der Stelle `<Ziel>` (Register oder Adresse) aus
    - `print <Ziel> = <Wert>`: setzt Wert an der Stelle `<Ziel>` auf den angegebenen `<Wert>`
    - Beispiele:
      - `print $rax = 0x1303`: setzt den Wert von Register `$rax` auf `0x1303`
      - `print *(long long *)($rsp+4*8)`: gibt das fünfte Wort (Offset `4` mal Wortlänge `8`) auf dem Stack aus (beachten Sie die C-Syntax!)
  - mittels `x` (nur für Speicher, vermeidet Typkonvertierung von `print`):
    - `x/<Format> <Ziel>`: gibt den oder die Werte an der Stelle `<Ziel>` im angegebenen `<Format>` aus; `<Format>` besteht dabei aus der *Anzahl* der auszugebenden Werte, dem *Format*, in dem diese ausgegeben werden sollen, sowie der *Größe* eines einzelnen Werts (Tipp: `help x`, siehe unten)
    - Beispiele:
      - `x/3i 0x1234`: gibt die nächsten drei Instruktionen (`i`) ab Adresse `0x1234` aus
      - `x/g $rsp+(4*8)`: gibt das fünfte Wort auf dem Stack aus (`g`: *giant*, 8 Bytes)
- Sonstiges:
  - `backtrace`: zeigt einen *Stacktrace* der aufgerufenen Funktionen (setzt *Framepointer* voraus)
  - `watch <Reg>`: pausiert die Ausführung, wenn sich der Wert in dem angegebenen Register ändert
  - `display/<Format> <Ziel>`: gibt den Wert eines Registers bzw. einer Speicherstelle bei *jeder* Pausierung der Programmausführung aus (`<Format>` und `<Ziel>` wie bei `x`, siehe oben)
    - Beispiel: `display/i $pc`: zeigt die aktuelle Instruktion bei Erreichen eines *Breakpoints* (`$pc`: *Program Counter*, Adresse des aktuellen Befehls)
  - `help <Kommando>`: zeigt eine kurze Hilfe zu dem angegebenen `<Kommando>`

Aufgabe 10.3: x86-64/Linux: Systemaufrufe

Schreiben Sie eine Assembler-Routine, die unter x86-64/Linux ein Zeichen auf der Standardausgabe ausgibt.

Tipps:

- „man 2 intro“ und „man 2 syscall“ liefern allgemeine Informationen zu Systemaufrufen unter Linux.
- „man 2 write“ beinhaltet Informationen zum write-Systemaufruf.
- Die im Linux-Header sys/syscall.h inkludierten Header-Dateien beinhalten symbolische Konstanten für die Nummern der Systemaufrufe. Der Dateipfad kann mittels „locate sys/syscall.h“ bestimmt werden.

Aufgabe 10.4: x86-64: Aufrufkonventionen

Implementieren Sie ein x86-64-Programm, das mit Hilfe einer *rekursiven* Routine den Binomialkoeffizienten $\binom{n}{k}$ zweier beliebiger natürlicher Zahlen n, k, $n \geq k$, berechnet und als *Exit-Code* zurückgibt:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad \binom{n}{0} = \binom{n}{n} = 1$$

Überlegen Sie sich dabei eine geeignete *Aufrufkonvention* (*Calling Convention*).

Projektübung 6 (Abgabe bis 21.01.2018)

Im Rahmen dieser Projektübung implementieren Sie die *Speicherplatzvergabe* in Ihrem e2-Übersetzer. Dabei werden die Variablen und virtuellen Register des IR-Programms mit der Information über den jeweiligen Speicherplatz attributiert. Diese Information wird später bei der Code-Generierung benötigt.

Hinweis: Ab dieser Aufgabe muss Ihr Übersetzer *keine* Unterstützung für Fließkommawerte und -operationen mehr haben. Sie *dürfen* dies jedoch selbstverständlich implementieren ☺

a) Da auch für Parameter Speicherplätze vergeben werden müssen, müssen Sie sich eine geeignete Aufrufkonvention überlegen, die Ihr Übersetzer verwenden soll. Sie *können* sich an dem *System V ABI* für x86-64/Linux[1] orientieren, aber Sie dürfen sich auch eine „eigene“ Aufrufkonvention überlegen.

Tipp: Die Speicherplatzvergabe und insbesondere die Code-Generierung (nächste Projektübung) werden an einigen Stellen einfacher (und weniger „fehleranfällig“), wenn alle Argumente über den Stack übergeben werden. Dies führt jedoch i.A. auch zu einer geringeren Effizienz...

b) Machen Sie sich mit den vorgegebenen Klassen in dem Paket e2c.backend.memory zur Repräsentation von Speicherstellen (mit der gemeinsamen Oberklasse MemoryLocation) vertraut:

- Globale Variablen liegen an statischen Adressen, die im Assembler-Code über symbolische Namen adressiert werden können. Diese werden durch Instanzen der Klasse StaticLocation repräsentiert.
- Eine Instanz der Klasse HardwareRegister repräsentiert ein physikalisches Register (z.B. %rax).
- Werte auf dem Stack sollen mit Hilfe eines Offsets relativ zum *Framepointer* adressiert werden. Ein solcher Speicherplatz soll durch eine Instanz der Klasse StackFrameLocation repräsentiert werden.

c) Die Vorlage stellt außerdem das Interface e2c.backend.memory.assignment.MemoryLocationAssigner zur Verfügung. Klassen, die dieses Interface implementieren, stellen mit assignMemoryLocations() eine Methode zur Vergabe von Speicherplätzen bereit, und zwar *unabhängig* von der Zielarchitektur (damit kann derselbe Algorithmus zur Speicherplatzvergabe für unterschiedliche Backends verwendet werden). Zu diesem Zweck bekommt die Methode eine Instanz der Klasse TargetInfo übergeben, die u.a. die auf der tatsächlichen Zielarchitektur verfügbaren Register beschreibt.

Sie haben im Rahmen dieser Projektübung zwei Möglichkeiten:

[1] siehe bspw. https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

- Ihre Speicherplatzvergabe implementiert das Interface `MemoryLocationAssigner`. In diesem Fall müssen Sie auch eine Unterklasse von `TargetInfo` für die `x86-64`-Architektur und die von Ihnen gewähle Aufrufkonvention implementieren (bspw. in `e2c.backend.codegen.x64.TargetInfoX64`).
- Sie ignorieren die Interfaces `MemoryLocationAssigner` und `TargetInfo` und implementieren die Speicherplatzvergabe fest für die `x86-64`-Architektur.

d) Implementieren Sie die eigentliche Speicherplatzvergabe. Dabei muss jeder `IRVariable` des Programms genau ein Speicherplatz zugewiesen werden (Methode `setMemoryLocation()`). Des Weiteren sollten Sie sich für die spätere Code-Generierung für jede Funktion des Eingabeprogramms merken, wie groß der jeweilige Kellerrahmen ist (d.h. wie viel Platz zu Beginn der Funktion auf dem Stack reserviert werden muss, Methode `setStackFrameSize()`). Die Klasse `IRFunction` erlaubt außerdem die Speicherung der innerhalb der jeweiligen Funktion verwendeten Caller-Save- und Callee-Save-Register. Dies ist u.U. bei der späteren Code-Generierung hilfreich, um die „richtigen“ Register zu sichern bzw. zu restaurieren.

Es ist Ihnen überlassen, wie „gut“ Sie die Speicherplatzvergabe implementieren; je nach Implementierung unterscheiden sich die Effizienz und der Speicherverbrauch des am Ende generierten Programms stark...

Tipp 1: Die naive (und im Rahmen der Projektübungen völlig ausreichende) Lösung platziert alle lokalen Variablen und virtuellen Register auf dem Stack (es werden also keine Register verwendet; dies macht insbesondere auch die spätere Code-Generierung einfacher und weniger „fehleranfällig“). Dabei erhält jede Variable und jedes virtuelle Register einen *eigenen Speicherplatz*[2] im Kellerrahmen. Die Implementierung der gesamten Projektübung sollte in diesem Fall in ca. 100 Zeilen Java-Code machbar sein... ☺

Tipp 2: Sollten Sie eine „bessere“ Speicherplatzvergabe implementieren, die Gebrauch von physikalischen Registern macht, sollten Sie mindestens zwei Register „freilassen“ (sog. *Scratch Register*). Diese können Sie später bei der Code-Generierung für die Speicherung von Zwischenergebnissen o.ä. verwenden, ohne diese sichern und restaurieren zu müssen.

Hinweis: Es gibt *keine* Testfälle zu dieser Aufgabe. Es kann durchaus sein, dass Sie bei der Implementierung der Code-Generierung in der nächsten Projektübung feststellen, dass Ihre Speicherplatzvergabe fehlerhaft ist...

Tipp: Testen Sie Ihre Implementierung selbst, indem Sie diese für einige Testprogramme geeignet aufrufen und sich den zugewiesenen Speicherplatz für jede Variable des Eingabeprogramms ausgeben lassen.

[2] Achtung: Die Kellerrahmen werden dabei je nach Funktion u.U. *sehr* groß.