

Prof. L. Thiele

## Technische Informatik 1 - HS 2011

### Lösungsvorschläge für Übung 3

Datum: 27.10.2011

#### Einleitung

Das Ziel dieser Übung ist, den Compilations-Vorgang und insbesondere den Aufbau von Objekt-Dateien kennenzulernen.

Die Übung findet auf einem Linux Rechner mit einer Intel x86 Architektur statt. Um auf diesem Rechner MIPS Programme zu compilieren, verwenden wir einen sogenannten *Cross-Compiler*, der Code für einen fremden Instruktionssatz generieren kann. Die generierten Applikationen können dann auf einem MIPS Prozessor oder auf einem MIPS Prozessor Emulator ausgeführt werden.

Um die Cross-Compilation-Tools für die MIPS Architektur vom Compiler für die x86 Architektur (so genannter Host-Compiler) unterscheiden zu können, tragen alle MIPS Tools einen *mips-elf*-Präfix. Der MIPS-C-Compiler wird folglich mit dem Namen *mips-elf-gcc* aufgerufen. Im Praktikum werden wir hauptsächlich folgende Tools verwenden:

<code>mips-elf-gcc</code>	C-Compiler für die MIPS Architektur
<code>mips-elf-objdump</code>	Liest Informationen aus Objekt-Dateien aus (Segmente, Symboltabelle, Programmcode)
<code>mips-elf-nm</code>	Gibt Symboltabelle von Objekt-Dateien aus
<code>mips-elf-run</code>	MIPS-Emulator für die Ausführung von MIPS Programmen auf dem Host-Prozessor

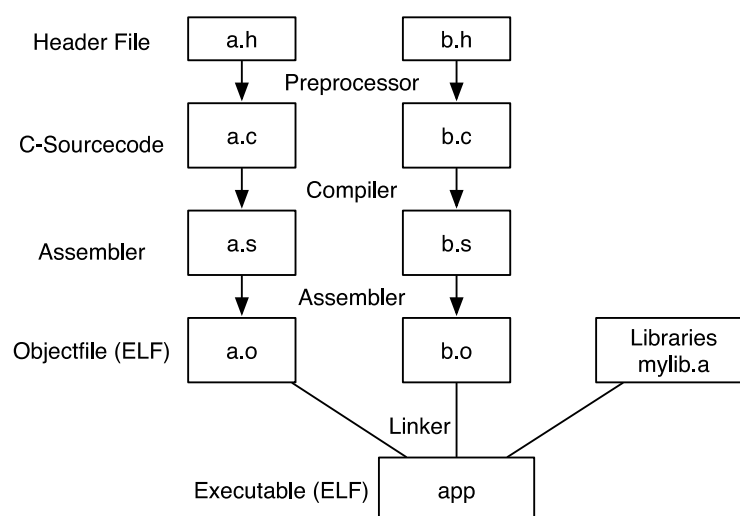


Abbildung 1: Übersicht über den Compilations-Toolflow für ein C Programm

Abbildung 1 präsentiert den Toolflow bei der Compilation eines allgemeinen C Programmes. Die Applikation besteht aus 2 Modulen (C-Quelldateien), die mit dem Compiler in Assembler und dann vom Assembler

in Maschinencode (Objekt-Dateien) übersetzt werden. Der Aufruf des Assemblers erfolgt üblicherweise implizit durch den Compiler. Die Objekt-Dateien beider Module (a.o und b.o) sowie zusätzliche System-Bibliotheken (Libraries) werden vom Linker in ein ausführbares Programm, ein sogenanntes Executable, transformiert.

Objekt-Dateien enthalten nicht nur den Maschinencode des entsprechenden Moduls, sondern auch zusätzliche Daten und Informationen, die für den Linker und für die Ausführung des Programms von Bedeutung sind. Insbesondere besteht eine Objekt-Datei aus folgenden Inhalten:

- Symbole (Abbildung von Funktions- und Variablenamen auf Adressen)
- Ausführbarer Code
- Daten (Speicherbedarf, Adressen und ggf. Initialwerte von globalen Variablen und Konstanten)
- Referenzen auf Funktionen und Daten in anderen Objekt-Dateien
- Relokationsinformationen
- Debuginformationen (z.B. Zusammenhang von Zeilennummern im Quellcode und dem generierten Assembler- bzw. Maschinencode)

Es gibt verschiedene Formate für Objekt-Dateien. In diesem Praktikum verwenden wir das weit verbreitete ELF Format, welches u.a. auch von Linux und Solaris verwendet wird. Beim Linken werden die einzelnen Objekt-Dateien in eine einzige ausführbare Objekt-Datei kombiniert, die ebenfalls im ELF Format gespeichert ist.

## 1 Anlegung von Objekt-Dateien im Speicher

Eine Objekt-Datei im ELF Format ist in verschiedene Teile gegliedert, die als Sektionen bezeichnet werden. Einige wichtige Sektionen sind in Tabelle 1 zusammengefasst.

Sektion	Inhalt
.text	Programmcode in Maschinsprache.
.rodata	Programmdaten, die initialisiert und zur Laufzeit nicht veränderbar sind (Konstanten).
.data	Programmdaten, die initialisiert und zur Laufzeit vom Programm veränderbar sind (Variablen).
.bss	Nicht initialisierte oder mit 0 initialisierte Variablen. Alle Variablen dieser Sektion werden vor der Ausführung automatisch mit 0 initialisiert.

Tabelle 1: Sektionen einer Objekt-Datei im ELF Format (unvollständig)

a) Nehmen Sie an, eine Applikation soll auf einem eingebetteten Mikroprozessorsystem ausgeführt werden, das auf einer MIPS CPU beruht und RAM sowie (Flash-)ROM Speicherbausteine besitzt. Geben Sie für jede der in Tabelle 1 genannten Sektionen an, in welchem Speichertyp der jeweilige Inhalt angelegt werden soll:

Sektion	RAM	ROM
.text	<input type="checkbox"/>	<input type="checkbox"/>
.rodata	<input type="checkbox"/>	<input type="checkbox"/>
.data	<input type="checkbox"/>	<input type="checkbox"/>
.bss	<input type="checkbox"/>	<input type="checkbox"/>

- b) Nehmen Sie an, dass zum Zeitpunkt des Bootens des eingebetteten Systems alle Sektionen im ROM gespeichert sind. Überlegen Sie, wie der Bootvorgang funktionieren muss und notieren Sie den zeitlichen Ablauf. Geben Sie dabei an, wie die einzelnen Sektionen behandelt werden müssen (z.B. wie und wann die Daten im RAM initialisiert werden).

**Lösung:**

a)

Sektion	RAM	ROM
.text		X
.rodata		X
.data	X	
.bss	X	

Die Ausführbaren Instruktionen in der Sektion `.text` sind in der Regel nicht veränderbar und werden daher typischerweise direkt aus dem ROM ausgeführt. Die Daten in der `.rodata` Sektion sind ebenfalls nur lesbar und werden daher bei der Ausführung direkt vom ROM Speicher gelesen. Der Inhalt der `.bss` und `.data` Sektionen ist veränderbar, daher müssen diese Sektionen auf ein RAM abgebildet werden.

*Bemerkung:* Wenn der ROM-Speicher im Vergleich zum RAM-Speicher sehr langsam ist, kann die Ausführung des Programms beschleunigt werden, wenn auch die read-only Sektionen (`.text`, `.rodata`) im RAM abgelegt werden. Dabei muss der Inhalt dieser Sektionen beim Booten des Systems in den RAM-Speicher kopiert werden.

- b) Der Bootprozess könnte wie folgt implementiert werden:
- 1) In einer ersten Phase werden die Prozessorregister initialisiert, die vom Speicherlayout des Systems abhängen (Stackpointer, Framepointer). Ebenso werden die Spezialregister des Prozessors (exception register, interrupt levels) initialisiert.
  - 2) Nun wird das RAM initialisiert. Dazu werden die Initialwerte für die initialisierten Daten (`.data`) aus dem ROM in das RAM kopiert. (Die Tatsache, dass die `.data` Sektion zur Laufzeit auf das RAM abgebildet wird, muss übrigens bereits beim Linken berücksichtigt werden. Dazu wird dem Linker mit einem Linkerskript mitgeteilt, welcher physikalische RAM Speicherbereich für die Data-Sektion reserviert ist).
  - 3) Für die nicht initialisierten Daten der `.bss` Sektion gibt es keinen entsprechenden Kopiervorgang. Die in `.bss` enthaltenen Variablen werden einfach im RAM angelegt und mit 0 initialisiert.
  - 4) Die read-only Daten (`.rodata`) werden bei der Ausführung direkt vom ROM gelesen und brauchen deshalb auch nicht kopiert zu werden. Der Bootprozess ist somit abgeschlossen und es kann mit der Ausführung des eigentlichen Programms (`.text`) begonnen werden.

## 2 Toolflow

In dieser Aufgabe untersuchen wir zunächst, wie der Compiler aus einem C-Programm Assemblercode generiert. Anschliessend betrachten wir die Generierung von Objekt-Dateien und ausführbaren Dateien.

### 2.1 Assemblercode

Gegeben ist das Beispielprogramm `sorter.c`, welches den Bubble-sort Sortieralgorithmus implementiert. Erzeugen Sie den entsprechenden MIPS Assemblercode mit Hilfe des folgenden Befehls:

```
> mips-elf-gcc -S -G0 sorter.c
```

Die Option `-S` bewirkt, dass das Programm compiliert aber nicht assembliert wird. Die Option `-G0` vermeidet für diese Übung unerwünschte Optimierungen.

Der in `sorter.s` generierte Assemblercode enthält neben den eigentlichen Instruktionen viele Assemblerdirektiven, die den Assembler bei der Generierung der Objekt-Datei steuern. Einige wichtige Assemblerdirektiven sind in Tabelle 2 zusammengefasst.

Direktive	Auswirkung
<code>.globl</code>	Exportiert ein Symbol, d.h. den Namen einer Funktion oder Variable. Dadurch werden der Name und die Adresse des Symbols beim Compilieren in der Symboltabelle der Objekt-Datei eingetragen.
<code>.rdata</code>	Die folgenden Daten werden in der Sektion <code>.rodata</code> angelegt.
<code>.data</code>	Die folgenden Daten werden in der Sektion <code>.data</code> angelegt.
<code>.bss</code>	Die folgenden Daten werden in der Sektion <code>.bss</code> angelegt.
<code>.text</code>	Die folgenden Instruktionen werden in der Sektion <code>.text</code> (Programmcode) angelegt.
<code>.word</code>	Die angegebenen Werte werden in aufeinanderfolgende Speicherzellen geschrieben. Die Werte werden als 32 bit Zahlen interpretiert. (Diese Direktive wird zur Initialisierung von Datenstrukturen verwendet.)
<code>.align <i>integer</i></code>	Die folgenden Daten werden in der Objekt-Datei Modulo <i>integer</i> Bytes ausgerichtet.

Tabelle 2: MIPS Assemblerdirektiven (unvollständig)

- Deklariieren Sie in `sorter.c` zwei globale Integer Variablen  $a$ ,  $b$  und eine globale Integer Konstante  $c$ . Initialisieren Sie diese Daten wie folgt:  $a = 7$ ,  $b = 0$ ,  $c = 33$ . Compilieren Sie das Programm erneut wie oben beschrieben und untersuchen Sie die generierten Assemblerdirektiven. In welchen Sektionen der Objekt-Datei legt der Assembler  $a$ ,  $b$  und  $c$  an? Welchen Sinn macht die unterschiedliche Behandlung von  $a$  und  $b$ ? Welchen Sinn macht die unterschiedliche Behandlung von  $a$  und  $c$ ?
- Deklariieren Sie in der Funktion `main` eine lokale Integer Variable  $d$  und initialisieren Sie diese mit einem beliebigen Wert. Compilieren Sie das Programm erneut wie oben beschrieben und untersuchen Sie den generierten Assemblercode. Warum wird  $d$  nicht in einer Daten-Sektion der Objekt-Datei angelegt?
- Trennen Sie die Funktionalität des C-Programmes in zwei Module. Lagern Sie hierfür den Quellcode zum Vertauschen zweier angrenzender Array-Elemente (siehe entsprechenden Kommentar in der Funktion `sort`) in eine eigene Funktion `swap` aus, die in einem separaten Modul `swap.c` erstellt werden soll. Zur Vereinfachung der Aufgabe ist eine entsprechende Header-Datei `swap.h` gegeben und bereits in `sorter.c` eingebunden (Achten Sie auf die Kompatibilität von `swap.h` und `swap.c`). Compilieren Sie die beiden Module wie oben beschrieben und untersuchen Sie den generierten Assemblercode. Wie wird in `swap.s` die Funktion `swap` als solche definiert? Wie wird der Aufruf von `swap` in `sorter.s` behandelt?

### Lösung:

- $a$  wird in Sektion `.data` angelegt (`.data` Direktive).
- $b$  wird in Sektion `.bss` angelegt (`.bss` Direktive).
- $c$  wird in Sektion `.rodata` angelegt (`.rdata` Direktive).

Die unterschiedliche Behandlung von  $b$  im Vergleich zu  $a$  (`.bss` statt `.data`) macht aufgrund der verschiedenen Initialwerte Sinn: Für Variablen, die mit einem Wert ungleich 0 initialisiert sind (Sektion

.data), muss der Initialwert explizit im Assemblercode gespeichert werden. Für Variablen, die mit 0 initialisiert sind (Sektion .bss) wäre dies Speicherverschwendung, da sie mit dem Default-Wert 0 angelegt werden können.

Die unterschiedliche Behandlung von *c* im Vergleich zu *a* macht aufgrund der verschiedenen Schreib Anforderungen Sinn: Variablen (Sektionen .data oder .bss) müssen zwangsläufig in einem beschreibbaren Speicher angelegt werden. Konstanten (Sektion .rodata) können zur Optimierung ggf. auch in einem read-only Speicher angelegt werden.

- b) *d* ist eine lokale Variable und wird deshalb im Unterschied zu *a*, *b*, oder *c* nicht einmal für das gesamte Programm, sondern einmal pro Funktionsaufruf angelegt (bei geschachtelten Funktionsaufrufen also auch mehrmals). Dementsprechend wird *d* im Stack-Speicher angelegt (im activation record des jeweiligen Funktionsaufrufes) und nicht im Daten-Segment des Programms.
- c) In `swap.s` wird die Funktion `swap` definiert, indem ihr Name mit einer `.globl` Direktive exportiert wird. Diese Direktive hat zur Folge dass der Name der Funktion zusammen mit Ihrer Adresse in die Symboltabelle der Objekt-Datei `swap.o` geschrieben wird. In `sorter.s` wird der Aufruf der Funktion `swap` mit einer `jump-and-link` Instruktion auf das entsprechende Label gehandhabt.

## 2.2 Objekt-Dateien und ausführbare Datei

Generieren Sie nun ein ausführbares Programm (Executable) aus den Modulen `swap.c` und `sorter.c`:

```
> mips-elf-gcc -c -G0 swap.c
> mips-elf-gcc -c -G0 sorter.c
> mips-elf-gcc -o app swap.o sorter.o -T idt.ld
```

Die ersten beiden Befehle übersetzen die beiden Quell-Dateien in entsprechende Objekt-Dateien `swap.o` und `sorter.o`. Der letzte Befehl ruft den Linker auf, der die Objekt-Dateien in eine ausführbare Datei `app` kombiniert. Mit der Option `-T` wird dem Linker ein Linker-Skript übergeben, welches die Anordnung der Sektionen im Executable definiert und bei Bedarf auch dafür sorgt, dass Standard-Bibliotheken in das Executable gelinkt werden.<sup>1</sup>

- a) Betrachten Sie die Symboltabelle der Objekt-Datei `sorter.o`:

```
> mips-elf-nm sorter.o
```

Welche Symbole sind definiert, welche undefiniert? Was ist der Unterschied zwischen den definierten und undefinierten Symbolen einer Objekt-Datei? (Die Angaben in der ausgegebenen Symboltabelle können in der Dokumentation von `mips-elf-nm` nachgeschlagen werden: `man mips-elf-nm`)

- b) Extrahieren Sie den Programmcode aus der Objekt-Datei `sorter.o`:

```
> mips-elf-objdump -d sorter.o > sorter.o.text.txt
```

Die Option `-d` steht für 'disassemble'. Sie sorgt dafür, dass der Programmcode nicht in binärer Maschinsprache, sondern in Assembler angezeigt wird. Da `objdump` viel Output generiert, leiten wir diesen in die Datei `sorter.o.text.txt` um. Betrachten Sie den Aufruf der Funktionen `swap` und `print_array` im extrahierten Programmcode (Tipp: Finden Sie die Funktionsaufrufe durch Vergleich mit `sorter.c`). Welche Adressen werden als Sprungziele in den entsprechenden `jal` Instruktionen verwendet? Würde das Programm in dieser Form funktionieren? Was muss der Linker tun, um die Funktionsaufrufe zu 'reparieren'?

- c) Extrahieren Sie die Relokations-Informationen aus der Objekt-Datei `sorter.o`:

```
> mips-elf-objdump -r sorter.o > sorter.o.reloc.txt
```

Betrachten Sie die extrahierte Relokationstabelle. Wozu benötigt sie der Linker beim Erstellen des ausführbaren Programms?

---

<sup>1</sup>Bei Verwendung des üblichen `gcc` Compilers für x86 Prozessoren wäre an dieser Stelle die Angabe eines Linker-Skripts überflüssig, da `gcc` auf vordefinierte Linker-Einstellungen zurückgreifen kann.

- d) Extrahieren Sie den Programmcode aus dem Executable `app`:
- ```
> mips-elf-objdump -d app > app.text.txt
```
- Betrachten Sie erneut den Aufruf der Funktionen `swap` und `print_array` im extrahierten Programmcode. Welche Adressen werden jetzt als Sprungziele in den entsprechenden `jal` Instruktionen verwendet? Rekonstruieren Sie die zwei grundlegenden Aktionen des Linkers.
- e) Weshalb ist die Datei `app` viel grösser als die summierten Dateigrößen von `swap.o` und `sorter.o`? Vergleichen Sie zur Beantwortung der Frage den disassemblierten Programmcode von `sorter.o` und `app`.

### Lösung:

- a) Definierte Symbole: `a`, `b`, `c`, `main`, `print_array`, `sort`, `unsorted_array`  
 undefinierte Symbole: `printf`, `swap`  
 Unterschied: Definierte Symbole beziehen sich auf Daten oder Funktionen, die im eigenen Modul definiert sind und für die Verwendung in anderen Modulen zu Verfügung stehen. undefinierte Symbole beziehen sich hingegen auf Daten oder Funktionen, die in anderen Modulen definiert sind.
- b) In der Objektdatei sind die Sprungziele aller `jal` Instruktionen vorläufig auf die Adresse 0 gesetzt. Der Grund dafür ist, dass vor dem Linken die effektive Position (Adresse) der Funktionen nicht bekannt ist. Das Programm würde in dieser Form folglich nicht korrekt funktionieren. Um die Funktionsaufrufe im Executable richtig zu stellen, muss der Linker nach dem Zusammenfügen aller Objektdateien die entsprechenden Funktionsadressen im Programmcode eintragen.
- c) Die Relokationstabelle der Objekt-Datei gibt an, an welchen Programminstruktionen der Linker Adressen ersetzen muss, damit Zugriffe auf Variablen und Funktionen korrekt funktionieren. Beispielsweise bedeutet die Zeile
- ```
00000044 R_MIPS_26 swap
```
- in der Relokationstabelle von `sorter.o`, dass der Linker an der (relativen) Adresse 00000044 im Programmcode, die der Instruktion `jal swap` entspricht, das Sprungziel ersetzen muss.
- d) Im Executable `app` sind die Sprungziele der `jal` Instruktionen auf die entsprechenden effektiven Funktionsadressen gesetzt. `swap`: `a00202b8`, `print_array`: `a002047c` (Ergebnisse programmabhängig). Die grundlegenden Aktionen des Linkers sind
- Programcode der verschiedenen Objekt-Dateien in eine einzelne ausführbare Datei zusammenfügen.
  - Alle Adressen von Funktionsaufrufen und Variablen ersetzen (Relokationstabelle abarbeiten).
- e) Der Programmcode im Executable `app` ist um ein Vielfaches länger als die Programmcode in `sorter.o` und `swap.o`. Grund dafür ist der Aufruf der Bibliotheks-Funktion `printf` in `sorter.c`. `printf` ruft sehr viele Unterfunktionen auf, die alle in das Executable `app` gelinkt werden müssen.

## 3 Analyse eines unbekanntes Programmes mit `objdump`

Gegeben sei ein Programm, das einen String mit einer unbekanntes Funktion verschlüsselt. Leider ist der Sourcecode des Programms verloren gegangen. Bekannt ist lediglich, dass jedes Zeichen des Strings einzeln verschlüsselt wird.

Versuchen Sie den Verschlüsselungsalgorithmus aus dem Executable `encrypt` zu rekonstruieren. Kennen Sie den Verschlüsselungsalgorithmus?

Hinweise:

- Benutzen Sie das Programm `mips-elf-objdump` mit geeigneten Parametern.

- Suchen Sie die Verschlüsselungsfunktion.
- Beachten Sie die genaue Semantik der MIPS-Instruktionen `addiu` und `sltiu`.
  - `addiu`: Beide Operanden werden wie bei `addi` mit dem jeweiligen Vorzeichen interpretiert (0xxxx positive Zahl, 1xxxx negative Zahl dargestellt im Zweierkomplement), jedoch wird bei `addiu`, im Gegensatz zu `addi`, im Falle eines Overflows keine Exception gemeldet.
  - `sltiu`: Beide Operanden werden, im Gegensatz zu `slti`, vorzeichenlos interpretiert. Das bedeutet, dass für `sltiu` Zahlen mit 1xxxx grösser sind als Zahlen mit 0xxxx).
- Zur Lösung dieser Aufgabe ist eine ASCII Tabelle hilfreich (z.B. <http://www.lookuptables.com>).

### Lösung:

Beim verwendeten Verschlüsselungsverfahren handelt es sich um das klassische ROT13 Verfahren. Dabei wird jeder Buchstabe um 13 Positionen im Alphabet verschoben.

Der Algorithmus ist, wie jedes Ersetzungsverfahren, kryptografisch nutzlos. Er wird jedoch in Newsreadern und Emailprogrammen häufig verwendet, um einen Text für Menschen unlesbar zu machen.

Der disassemblierte Code (mit Kommentaren) lautet:

```
a0 = c (argument)
```

ASCII Codes

```
'a' = 97  'm' = 109  'z' = 122
'A' = 65  'M' = 77   'Z' = 90
```

```
encrypt_char:
```

```
// check whether 'a' <= c <= 'm' or 'A' <= c <= 'M'
// if yes, return c+13
```

```
a0020470 <encrypt_char>:
```

```
a0020470: addiu   v0,a0,-97           // ASCII 'a' = 97
a0020474: sltiu   v0,v0,13           // check if 'a' <= c <= 'm'
a0020478: bnez    v0,a0020490 <encrypt_char+0x20> // if yes goto add13
```

```
a002047c: nop
```

```
a0020480: addiu   v0,a0,-65           // ASCII 'A' = 65
a0020484: sltiu   v0,v0,13           // check if 'A' <= c <= 'M'
a0020488: beqz    v0,a002049c <encrypt_char+0x2c> // if no goto check_upper
```

```
a002048c: nop
```

```
add_13:
```

```
a0020490: addiu   a0,a0,13           // c = c + 13
a0020494: j       a00204c0 <encrypt_char+0x50> // goto end
```

```
// check whether 'n' <= c <= 'z' or 'N' <= c <= 'Z'
// if yes, return c-13
```

```
a0020498: nop
```

```
check_upper:
```

```
a002049c: addiu   v0,a0,-110          // ASCII 'n' = 110
a00204a0: sltiu   v0,v0,13           // check if 'n' <= c <= 'z'
a00204a4: bnez    v0,a00204bc <encrypt_char+0x4c> // if yes goto subtract_13
```

```
a00204a8: nop
```

```
a00204ac: addiu   v0,a0,-78           // ASCII 'N' = 78
a00204b0: sltiu   v0,v0,13           // check if 'N' <= c <= 'Z'
```

```

a00204b4: beqz    v0,a00204c0 <encrypt_char+0x50>// if no goto end

a00204b8: nop
subtract_13:
a00204bc: addiu   a0,a0,-13                // c = c - 13

end:
a00204c0: move   v0,a0                    // store result
a00204c4: jr     ra                       // return result

a00204c8: nop

```

Das ursprüngliche Programm lautete:

```

int encrypt_char( int c )
{
  if (((c >= 'a') && (c <= 'm')) || ((c >= 'A') && (c <= 'M'))) {
    return c+13;
  } else if (((c > 'm') && (c <= 'z')) || ((c > 'M') && (c <= 'Z'))) {
    return c-13;
  } else {
    return c;
  }
}

```