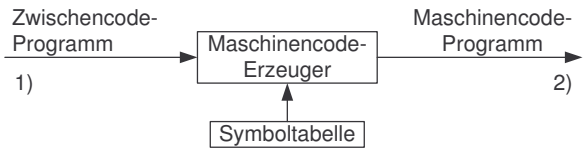
7 Maschinencode-Erzeugung

Dieser Abschnitt beschäftigt sich mit der Maschinencode-Erzeugung aus einem vorher generierten und optimierten Zwischencode. Dieser Abschnitt ist natürlich in hohem Maße von der Architektur der Maschinenbefehle der Zielmaschine abhängig. Aus diesem Grund werden einige Verfahren auch nur ansatzweise erläutert, damit die prinzipielle Vorgehensweise nicht durch die Komplikationen der realen Maschinenstruktur überdeckt wird.

7.1 Einführung

Der prinzipielle Aufbau eines Maschinencode-Erzeugers wird durch folgende Zeichnung ersichtlich:



Zu 1) Übersetzung ist soweit durchgeführt, dass

- Werte in der Zwischensprache direkt auf der Zielmaschine dargestellt werden können (integer, real, bits, etc.)
- notwendige Typanpassungen eingefügt wurden.

Zu 2) Verschiedene Formen sind möglich:

- absolutes Maschinenprogramm (direkt ausführbar)
- Objektprogramm (als Eingabe für den Binder)
- Assemblerprogramm

Welche Probleme treten bei der Maschinencode Erzeugung auf?

1) Auswahl der Maschinenbefehle speziell bei Zielmaschinen mit einem Befehlscode, der viele Spezialfälle enthält.

   Beispiel 7.1:
   Bei den Motorola 68000-Prozessoren gibt es Varianten einfacher Befehle, bei denen der Operand im Maschinenbefehl kodiert werden kann, wenn er klein genug ist. Bei größeren Operanden muss der Befehl um ein sogenanntes Erweiterungswort verlängert werden:

Drei-Adress-Code		Assemblercode	Länge des Befehls
`A := 4`	$\longrightarrow$	`MOVEQ #4,A`	1 Wort
`A := 100`	$\longrightarrow$	`MOVE #100,A`	2 oder 3 Worte
`A := A + 3`	$\longrightarrow$	`ADDQ #3,A`	1 Wort
`A := A + 10`	$\longrightarrow$	`ADDI #10,A`	2 Worte

   Die „richtige“ Antwort kann eigentlich nur getroffen werden, wenn möglichst viele Informationen über Zeit- und Platzbedarf einzelner Befehle vorhanden sind. Sicherlich ist die Übersetzung von Drei-Adress-Befehlen nicht kontextfrei – eine Übersetzung, die isoliert nur einen Befehl nach dem nächsten behandelt, ist nicht günstig.

Die Entscheidung, welche Werte über einen längeren Zeitraum in Registern gehalten werden sollen, kann nur nach genauer Datenfluß-Analyse getroffen werden.

Die Erzeugung von schnellen und platzsparenden Maschinencode macht bei RISC-Maschinen zwar weniger Probleme bei der Auswahl der Befehle, dafür ist die Reihenfolge der Maschinenbefehle wichtig (Pipelining).

2) Registerzuordnung
Befehle, deren Operanden in Hardwareregistern vorliegen, sind meist kürzer und schneller abzuarbeiten als äquivalente Befehle, deren Operanden im Speicher abgelegt sind. Daher ist es wichtig, häufig benutzte Variablen bzw. Werte in Registern zwischenzuspeichern. Erschwerend kommt hinzu, dass manche Befehle nur für spezielle Register oder auch Registerpaare erlaubt sind, z.B. bei der PDP 11 (einer 16-Bit-Maschine):

`MUL A,R0`	$\longrightarrow$	Das Ergebnis der Multiplikation der beiden Zahlen in Speicherzelle `A` und im Register `R0` wird in `R0` *und* `R1` abgelegt.
`DIV A,R0`	$\longrightarrow$	`R0` *und* `R1` werden als 32 Bit Zahl interpretiert und durch die Zahl in Speicherzelle `A` dividiert, `R0` enthält danach den Quotienten, `R1` den Rest.

Leider sind die meisten in diesem Zusammenhang auftretenden Probleme NP-vollständig, d.h. es ist nicht zu erwarten, dass effiziente Algorithmen existieren, die optimale Registerzuordnungen oder optimale Maschinencodes liefern.

7.2 Die Zielmaschine

Im folgenden soll die grundlegende Vorgehensweise bei der Maschinencode-Erzeugung anhand einer einfachen, hypothetischen Maschine aufgezeigt werden. Unser hypothetischer Beispielrechner besitzt (im Gegensatz zur Realität) nur wenige Befehlstypen mit relativ klarer Semantik. Es sollen die folgenden Annahmen gelten:

- Die Hardware ist byte-orientiert,
- Die Maschine besitzt Drei-Adress-Befehle
- Es gibt n allgemeine Register `R0`, `R1`, ..., `Rn`
- Es gibt Lade-Operationen der Form `LD Ri,`*src* mit der Bedeutung, dass der Wert in Adresse *src* in das Zielregister `Ri` geladen wird. `LD R0,A` speichert also den Inhalt der Speicherzelle `A` im das Register `R0`.
- Es gibt Speicher-Operationen der Form `ST` *dst*`,Ri` mit der Bedeutung, dass der Wert im Register `Ri` in die Speicherzelle *dst* gespeichert wird.
- Es gibt übliche Operationen der Form `OP Ri`, *src1*, *src2*, wobei `OP` eine „übliche“ Operation wie `MOV, ADD, SUB, ...` ist und *src1* und *src2* (nicht notwendigerweise unterschiedliche) Speicherzellen sind. Bei einstelligen Operationen wie etwa `NEG, INC,...` entfällt *src2*. Der Befehl `SUB R0, R1, A` subtrahiert zum Beispiel vom Wert im Register `R1` den Wert in der Speicherzelle `A` und speichert das Ergebnis im Register `R0`. `INC R1` erhöht den Inhalt des Registers `R1`.
- Es gibt unbedingte Sprünge der Form `BR` L, bei denen der Rechner zum Befehl mit Label L verzweigt.

- Es gibt bedingte Sprünge der Form `B`*cond* `Ri`, *L*, dabei steht *cond* für eine der „üblichen“ Bedingungen, die an den Wert im Register `Ri` gestellt werden. `BLTZ R1,L1` verzweigt nur dann zum Label `L1`, wenn der Inhalt im Register `R1` kleiner oder gleich 0 ist.
- Der Rechner hat eine Vielzahl von Adressierungsarten,
  - Ein Variablenname bezeichnet den Inhalt der Speicheradresse für diese Variable. Ist die Variable *a* zum Beispiel ab Adresse 1000 im Speicher abgelegt, so bezeichnet diese Adressierung den Inhalt der Speicherzelle 1000, geschrieben *contents*(1000).
  - Ein Konstrukt der Form *num*`(Ri)`, wobei *num* eine Integer-Zahl ist, bezeichnet den Inhalt einer Speicherzelle, deren Adresse man durch Addition der Zahl auf den Inhalt des Registers `Ri` erhält, also *contents*(*num* + *contents*(`Ri`)). Manchmal schreibt man dies auch in der Form *a*`(Ri)`, wobei der Variablenname *a* dann für die Adresse steht, ab der *a* gespeichert ist.
  - Ein Konstrukt der Form `*Ri` steht für eine indirekte Adressierung und bezeichnet den Inhalt einer Speicherzelle, deren Adresse in einer Speicherzelle steht, deren Adresse wiederum in Register `Ri` steht. Also bezeichnet `*Ri` den Wert *contents*(*contents*(*contents*(`Ri`)))
  - Ein Konstrukt der Form `*`*num*`(Ri)` steht für eine indirekte Adressierung und bezeichnet den Inhalt einer Speicherzelle, deren Adresse in einer Speicherzelle steht, deren Adresse sich durch Addition der Zahl *num* auf den Wert im Register `Ri` ergibt. Wir haben also *contents*(*contents*(*num* + *contents*(`Ri`)))
  - Ein Konstrukt der Form `#`*num* beschreibt eine Konstante. Der Befehl `LD R1,#100` lädt zum Beispiel die Konstante 100 in das Register `R1`.

Als Näherung für den Zeitbedarf zur Abarbeitung eines Befehls werden die Kosten eines Befehls definiert.
Die Kosten eines Befehls seien definiert als die Länge des Befehls (1 plus der Anzahl der zusätzlichen Speicherzugriffe zum Zugriff auf Konstanten, die nach dem Befehl gespeichert werden) plus der Zahl der Speicherzugriffe auf Operanden der Operation. Die Kosten entsprechen also der Gesamtzahl der Speicherzugriffe.

Beispiel 7.2:

Befehl	Länge	Kosten
`LD R0,R1`	1	1
`LD R0,M`	2	3
`LD R0,#100`	2	2
`LD R0,100(R1)`	2	3
`LD R0,*R1`	1	2
`LD R0,*100(R1)`	2	4
`ADD R1,*50(R0),*(R2)`	2	6
`BLTZ R1,M`	2	2

Beispiel 7.3:
Beispielsweise sollen hier mögliche Übersetzungen von `a := b + c` betrachtet werden.
Annahme: Register `R1` und `R2` sind frei:

1)
```
LD  R1,b
LD  R2,c
ADD R1,R1,R2
ST  a,R1
```
Länge 7, Kosten 10

2)
```
ADD R1,b,c
ST  a,R1
```
Länge 5, Kosten 8

Wenn `R3` und `R4` die Adressen von `b` und `c` enthalten, dann:

3)
```
ADD R1,*R3,*R4
ST  a,R1
```
Länge 3, Kosten 6

Wenn `R1` und `R2` die Werte von `b` und `c` enthalten und der Wert von `b` ab dieser Stelle nicht weiter benötigt wird:

4)
```
ADD R1,R1,R2
ST  a,R1
```
Länge 3, Kosten 4

Man sieht hieran, das eine der wesentlichen Aufgaben der Maschinencode-Erzeugung die Registerzuordnung ist! Außerdem erkennt man, wie in der vorangehenden Phase gewonnene Informationen zur Maschinencode-Erzeugung benutzt werden können und dass man Informationen über den momentan Speicherort der Werte von Variablen, in diesem Fall die Tatsache, dass sich im Register `R1` der Wert von a befindet, für die Übersetzung des nächsten Drei-Adress-Befehls benutzen kann.
Betrachten wir mögliche Übersetzungen weiterer Drei-Adress-Befehle:
Annahme: a ist ein Feld von Objekten, die jeweils 8 Bytes belegen. Die untere Indexgrenze sei 0. Dann kann man den Befehl `b = a[i]` übersetzen in:

5)
```
LD  R1,i
MUL R1,R1,#8
LD  R2,a(R1)
ST  b,R2
```
Länge 8, Kosten 11

Einen Befehl der Form `a[i] = b` kann man übersetzen in:

6)
```
LD  R1,b
LD  R2,i
MUL R2,R2,#8
ST  a(R2),R1
```
Länge 8, Kosten 11

Einen Befehl der Form `if x<y goto L` übersetzt man in:

7)
```
LD   R1,x
LD   R2,y
SUB  R1,R1,R2
BLTZ R1,M
```
Länge 7, Kosten 9

wobei `M` die dem Label `L` zugeordnete Speicheradresse ist. Sind die Variablen `x` und `y` nach diesem Befehl tot, wäre es günstiger, die ersten drei Befehle durch `SUB R1,x,y` zu ersetzen!