Universität Karlsruhe (TH)

Institut für Innovatives Rechnen und Programmstrukturen (IPD)

Übersetzerbau WS 2003/04 — http://www.info.uni-karlsruhe.de/

Dozent: Prof. Dr.rer.nat. G. Goos — goos@ipd.info.uni-karlsruhe.de

Übungsleiter: Rubino Geiß — rubino@ipd.info.uni-karlsruhe.de

Übungsblatt 7 | Ausgabe: 27.01.2004 | Besprechung: 29.01.2004

Aufgabe 1: *Auswertung Boolescher Ausdrücke*

Boolesche Ausdrücke können oft mit bedingten Sprüngen ausgewertet werden. Dabei muss jeweils die Adresse spezifiziert werden, an der die Berechnung nach dem Sprung fortgesetzt wird. Gegeben sei folgende Syntax Boolescher Ausdrücke:

conditional_clause	::=	`if` *boolean_expr* `then` *stmt_list* `else` *stmt_list* `end`
boolean_expr	::=	*boolean_expr* *boolean_op* *boolean_expr*
boolean_expr	::=	`not` *boolean_expr*
boolean_op	::=	`and` | `or`

Sprungmarken sollen mit Referenzen auf Knoten des Syntaxbaumes spezifiziert werden, für die der Assembler später Adressen einsetzen muss. Die Funktion *new_label* liefert jeweils eine neue Sprungmarke.

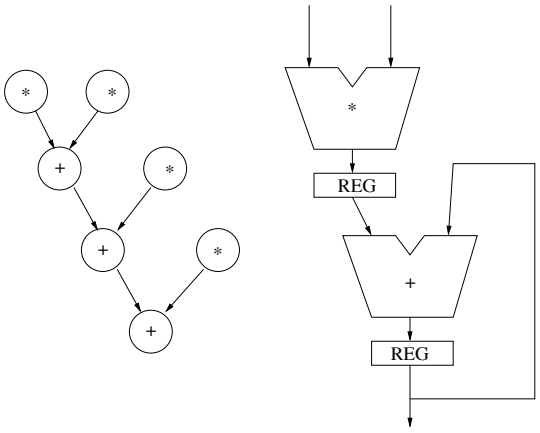
Geben Sie eine attributierte Grammatik an, die die Auswertung Boolescher Ausdrücke spezifiziert.

Aufgabe 2: *Codegenerierung für Signalprozessoren*

Betrachten wir eine Standardfilteroperation:

$$y_n := c_0 \cdot x_n + c_1 \cdot x_{n-1} + c_2 \cdot x_{n-2} + c_3 \cdot x_{n-3}$$

Der Einfachheit wegen ignorieren wir Lade- und Speicheroperationen. Berechnungsbaum und Recheneinheit sind wie unten beschrieben.



2.1 *Maschinensimulation*

Angenommen Multiplikation und Addition können gleichzeitig ausgeführt werden und benötigen einen Takt. Mit wievielen Takten kann die Berechnung durchgeführt werden?

2.2 *Verallgemeinerung*

Versuchen Sie, Ihr Vorgehen in einem Algorithmus formulieren!

Aufgabe 3: *Codeselektion*

Der *VAX 11* Prozessor kennt u.a. die folgenden Befehle (vereinfacht):

Befehl	Bedeutung
`movea src, dst`	Berechne Adresse von `src` und speichere in `dst`.
`move src, dst`	Kopiere `src` nach `dst`.
`add2 op1, op2`	`op2 := op1 + op2`
`add3 op1, op2, op3`	`op3 := op1 + op2`
`inc op`	`op := op + 1`
`tst op`	Vergleicht `op` mit `0` und setzt die *Conditioncode (CC) bits*.
`cmp op1, op2`	Vergleicht `op1` mit `op2` und setzt die CC bits.
`beql label`	Wenn das CC bit *equal* gesetzt ist, springe zu `label`, sonst führe nächste Anweisung aus.
`blss label`	Wenn das CC bit *less* gesetzt ist, springe zu `label`, sonst führe nächste Anweisung aus.
`jump label`	Springe zu `label`
`label:`	Setzt die Codemarke `label`.

Die Operanden können sein:	
`r0 ... r13`	Register, `r14` und `r15` werden für andere Zwecke benötigt.
$\texttt{r}_i$	Register direkt, d.h. der Inhalt des Registers $\texttt{r}_i$ wird benutzt.
$(\texttt{r}_i)$	Register indirekt, d.h. der Inhalt des Speichers unter der Adresse Inhalt von $\texttt{r}_i$.
`identifier`	Speicher direkt, d.h. `identifier` steht für die Adresse, an der die Variable `identifier` gespeichert ist.
`#int_const`	Integerkonstante mit dem Wert *int_const*.
`label`	Codemarke.

Gegeben sei die folgende 3-Adreßzwischensprache:

1	$\texttt{t}_i$ `:` `adr` *identifier*	$\texttt{t}_i$:= Speicheradresse der Variablen *identifier*
2	$\texttt{t}_i$ `:` `load` $\texttt{t}_j$	$\texttt{t}_i$:= Memory $[\texttt{t}_j]$
3	$\texttt{t}_i$ `:` `const` *int_const*	$\texttt{t}_i$:= *int_const*
4	$\texttt{t}_i$ `:` `plus` $\texttt{t}_j$ $\texttt{t}_k$	$\texttt{t}_i := \texttt{t}_j + \texttt{t}_k$
5	$\texttt{t}_i$ `:` `less` $\texttt{t}_j$ $\texttt{t}_k$	$\texttt{t}_i := \texttt{t}_j < \texttt{t}_k$
6	$\texttt{t}_i$ `:` `if` $\texttt{t}_j$ $\texttt{t}_k$	Wenn $\texttt{t}_j$ `= 0` springe Anweisung $\texttt{t}_k$ sonst führe $\texttt{t}_{i+1}$ aus
7	$\texttt{t}_i$ `:` `goto` $\texttt{t}_j$	Springe zur Anweisung $\texttt{t}_j$
8	$\texttt{t}_i$ `:` `store` $\texttt{t}_j$ $\texttt{t}_k$	Memory $[\texttt{t}_j] := \texttt{t}_k$

3.1 *„on the fly"-Registerzuteilung*

Die Register werden „on the fly" zugeteilt, d.h. wird ein Wert berechnet, muß ein „freies" Register beschafft werden, das diesen Wert aufnimmt. Nachdem der Wert eines Registers in einer anderen Instruktion benutzt wurde, wird das Register wieder freigegeben. Falls nicht mehr ausreichend viele Register zur Verfügung stehen, wird die Fehlermeldung *Expression too complex* ausgegeben und das Programm abgebrochen.

Die Verwaltung der freien/belegten Register wird durch die folgenden Prozeduren erledigt:

`PROCEDURE new_reg() : INTEGER`

liefert die Nummer eines freien Registers,

`PROCEDURE release_reg(reg : INTEGER)`

gibt das Register `reg` wieder frei.

Implementieren Sie die Prozeduren.

3.2 *Codeselektion mit AG's*

Entwerfen Sie eine Attributgrammatik, die für diese Zwischensprache VAX-Code erzeugt, und Register zuteilt. Die $\texttt{t}_i$ haben die Attribute `reg`, `code` und `label`[1]. Der Aufruf der Registerzuteilungsprozeduren kann man sich

[1] `PROCEDURE new_label():tLabel` liefert bei jedem Aufruf eine neue Labelkennung.

als Berechnungsvorschriften für ein zusätzliches Attribut vorstellen, das in jedem Knoten berechnet werden muß (vgl. Testen von Bedingungen).

3.3 *Codeerzeugung am Beispiel*

Erzeugen Sie Zwischen- und VAX-Code für die folgenden Quellsprachanweisungen:
`i := 1`,
`i := i + 1`,
`i := j + k` und
`IF i < j THEN min := i ELSE min := j END`.

3.4 *Optimierung*

Geben Sie besseren VAX-Code an.

3.5 *Baumüberdeckungen*

Wie könnte dieser Code erzeugt werden? Beschreiben Sie Ihr Verfahren.

(Hinweis: Betrachten Sie eine andere Darstellungsform für die Liste der 3-Adreßanweisungen, z.B.: Jede Anweisung ist ein Knoten in einem Baum, die t_i sind Kanten zwischen diesen Knoten; oder: Jede Anweisung ist ein Termkonstruktor, die t_i repräsentieren entsprechende Subterme. Anstatt nun für jede Anweisung einzeln Code zu erzeugen (Makroexpansion) könnte nun Code für einen komplexeren Teilbaum / Subterm erzeugt werden, vgl. Patternmatching auf Bäumen / Termen.)

3.6 *„delayed branch“*

Ältere RISC-Prozessoren, deren Instruktionen in einer Pipeline ausgeführt werden, kennen sogenannte *delayed branch* Instruktionen. Bei ihnen wird die ihnen folgende Instruktion ausgeführt, bevor der eigentliche Sprung erfolgt. Z.B. wird bei der Instruktionsfolge: `jump lab; move r1, (r2);` vor dem Sprung nach `lab` noch die `move` Instruktion ausgeführt. Man sagt, daß die `move` Instruktion im *delay slot* des verzögerten Sprunges steht.

Angenommen, die VAX hätte *delayed branch* Instruktionen und einen Befehl `nop`, der nichts macht.

1. Was ist bei der Codeerzeugung zu beachten?
2. Wann kann eine Instruktion, die ursprünglich vor dem „normalen“ Sprung ausgeführt werden soll, in das *delay slot* eines verzögerten Sprunges verschoben werden?
3. Kann auch eine Instruktion nach vorne verschoben werden, die ursprünglich nach dem „normalen“ Sprung ausgeführt werden sollte?
4. Es gibt allerdings nicht nur verzögerte Sprünge sondern, z.B. auch verzögerte Speicherladebefehle, deren Ergebnis erst in der übernächsten Instruktion verwendbar ist. (Der Grund liegt in dem im Vergleich zum Prozessor langsameren Speicher.) Was ist hierbei zu beachten?
5. Neuere Architekturen verwenden *spekulative Auswertung*, bei denen ein Zweig als beforzugt angenommen wird. Was muß hierbei beachtet werden?