Prof. Dr. M. Jäger FB MNI

Praktikumshinweise Compilerbau SS 2018

Assembler-Code-Generator für SPL (Version 2 vom 1.7.2018)

Zielplattform

Zielplattform ist die Puck Vitual Machine (https://homepages.thm.de/~hg52/lv/compiler/praktikum/puck/Puck_VM_Specification.pdf). Die 32-Bit RISC-Maschine hat 32 Mehrzweckregister ($0, ..., $31) der Größe 32 Bit. Folgende Register werden von der Codegenerierung für spezielle Zwecke genutzt:

- $0 hat immer den Wert 0 und kann nicht verändert werden,
- $29 speichert den Framepointer (FP),
- $30 enthält die Rücksprungadresse (RETURN)
- $31 speichert den Stackpointer (SP).

Bei Prozeduraufrufen ist der Caller für die korrekte Belegung von RETURN verantwortlich. Der Callee muss SP und FP beim Aufruf aktualisieren (Prolog) und vor dem Rücksprung auf die alten Werte zurücksetzen (Epilog).

Die Mehrzweckregister $8-$23 werden für die Speicherung von Zwischenwerten genutzt.

Stackmaschine

Eine Stackmaschine ist ein Prozessor, dessen Mehrzweckregister als Stack organisiert sind. Um Verwechslungen zu vermeiden, bezeichnen wir diesen Stack im folgenden auch als Registerstack (RSTACK) und den für die Prozeduraufrufe im Hauptspeicher genutzten Stack der Aktivierungsrahmen als Laufzeitstack (LZSTACK). Für den Registerstack ist ein Spezialregister vorhanden, das auf das letzte belegte Register von RSTACK verweist: Der Registerstack-Pointer (RSP). Die Ausführung einer Maschineninstruktion lässt sich in verschiedene Teilschritte untergliedern. Dabei werden viele Maschineninstruktionen RSP nutzen und auch verändern.

Beispiel:

Der Ausdruck `3*5+2*6` erfordert zunächst zwei Multiplikationen und am Ende eine Addition. Jede dieser Rechenoperationen entspricht einer Maschineninstruktion (ADD, MUL). Diese Instruktionen erwarten ihre Operanden oben auf dem RSTACK, genauer: Der linke Operand steht in `RSTACK[RSP-1]` der rechte in `RSTACK[RSP]`. Bei der Ausführung der Instruktion werden implizit die Operanden vom Registerstack entfernt und durch das Ergebnis ersetzt:

```
PUSH 3        ; RSTACK[++RSP]<--3
PUSH 5        ; RSTACK[++RSP]<--5
MUL           ; RSTACK[RSP-1]<--RSTACK[RSP-1]*RSTACK[RSP], RSP--
PUSH 2        ; RSTACK[++RSP]<--2
PUSH 6        ; RSTACK[++RSP]<--6
MUL           ; RSTACK[RSP-1]<--RSTACK[RSP-1]*RSTACK[RSP], RSP--
ADD           ; RSTACK[RSP-1]<--RSTACK[RSP-1]+RSTACK[RSP], RSP--
```

Der Codegenerator für eine binäre Operation verwendet also immer das gleiche Muster:

1. Code zur Berechnung des linken Operanden generieren (Rekursion)
2. Code zur Berechnung des rechten Operanden generieren (Rekursion)
3. Maschinenbefehl für die Rechenoperation anhängen

Simulation der Stackmaschine durch RISC-Prozessor

Ein einfaches Prinzip für die Nutzung der Mehrzweckregister unserer Puck VM ist die Simulation der Stackmaschine. Der Codegenerator nutzt die 16 Register $8-$23 in Form eines Register-Stacks. Beispielsweise könnte man diese Register gemäß aufsteigender Nummerierung benutzen. Um das RSP-Register der Stackmaschine zu ersetzen, merkt sich der SPL-Compiler immer das letzte belegte Register in einer internen Variable und generiert den Code so, dass die aktuelle Instruktion immer die Register nutzen, die den „obersten“ des simulierten Registerstacks entsprechen. Das heißt, wenn ein Speicherplatz benötigt wird, wird immer das freie Register mit der kleinsten Nummer verwendet. Sobald ein Wert nicht mehr gebraucht wird, steht dieses Register wieder zur Verfügung.

Die Simulation der Berechnung von `3*5+2*6` im Vergleich mit der Stackmaschine unter der Annahme, dass zu Beginn das Register $8 schon belegt und $9 das erste freie Register ist:

```
Stack-Maschine              Puck VM

 PUSH 3                     SETW $9 3
 PUSH 5                     SETW $10 5
 MUL                        MULI $9 $9 $10
 PUSH 2                     SETW $10 2
 PUSH 6                     SETW $11 6
 MUL                        MULI $10 $10 $11
 ADD                        ADD  $9 $9 $10
```

Lokale Variablen und Parameter

Lokale Variablen und Parameter stehen im Laufzeitstack und werden in der Form *FP + Offset* adressiert. Ein lesender Zugriff (VarExp-Knoten im AST) wird durch eine LDW-Instruktion (Load Word) umgesetzt, ein schreibender Zugriff (Wertzuweisung) durch ein STW-Instruktion (Store Word). In beiden Fällen muss vorher die Adresse berechnet werden.

Die Puck VM hat im Gegensatz zu anderen RISC-Maschinen keine Möglichkeit, bei LDW und STW die Adressen in der Form *Registerinhalt+Offset* anzugeben. Daher muss die Adresse einer Variablen durch eine Additionsinstruktion berechnet werden. Hier bietet sich die *ADDC*-Instruktion an, die den Inhalt eines Registers zu einem 4 Byte großen Direktwert addiert. Als Register wird der Framepointer verwendet, der Direktwert ist der *Offset*, den der Compiler im Symboltabelleneintrag der Variablen findet.

Beispiel:

```
proc p(){
  var i: int;
  var j: int;
  i := 5;
  j := i+1;
}
```

Da *Offset(i)* = -4, *Offset(j)* = -8, FP=$29, ist der Assemblercode zur Wertzuweisung `j := i+1;` wie folgt untergliedert:

1. Zieladresse (linke Seite der Wertzuweisung) berechnen und in Register 8 speichern

   ```
   ADDC  $8 $29 -8   ; $8 <-- Adresse(j)
   ```

2. Wert der rechten Seite berechnen und in Register 9 speichern.

```
                    ; linker Operand der Addition
ADDC $9 $29 -4      ; $9 <-- Adresse(i)
LDW  $9 $9          ; $9 <-- Wert(i)
                    ; rechter Operand der Addition
SETW $10 1
                    ; Maschineninstruktion entsprechend SPL-Operator
ADD $9 $9 $10
```

3. Mit STW den Inhalt von Register 9 in den Hauptspeicher kopieren.

```
STW $9 $8
```

Referenzparameter

Da der Wert eines Referenzparameters des Callee die Adresse einer Variable des Callers ist, muss die im SPL-Quelltext nicht sichtbare zusätzliche Dereferenzierung auf der Maschinenebene durch eine zusätzliche LDW-Instruktion umgesetzt werden.

Beispiel:

```
proc p(ref i:int){
  i := i+1;
}
```

Da *Offset(i)* = 0, sieht der Code für die linke Seite der Wertzuweisung wie folgt aus:

```
ADDC $8 $29 0       ; $8 <-- Adresse(i)
LDW  $8 $8          ; $8 <-- Adresse der Variablen, auf die i verweist
```

Wert der rechten Seite berechnen und in Register 9 speichern.

```
ADDC $9 $29 0       ; $9 <-- Adresse(i)
LDW  $9 $9          ; $9 <-- Adresse der Variablen, auf die i verweist
LDW  $9 $9          ; $9 <-- Wert der Variablen, auf die i verweist
SETW $10 1
ADD $9 $9 $10
```

Array-Komponenten

Bei Array-Komponenten wird die Adresse wie folgt berechnet:

Feldanfangsadresse $+$ *Indexwert* $*$ *Bytegröße der Feldkomponenten*

Die Feldanfangsadresse wird wie bei einfachen Variablen aus *FP+Offset* bestimmt. Die Bytegröße der Feldkomponenten findet der Compiler im Basistyp des Array-Typs des Feldes. Der Compiler soll durch zusätzliche Instruktionen sicher stellen, dass zur Laufzeit eine Indexprüfung erfolgt.

Bedingte Anweisungen und Schleifen

Zu jeder SPL-Vergleichsoperation gibt es in der Puck VM eine passende Machineninstruktion, die das Vergleichsergebnis in einem Register speichert. Mit bedingten Sprunganweisungen kann man abhängig vom Testausgang zu einer anderen Stelle im Code springen. Die Sprungadressen werden im Assembler durch Labels repräsentiert.

Ein Beispiel: IF-Anweisung ohne ELSE

```
if ( 3 < 5 )
  STMT
```

Zunächst wird der Maschinencode für EXP erzeugt. Nach dessen Ausführung steht das Testergebnis im ersten freien Register ($8):

```
SETW $8 3
SETW $9 5
LTI  $8 $8 $9
```

Die LTI-Instruktion speichert eine 1 (TRUE) in Register 8. Anschließend erfolgt ein bedingter Sprung zum Ausgang der IF-Anweisung mit der Instruktion BRF (Branch if FALSE):

```
BRF $8 label_001
```

Ein Label-Bezeichner wie `label_001` repräsentiert eine Instruktionsadresse auf der Assemblerebene, die vom Assembler in eine Hauptspeicheradresse umgesetzt wird. Das Sprungziel wird durch eine Markierung definiert, die aus einer Zeile besteht, in der der Label-Bezeichner gefolgt von einem Doppelpunkt steht. Im Bsp muss diese Markierung hinter dem letzten Maschinenbefehl von STMT stehen, so dass der gesamte Code wie folgt aussieht:

```
    SETW $8 3
    SETW $9 5
    LTI  $8 $8 $9     ; $8 <--  3<5
    BRF $8 label_001  ; falls $8 == FALSE, springe zu label_001
       .
       .         Code für STMT
       .
 label_001:
```

Das Sprungziel ist also die Adresse der ersten Maschineninstruktion, die hinter dem Code der IF-Anweisung folgt. Die Labels werden vom Compiler nach Bedarf generiert und können beliebige Bezeichner sein.

Unterprogrammaufrufe

Caller

Im Caller muss zum Aufruf zunächst Code für die Übergabe der Argumente erzeugt werden. Da die Argumente am Ende des Caller-Stackframe übergeben werden, handelt es sich um Hauptspeicher-Schreibzugriffe, die mit STW-Instruktionen implementiert werden. Die Übergabe eines Arguments ist bei einem Wertparameter vergleichbar mit einer Wertzuweisung: Zieladresse berechnen, Wert berechnen, STW-Instruktion. Bei Referenzparametern muss die Adresse übergeben werden. Die Adressen werden in der Form *SP+Offset* angegeben, wobei der *Offset* eines Arguments aus der Parametertypliste des Callee entnommen werden kann.

Nach der Übergabe der Argumente muss der Einsprung in den Callee und die Übergabe der Rücksprungadresse im Register $30 erfolgen. Dies erledigt die CALL-Instruktion, die das Register für die Rücksprungadresse und das Label für den ersten Maschinenbefehl des Callee als Argumente bekommt.

Beispiel:

```
proc p(i:int, ref j:int) {
  var m: int;
  j := i;
  printi(j);
}

proc q(){
   var k: int;
   k := 1;
   p(k, k);
}
```

Die Offset-Werte sind 0 für das erste und 4 für das zweite Argument von p. Der Code für `p(k, k)`:

```
                ; Zieladresse für 1. Argument berechnen
ADDC $8 $31 0   ; $8 <-- Adresse 1. Argument
                ; Wert des 1. Arguments berechnen: Wert(k)
ADDC $9 $29 -4  ; $9 <-- Adresse(k)
LDW  $9 $9      ; $9 <-- Wert(k)

STW  $9 $8      ; Argument im Stackframe speichern

                ; Zieladresse für 2. Argument berechnen
ADDC $8 $31 4   ; $8 <-- Adresse 2. Argument
                ; Adresse 2. Arguments berechnen: Adresse(k)
ADDC $9 $29 -4  ; $9 <-- Adresse(k)
                ; kein LDW hier, da ein ref-Parameter vorliegt
STW  $9 $8      ; Argument im Stackframe speichern
CALL $30 p      ; Einsprung mit Übergabe der Rücksprungadresse in $30
```

Für die CALL-Instruktion muss vor dem ersten Befehl einer Prozedur ein entsprechendes Label in den Assemblercode eingefügt werden (siehe Prolog von *q* im nächsten Abschnitt).

Callee

Im Callee müssen SP und FP aktualisiert werden, so dass diese auf Ende bzw. Anfang des neuen Stackframe verweisen. Der alte FP muss vor dem Überschreiben gerettet werden. Falls im Callee-Rumpf selbst wieder Aufrufe anderer Prozeduren vorkommen, muss auch Register 30 gerettet werden:

Prolog von q aus dem Beispiel von oben:

```
q:                  ; Prozedurname als Label für den CALL-Befehl
                    ; SP aktualisieren
    SUBC $31 $31 16 ; SP <-- SP - Framesize(q)
                    ; FP retten und danach aktualisieren
    ADDC  $8 $31 8  ; $8 <-- Adresse des Speicherplatzes für FP alt
    STW  $29 $8     ; FP alt speichern
    ADDC $29 $31 16 ; FP neu <-- SP neu + Framesize(q)
                    ; RETURN retten
```

```
ADDC  $8 $29 -12  ; $8 <-- Adresse des Speicherplatzes für RETURN alt
STW  $30 $8       ; RETURN speichern
```

Vor dem Rücksprung müssen alle Aktionen aus dem Prolog wieder rückgängig gemacht, also FP, SP und RETURN wieder auf die alten Werte gesetzt werden. Der Rücksprung erfolg mit JMPR.

Epilog von q:

```
ADDC  $8 $29 -12  ; $8 <-- Adresse des Speicherplatzes für RETURN alt
LDW   $30 $8      ; RETURN restaurieren
ADDC  $8 $31 8    ; $8 <-- Adresse des Speicherplatzes für FP alt
LDW   $29 $8      ; FP alt restaurieren
ADDC  $31 $31 16  ; SP <-- SP + Framesize(q)
JMPR  $30         ; Rücksprung
```

Hinweis: Man beachte, dass bei Prozeduren, die selbst keine anderen Prozeduren aufrufen, die Rettung und das Restaurieren von `$30` entfallen. Der Codegenerator muss also für Prolog und Epilog eine entsprechende Fallunterscheidung machen. Dazu ist entweder für jede Prozedur ein zusätzliches boolesches Attribut (z.B. *isCaller*) nötig, oder die *VarAlloc*-Komponente vergibt z.B. einen negativen Wert für *outgoingArea*.

Direktiven

Für den Linker müssen in Ergänzung zu den Maschineninstruktionen und Labels noch bestimmte Assembler-Direktiven in den Code eingefügt werden:

- Als erste Zeile die Deklaration des Objektmodul-Bezeichners. Der Name sollte der Dateiname ohne die Endung `.spl` sein. Beispiel für `queens.spl`:

  ```
  .object queens
  ```

- Import der Prozeduren aus der Standardbibliothek:

  ```
  .import spllib printi
  .import spllib printc
  .import spllib readi
  .import spllib readc
  .import spllib _indexError
  ```

- Kennzeichnung der Startprozedur *main*:

  ```
  .executable main
  ```