Register Allocation

Consider the following assignment statement:

\[ x = (a*b) + ((c*d) + (e*f)) \]

In postfix notation:

\[ ab*cd*ef*++x \]

Assume that two registers are available. Starting from the left a compiler would generate the following sequence:

- \( l \) \( R1,a \)
- \( mpy \) \( R1,b \)
- \( l \) \( R2,c \)
- \( mpy \) \( R2,d \)

For the next operation \((e*f)\) no register is available and spill is needed. There are two possibilities for the rest of the code:

- \( st \) \( R2,\text{temp} \)
- \( l \) \( R2,e \)
- \( mpy \) \( R2,f \)
- \( add \) \( R2,\text{temp} \)
- \( add \) \( R2,R1 \)
- \( st \) \( R1,x \)
or

```
st R1,temp
l  R1,e
mpy R1,f
add R1,R2
add R1,temp
st  R1,x
```

However, the spill can be avoided if the expression is evaluated in a different order:

```
l  R1,c
mpy R1,d
l  R2,e
mpy R2,f
add R1,R2
l  R2,a
mpy R2,b
add R2,R1
st  R2,x
```
Three address representation

Source-to-source transformations may be applied to the AST.

The next step is to generate three address code.

In many cases, the compound statements (e.g. for or do loops and if statements) are transformed into sequences of instructions which include three-address operations as well as cmp and jump instructions. The original structure of the program (e.g. loops) is recovered by analyzing the program flow graph.

In some cases (e.g. SUIF) information on the high-level constructs is kept as annotations to the three address representation of the program.
A program flow graph is also necessary for compilation. The nodes are the basic blocks. There is an arc from block $B_1$ to block $B_2$ if $B_2$ can follow $B_1$ in some execution sequence.

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halts or possibility of branching except at the end.

**Algorithm BB: Basic Block Partition**

Input: A program PROG in which instructions are numbered in sequence from 1 to $|\text{PROG}|$. INST(i) denotes the ith instruction.

Output
1. the set of LEADERS of initial block instructions.
2. for all x in LEADERS, the set BLOCK(x) of all instructions in the block beginning at x
3. Method:
begin
  LEADERS := \{1\}
  for j := 1 to |PROG| do
    if INST(j) is a branch then
      add the index of each potential target to LEADERS
    fi
  od
  TODO := LEADERS
  while TODO \neq \emptyset do
    x := element of TODO with smallest index
    TODO := TODO - \{x\}
    BLOCK(x) := \{x\}
    for i := x+1 to |PROG| while i \notin LEADERS do
      BLOCK(x) := BLOCK(x) + \{i\}
    od
  od
end
A Simple Code Generator

Our objective is to make a reasonable use of the registers when generating code for a basic block. Consider for example:

\[
\begin{align*}
t &= a - b \\
u &= a - c \\
v &= t + u \\
d &= v + u
\end{align*}
\]

Each instruction could be treated like a macro which expand into something like:

```
l R1,a
sub R1,b
st R1,t
l R1,a
sub R1,c
st R1,u
l R1,t
add R1,u
st R1,v
l R1,v
add R1,u
st R1,d
```

The resulting code is not very good. Only one register is used and there is much redundant code. A more sophisticated algorithm is needed.
Target Machine Language

We use the following target machine language:

The machine has two address instructions of the form

\[ \text{op} \ \text{destination}, \text{source} \]

The destination has to be a register. The machine has several op-codes including

- \( l \) (move source to destination)
- \( \text{add} \) (add source to destination)
- \( \text{sub} \) (subtract source from destination)

There is also a store (\( \text{st} \)) instruction.

The source (destination in the case of the store instruction) can be

1. an absolute memory address (a variable name is used),
2. a register,
3. indexed (written \( c(R) \), where \( c \) is a constant and \( R \) a register),
4. indirect (written \( R \) where \( R \) is a register), and
5. immediate (denoted \( \#c \) where \( c \) is a constant)
Algorithm SCG: A Simple Code Generator

Input:
1. A basic block of three address statements.
2. A symbol table SYMTAB

Output:
1. Machine code

Intermediate:
1. A register descriptor RD(R). The set variables whose values are in register \( R \)
2. An address descriptor AD(variable). The set of locations (register, stack, memory) where the value of variable can be found.

\[
\begin{array}{c|c|c|c|c|c|c}
R & Var & & & & & \\
\hline
1 & x & y & & & & \text{RD(R)} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
AD & Ri... & Mem & Stack & & & \\
\hline
x & r1,r2 & & & & & \text{AD(var)} \\
\end{array}
\]

Assumption: save all live variables at the end of the BB (conservative)
begin
  for each instruction I in basic block do
  if I is of form (x := y op z) then
    L := getreg(y, I);
    if L not in AD(y) then
      y' := select (y)
      generate(l L, y')
    endif
    z' := select (z)
    generate(op L, z')
    AD(y) := AD(y) - {L}
  endfor
  for all R in REGISTERS do
    RD(R) := RD(R) - {x}
  endfor
  RD(L) := {x}
  AD(x) := {L}
  if NEXTUSE(y, I) empty and
    LIVEONEXIT(y) is false then
    forall R in REGISTERS do
      RD(R) := RD(R) - {y}
    endforall
  endif
  ...same as above for z...
  elseif I is of form (x := op y) then
    ... similar code as above...
  elseif I is of form (x := y) then
    if there is register R in AD(y) then
      RD(R) := RD(R) + {x}
      AD(x) := {R}
else
    \[ L := \text{getreg}(y, I) \]
    \[ \text{generate}(L, y) \]
    \[ \text{RD}(L) := \{ x, y \} \]
    \[ \text{AD}(x) = \{ L \} \]
    \[ \text{AD}(y) = \text{AD}(y) + \{ L \} \]
endif
endif
enddo
forall \, R \, \text{in} \, \text{REGISTERS} \, \text{do}
    forall \, v \, \text{in} \, \text{RD}(R) \, \text{do}
        forall
            \text{if} \quad \text{LIVEONEXIT}(v)
                \text{and} \quad \text{SYMTAB.loc}(v) \, \text{not in} \, \text{AD}(v) \, \text{then}
                    (\text{only once for each} \, v)
                    \text{generate(st} \, \, R, v)
            \text{endif}
        endforall
    endforall
endforall
end

The \textit{select} routine returns a register \( R \) if the value of the parameter is in \( R \) otherwise it returns the memory location containing the value of the parameter.
**NEXTUSE and LIVEONEXIT**

The LIVEONEXIT(\(v\)) boolean value is true if \(v\) may be used after the basic block completes. It is computed using global flow analysis techniques to be discussed later in the course.

The NEXTUSE(\(v, I\)) is the statement number where \(v\) is used next in the basic block. It is empty if \(v\) is not used again.

NEXTUSE(\(v, I\)) can be computed as follows:

\[
\text{forall variables } v \text{ in basicblock do}
\text{USE}(v) := \{\}
\text{od}
\]

\[
\text{forall instructions } I \text{ in basic block in reverse order do}
\text{if } I \text{ is of form } (x := y \text{ op } z) \text{ then}
\text{NEXTUSE}(x, I) = \text{USE}(x)
\text{NEXTUSE}(y, I) = \text{USE}(y)
\text{NEXTUSE}(z, I) = \text{USE}(z)
\text{USE}(x) = \{\}
\text{USE}(y) := \{I\}
\text{USE}(z) := \{I\}
\text{elseif } \ldots
\text{fi}
\text{od}
\]
**getreg**(y, I)

if there is register \( R \) such that \( RD(R) = \{ y \} \) and both \( LIVEONEXIT(y) \) and \( NEXTUSE(y, I) \) are empty then return (\( R \))

else if there is \( R \) in \( REGISTERS \) such that \( RD(R) \) empty then return(\( R \))

else

\( R := \) getanyregister()

forall \( v \) in \( RD(R) \) do

\( AD(v) := AD(v) - \{ R \} \)

if SYMTAC.loc(\( v \)) is not in \( AD(v) \) then

generate(\( stR, v \))

\( AD(v) := AD(v) + \{ SYMTAB.loc(v) \} \)

endif

endforall

dendif
Two special operators

The [ ] operator is used to index a (one dimensional) array

\[ a := b[i] \]

can be translated as

(1)

1 \ R, b (R)

if \( i \) is in register \( R \)

(2)

1 \ R, M
1 \ R, b (R)

if \( i \) is memory location \( M \)

(3)

1 \ R, S (A)
1 \ R, b (R)

if \( i \) is in stack offset \( S \).

The * operator is similar. For example, (2) above is replaced by

1 \ R, M
1 \ R, *R