16 Code generation 1 - Register Allocation

What is the register allocation problem?

Most computers are based on a mutiple general purpose register architecture. Others have separate sets of base (addressing) and data registers. In both cases the number of registers is finite, although some RISC machines have significantly more registers than the 8 or 16 in traditional machines. Registers are needed both for arithmetic and for address calculation and referencing of data in memory.

Thus there is a need to allocate registers as computation proceeds, keeping track of those currently in use and those which no longer contain values of use in the current instruction. A simple minded register allocation policy merely has a flag for each register indicating whether it is in use or not. This is reset as soon as the registers value is used for its current purpose.

A more general view of the same problem would note that the values in registers may still be needed beyond the end of the current instruction or statement and that, in fact, they are frequently stored into memory and subsequently reloaded into registers. In many cases this storing is redundant if the value could instead be kept in the same register between uses. Indeed, if there are sufficient registers available it would be best to keep all values in registers at all times. Since the number of registers is finite and there are times, such as procedure calls, when code may be entered which does not share the same picture of register allocation, such a policy is generally impractical. Thus the ideal register allocation problem is to minimise the number of unnecessary stores and loads of values into and from memory, subject to the constraints of the machine and the programs data flows.

In practice this problem is difficult to optimise fully. It can be thought of as equivalent to graph colouring, if each values to be stored is assigned a node in a conflict graph and values which must co-exist are linked by an edge. The registers are then the colours of the graph. The execution of the program yields a series of sub-graphs, as some values are killed and new ones are generated. The goal is to colour this series so that each graph requires no shuffling of colours to derive its successor, which is the same as colouring the whole graph. In real compilers various heuristics are used instead of this hard optimisation.

Next use of values

To track the use of values within a basic block, their next use may be established by a backwards pass throught the tuples of that block. If control flow analysis has not been done earlier, partitioning on the fly is simple, as we are only interested the boundaries between blocks, not their links in a flow graph.

Final liveness can be assumed for all variables or deduced from earlier DAG analysis. Each time a statement such as

$$V \quad := \quad O_1 \; op \; O_2$$

is found

(a) this tuple has recorded for it the current liveness and next use properties of V, O_1 and O_2;

(b) O_1 and O_2 are recorded as next used in this tuple and as live to this point;

(c) V is recorded as no next use and killed here.

Recording register allocation

There are many variants on remembering register allocation. The aim is to remember as much about the current contents of a register as is possible and safe, to avoid reloading. In general the idea is to track through each basic block the values stored in each register by storing the names of such values (for constants the actual value or its string of digits etc.) in the descriptor for that register. A register may have more than one name associated with its value if a copy assignment such as `A:=B` is treated as associating both `A` and `B` with the same register.

Address descriptors

Each named value will be stored in the symbol table. A record of the location or locations where it is currently held can be maintained there, also. A value may be in more than one location if, for instance it has been loaded into a register in an earlier operation and not yet killed there. It is important to maintain consistency between multiple copies of the same value while minimising the writing back to memory.

Generating code for assignments

To illustrate the way this information can be used, we consider the generation of code for a two operand assignment of the form considered above. Other forms of statement are handled in similar ways, although they may require their own special cases to be handled.

To generate code for an assignment we might:

1. Determine where to place the result, V. This would typically be a register. Following Aho and Ullman we call this location L. We will cosider this more fully below.

2. Find a location holding the first operand, O_1,. We prefer a register to a memory location holding the same value. For a two operand machine code, we move O_1 into L.

3. Generate the machine instruction

`op O2, L`	for a two operand machine
`op O1,O2,L`	for a three operand machine

   Update L's descriptor to show it holds A.

4. If O_1 and O_2 have no next use and are not live on exit, mark any registers holding them as free for re-use.

An alternative to treating the three operand machine specially exists on machines like the VAX where both two and three operand forms exist for most instructions. There it is possible to generate for the two operand form and use a peephole optimiser to replace this later.

Selecting the destination

The problem of selecting an appropriate destination is not straightforward in all cases, either. Ideally we would replace the value in the register holding O_1 or O_2. If the original value in this register is required again later, however, we cannot overwrite it. Where V and O_1 or O_2 are the same variable it is clearly desirable, but other names sharing the old value must be protected. In general the following may be safe:

1. If O_1 is in a register with no other names attached and no next use and is not live on exit, use it, cancelling its descriptor.

2. Otherwise use an empty register if one is free.
3. Otherwise store a used register to memory and use that. Where a register holds a value that is also in memory use that register and update the descriptor accordingly. (But beware storing a value needed immediately afterwards.)
4. If all else fails and the machine allows direct use of memory locations, use the location of O_1.

To allow for case (??) this function must return a descriptor capable of recording both a register and a memory location.

Exploiting algebraic properties

Clearly the destination and other register allocations can be tuned by exploiting the commutative properties of some operators, allowing either O_1 or O_2 to be the first operand in the actual macine code. This is quite straightforward in most cases. Other algebraic properties may also be used with care, such as adding the negated form of a constant value to replace a subtraction with a commutative addition.

Global register allocation

As mentioned at the start of this note, fully optimal global register allocation is not practical in most cases. Instead, further heuristics are used to produce good code.

A simple decision is to allocate some registers permanently to certain purposes, such as loop control variables (assuming nesting is not too deep). Some are always reserved for return values from functions etc. This approach has limited use in practice. If mixed language programming is to be supported, such as C programs calling functions from Fortran libraries, both compilers must respect the same conventions about reserving registers across procedure calls.

A more flexible approach is to try to relax the saving of registers at the end of all basic blocks. First, it is always possible to save only those actually live at block end. This is allowable even for a procedure call.

The other common approach is to try to preserve registers use across loops. This requires setting aside some registers and allocating them to the most frequently used or stored/loaded values. This is assessed in practice by attaching weights to different uses of values within the loops blocks and choosing those values with the highest weights to keep in registers. The key is to attach high weights to expensive operations which are much cheaper if the value is already in a register.

The usage counting can be performed most usefully on innermost loops and then, if there are spare registers, on enclosing loops and so on.

A note on RISC machines

As noted earlier, RISC machines have often got many more registers than traditional architectures. This relaxes many of the constraints on register allocation and makes optimal allocation more practical. In terms of graph colouring, the number of clours is much greater. It is still important to track register use, however, to avoid unnecessary copying.

RISC machines often go as far as using a sliding register window to pass parameters. This works well for C, which has little value passing of structures, or for Fortran, which passes everything by reference. It works much less well for Pascal.