Chapter 9

Register Allocation

9.1 Introduction

When generating intermediate code in chapter 7, we have freely used as many variables as we found convenient. In chapter 8, we have simply translated variables in the intermediate language one-to-one into registers in the machine language. Processors, however, do not have an unlimited number of registers, so we need *register allocation* to handle this conflict. The purpose of register allocation is to map a large number of variables into a small(ish) number of registers. This can often be done by letting several variables share a single register, but sometimes there are simply not enough registers in the processor. In this case, some of the variables must be temporarily stored in memory. This is called *spilling*.

Register allocation can be done in the intermediate language prior to machine-code generation, or it can be done in the machine language. In the latter case, the machine code initially uses symbolic names for registers, which the register allocation turns into register numbers. Doing register allocation in the intermediate language has the advantage that the same register allocator can easily be used for several target machines (it just needs to be parameterised with the set of available registers).

However, there may be advantages to postponing register allocation to after machine code has been generated. In chapter 8, we saw that several instructions may be combined to a single instruction, and in the process a variable may disappear. There is no need to allocate a register to this variable, but if we do register allocation in the intermediate language we will do so. Furthermore, when an intermediate-language instruction needs to be translated into a sequence of machine-code instructions, the machine code may need an extra register (or two) for storing temporary values (such as the register needed to store the result of the `SLT` instruction when translating a jump on $<$ to MIPS code). Hence, the register allocator must make sure that there is always at least one spare register for temporary storage.

The techniques used for register allocation are more or less the same regardless of whether register allocation is done on intermediate code or on machine code. So, in this chapter, we will describe register allocation in terms of the intermediate language introduced in chapter 7.

As in chapter 7, we operate on the body of a single procedure or function, so when we below use the word "program", we mean it to be such a body. In chapter 10, we will look at how to handle programs consisting of several functions that can call each other.

9.2 Liveness

In order to answer the question "When can two variables share a register?", we must first define the concept of *liveness*:

Definition 9.1 *A variable is* live *at some point in the program if the value it contains at that point might conceivably be used in future computations. Conversely, it is* dead *if there is no way its value can be used in the future.*

We have already hinted at this concept in chapter 8, when we talked about last-uses of variables.

Loosely speaking, two variables may share a register if there is no point in the program where they are both live. We will make a more precise definition later.

We can use some rules to determine when a variable is live:

1) If an instruction uses the contents of a variable, that variable is *live* at the start of that instruction.

2) If a variable is assigned a value in an instruction, and the same variable is not used as an operand in that instruction, then the variable is *dead* at the start of the instruction, as the value it has at this time is not used before it is overwritten.

3) If a variable is live at the end of an instruction and that instruction does not assign a value to the variable, then the variable is also live at the start of the instruction.

4) A variable is live at the end of an instruction if it is live at the start of any of the immediately succeeding instructions.

Rule 1 tells how liveness is *generated*, rule 2 how liveness is *killed*, and rules 3 and 4 how liveness is *propagated*.

9.3 Liveness analysis

We can formalise the above rules as equations over sets of variables. The process of solving these equations is called *liveness analysis*, and will at any given point in the program determine which variables are live at this point. To better speak of points in a program, we number all instructions as in figure 9.2.

For every instruction in the program, we have a set of *successors*, *i.e.*, instructions that may immediately follow the instruction during execution. We denote the set of successors to the instruction numbered i as $succ[i]$. We use the following rules to find $succ[i]$:

1) The instruction numbered j (if any) that is listed just after instruction number i is in $succ[i]$, unless i is a `GOTO` or `IF-THEN-ELSE` instruction. If instructions are numbered consecutively, $j = i + 1$.

2) If instruction number i is of the form `GOTO` l, (the number of) the instruction `LABEL` l is in $succ[i]$. Note that there in a correct program will be exactly one `LABEL` instruction with the label used by the `GOTO` instruction.

3) If instruction i is `IF` p `THEN` l_t `ELSE` l_f, (the numbers of) the instructions `LABEL` l_t and `LABEL` l_f are in $succ[i]$.

Note that we assume that both outcomes of an `IF-THEN-ELSE` instruction are possible. If this happens not to be the case (*i.e.*, if the condition is always true or always false), our liveness analysis may claim that a variable is live when it is in fact dead. This is no major problem, as the worst that can happen is that we use a register for a variable that is not going to be used after all. The converse (claiming a variable dead when it is, in fact, live) is worse, as we may overwrite a value that could be used later on, and hence get wrong results from the program. Precise liveness informatin depends on knowing exactly which paths a program may take through the code when executed, and this is not possible to compute exactly (it is a formally undecidable problem), so it is quite reasonable to allow imprecise results from a liveness analysis, as long as we err on the side of safety, *i.e.*, calling a variable live unless we can prove it to be dead

For every instruction i, we have a set $gen[i]$, which lists the variables that may be read by instruction i and, hence, are live at the start of the instruction. In other words, $gen[i]$ is the set of variables that instruction i *generates* liveness for. We also have a set $kill[i]$ that lists the variables that may be assigned a value by the instruction. Figure 9.1 shows which variables are in $gen[i]$ and $kill[i]$ for the types of instruction found in intermediate code. x, y and z are (possibly identical) variables and k denotes a constant.

Instruction i	$gen[i]$	$kill[i]$
`LABEL` l	$\emptyset$	$\emptyset$
$x := y$	$\{y\}$	$\{x\}$
$x := k$	$\emptyset$	$\{x\}$
$x :=$ **unop** y	$\{y\}$	$\{x\}$
$x :=$ **unop** k	$\emptyset$	$\{x\}$
$x := y$ **binop** z	$\{y,z\}$	$\{x\}$
$x := y$ **binop** k	$\{y\}$	$\{x\}$
$x := M[y]$	$\{y\}$	$\{x\}$
$x := M[k]$	$\emptyset$	$\{x\}$
$M[x] :=$ y	$\{x,y\}$	$\emptyset$
$M[k] :=$ y	$\{y\}$	$\emptyset$
`GOTO` l	$\emptyset$	$\emptyset$
`IF` x **relop** y `THEN` l_t `ELSE` l_f	$\{x,y\}$	$\emptyset$
$x :=$ `CALL` $f(args)$	$args$	$\{x\}$

Figure 9.1: Gen and kill sets

For each instruction i, we use two sets to hold the actual liveness information: $in[i]$ holds the variables that are live at the start of i, and $out[i]$ holds the variables that are live at the end of i. We define these by the following equations:

$$in[i] = gen[i] \cup (out[i] \setminus kill[i]) \tag{9.1}$$

$$out[i] = \bigcup_{j \in succ[i]} in[j] \tag{9.2}$$

These equations are recursive. We solve these by fixed-point iteration, as shown in appendix A: We initialise all $in[i]$ and $out[i]$ to be empty sets and repeatedly calculate new values for these until no changes occur. This will eventually happen, since we work with sets with finite support (*i.e.*, a finite number of possible values) and because adding elements to the sets $out[i]$ or $in[j]$ on the right-hand sides of the equations can not reduce the number of elements in the sets on the left-hand sides. Hence, each iteration will either add elements to some set (which we can do only a finite number of times) or leave all sets unchanged (in which case we are done). It is also easy to see that the resulting sets form a solution to the equation – the last iteration essentially verifies that all equations hold. This is a simple extension of the reasoning used in section 2.6.1.

The equations work under the assumption that all uses of a variable are visible in the code that is analysed. If a variable contains, *e.g.*, the output of the program,

```
1:  a := 0
2:  b := 1
3:  z := 0
4:  LABEL loop
5:  IF n = z THEN end ELSE body
6:  LABEL body
7:  t := a + b
8:  a := b
9:  b := t
10: n := n − 1
11: z := 0
12: GOTO loop
13: LABEL end
```

Figure 9.2: Example program for liveness analysis and register allocation

it will be used after the program finishes, even if this is not visible in the code of the program itself. So we must ensure that the analysis makes this variable live at the end of the program.

Equation 9.2, similarly, is ill-defined if $succ[i]$ is the empty set (which is, typically, the case for any instruction that ends the program), so we make a special case: $out[i]$, where i has no successor, is defined to be the set of all variables that are live at the end of the program. This definition replaces (for these instructions only) equation 9.2.

Figure 9.2 shows a small program that we will calculate liveness for. Figure 9.3 shows *succ*, *gen* and *kill* sets for the instructions in the program.

The program in figure 9.2 calculates the Nth Fibonacci number (where N is given as input by initialising n to N prior to execution). When the program ends (by reaching instruction 13), a will hold the Nth fibonacci number, so a is live at the end of the program. Instruction 13 has no successors ($succ[13] = \emptyset$), so we set $out[13] = \{a\}$. The other *out* sets are defined by equation 9.2 and all *in* sets are defined by equation 9.1. We initialise all *in* and *out* sets to the empty set and iterate until we reach a fixed point.

The order in which we treat the instructions does not matter for the final result of the iteration, but it may influence how quickly we reach the fixed-point. Since the information in equations 9.1 and 9.2 flow backwards through the program, it is a good idea to do the evaluation in reverse instruction order and to calculate $out[i]$ before $in[i]$. In the example, this means that we will in each iteration calculate the sets in the order

i	$succ[i]$	$gen[i]$	$kill[i]$
1	2		a
2	3		b
3	4		z
4	5		
5	6, 13	n, z	
6	7		
7	8	a, b	t
8	9	b	a
9	10	t	b
10	11	n	n
11	12		z
12	4		
13			

Figure 9.3: *succ*, *gen* and *kill* for the program in figure 9.2

$$out[13], in[13], out[12], in[12], \ldots, out[1], in[1]$$

Figure 9.4 shows the fixed-point iteration using this backwards evaluation order. Note that the most recent values are used when calculating the right-hand sides of equations 9.1 and 9.2, so, when a value comes from a higher instruction number, the value from the same column in figure 9.4 is used.

We see that the result after iteration 3 is the same as after iteration 2, so we have reached a fixed point. We note that n is live at the start of the program, which is to be expected, as n is expected to hold the input to the program. If a variable that is not expected to hold input is live at the start of a program, it might in some executions of the program be used before it is initialised, which is generally considered an error (since it can lead to unpredictable results and even security holes). Some compilers issue warnings about uninitialised variables and some compilers add instructions to initialise such variables to a default value (usually 0).

Suggested exercises: 9.1(a,b).

9.4 Interference

We can now define precisely the condition needed for two variables to share a register. We first define *interference*:

	Initial		Iteration 1		Iteration 2		Iteration 3	
i	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$
1			n,a	n	n,a	n	n,a	n
2			n,a,b	n,a	n,a,b	n,a	n,a,b	n,a
3			n,z,a,b	n,a,b	n,z,a,b	n,a,b	n,z,a,b	n,a,b
4			n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b
5			a,b,n	n,z,a,b	a,b,n	n,z,a,b	a,b,n	n,z,a,b
6			a,b,n	a,b,n	a,b,n	a,b,n	a,b,n	a,b,n
7			b,t,n	a,b,n	b,t,n	a,b,n	b,t,n	a,b,n
8			t,n	b,t,n	t,n,a	b,t,n	t,n,a	b,t,n
9			n	t,n	n,a,b	t,n,a	n,a,b	t,n,a
10				n	n,a,b	n,a,b	n,a,b	n,a,b
11					n,z,a,b	n,a,b	n,z,a,b	n,a,b
12					n,z,a,b	n,z,a,b	n,z,a,b	n,z,a,b
13			a	a	a	a	a	a

Figure 9.4: Fixed-point iteration for liveness analysis

Definition 9.2 *A variable x interferes with a variable y if $x \neq y$ and there is an instruction i such that $x \in kill[i]$, $y \in out[i]$* and *instruction i is not $x := y$.*

Two different variables can share a register precisely if neither interferes with the other. This is almost the same as saying that they should not be live at the same time, but there are small differences:

- After $x := y$, x and y may be live simultaneously, but as they contain the same value, they can still share a register.

- It may happen that x is not in $out[i]$ even if x is in $kill[i]$, which means that we have assigned to x a value that is definitely not read from x later on. In this case, x is not technically live after instruction i, but it still interferes with any y in $out[i]$. This interference prevents an assignment to x overwriting a live variable y.

The first of these differences is essentially an optimisation that allows more sharing than otherwise, but the latter is important for preserving correctness. In some cases, assignments to dead variables can be eliminated, but in other cases the instruction may have another visible effect (*e.g.*, setting condition flags or accessing memory) and hence can not be eliminated without changing program behaviour.

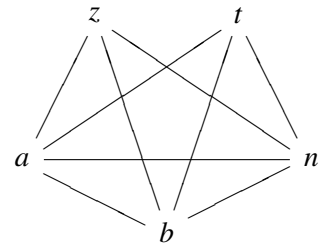

Figure 9.5: Interference graph for the program in figure 9.2

We can use definition 9.2 to generate interference for each assignment statement in the program in figure 9.2:

Instruction	Left-hand side	Interferes with
1	a	n
2	b	n,a
3	z	n,a,b
7	t	b,n
8	a	t,n
9	b	n,a
10	n	a,b
11	z	n,a,b

We will do *global register allocation*, *i.e.*, find for each variable a register that it can stay in at all points in the program (procedure, actually, since a "program" in terms of our intermediate language corresponds to a procedure in a high-level language). This means that, for the purpose of register allocation, two variables interfere if they do so at *any* point in the program. Also, even though interference is defined in an assymetric way in definition 9.2, the conclusion that the two involved variables cannot share a register is symmetric, so interference defines a symmetric relation between variables. A variable can never interfere with itself, so the relation is not reflective.

We can draw interference as an undirected graph, where each node in the graph is a variable, and there is an edge between nodes x and y if x interferes with y (or *vice versa*, as the relation is symmetric). The *interference graph* for the program in figure 9.2 is shown in figure 9.5.

9.5 Register allocation by graph colouring

Two variables can share a register if they are not connected by an edge in the interference graph. Hence, we must assign to each node in the interference graph a register number such that:

1) Two nodes that share an edge have different register numbers.

2) The total number of different register numbers is no higher than the number of available registers.

This problem is well-known in graph theory, where it is called *graph colouring* (in this context a "colour" is a register number). It is known to be NP-complete, which means that no effective (*i.e.*, polynomial-time) method for doing this optimally is known. In practice, this means that we need to use a heuristic method, which will often find a solution but may give up in some cases even when a solution does exist. This is no great disaster, as we must deal with non-colourable graphs anyway (by moving some variables to memory), so at worst we get slightly slower programs than we would get if we could colour the interference graphs optimally.

The basic idea of the heuristic method we use is simple: If a node in the graph has strictly fewer than N edges, where N is the number of available colours (*i.e.*, registers), we can set this node aside and colour the rest of the graph. When this is done, the (at most $N-1$) nodes connected by edges to the selected node can not possibly use all N colours, so we can always pick a colour for the selected node from the remaining colours.

We can use this method to four-colour the interference graph from figure 9.5:

1) z has three edges, which is strictly less than four. Hence, we remove z from the graph.

2) Now, a has less than four edges, so we also remove this.

3) Only three nodes are now left (b, t and n), so we can give each of these a number, *e.g.*, 1, 2 and 3 respectively for nodes b, t and n.

4) Since three nodes (b, t and n) are connected to a, and these use colours 1, 2 and 3, we must choose a fourth colour for a, *e.g.,* 4.

5) z is connected to a, b and n, so we choose a colour that is different from 4, 1 and 3. Giving z colour 2 works.

The problem comes if there are no nodes that have less than N edges. This in itself does not imply that the graph is uncolourable. As an example, a graph with four nodes arranged and connected as the corners of a square can, even though all nodes

have two neighbours, be coloured with two colours by giving opposite corners the same colour. This leads to the following so-called "optimistic" colouring heuristics:

Algorithm 9.3

initialise: *Start with an empty stack.*

simplify: *If there is a node with less than N edges, put this on the stack along with a list of the nodes it is connected to, and remove it and its edges from the graph.*

If there is no node with less than N edges, pick any node and do as above.

If there are more nodes left in the graph, continue with **simplify***, otherwise go to* **select**.

select: *Take a node and its list of connected nodes from the stack. If possible, give the node a colour that is different from the colours of the connected nodes (which are all coloured at this point). If this is not possible, colouring fails and we mark the node for spilling (see below).*

If there are more nodes on the stack, continue with **select**.

The idea in this algorithm is that, even though a node has N or more edges, some of the nodes it is connected to may have been given identical colours, so the total number of colours used for these nodes is less than N. If this is the case, we can use one of the unused colours. If not, we must mark the node for spill.

There are several things left unspecified by algorithm 9.3:

- Which node to choose in **simplify** when none have less than N edges, and
- Which colour to choose in **select** if there are several choices.

If we choose perfectly in both cases, algorithm 9.3 will do optimal colouring. But perfect choices are costly to compute so, in practice, we will sometimes have to guess. We will, in section 9.7, look at some ideas for making qualified guesses. For now, we just make arbitrary choices.

Suggested exercises: 9.1(c,d).

9.6 Spilling

If the **select** phase is unable to find a colour for a node, algorithm 9.3 cannot colour the graph. This means we must give up on keeping all variables in registers throughout the program. We must, hence, select some variables that will reside in memory

(except for brief periods). This process is called *spilling*. Obvious candidates for spilling are variables at nodes that are not given colours by **select**. We simply mark these as *spilled* and continue **select** with the rest of the stack, ignoring spilled nodes when selecting colours for the remaining nodes. When we finish algorithm 9.3, several variables may be marked as spilled.

When we have chosen one or more variables for spilling, we change the program so these are kept in memory. To be precise, for each spilled variable x we:

1) Choose a memory address $address_x$ where the value of x is stored.

2) In every instruction i that reads or assigns x, we locally in this instruction rename x to x_i.

3) Before an instruction i that reads x_i, insert the instruction $x_i := M[address_x]$.

4) After an instruction i that assigns x_i, insert the instruction $M[address_x] := x_i$.

5) If x is live at the start of the program, add an instruction $M[address_x] := x$ to the start of the program. Note that we use the original name for x here.

6) If x is live at the end of the program, add an instruction $x := M[address_x]$ to the end of the program. Note that we use the original name for x here.

After this rewrite of the program, we do register allocation again. This includes re-doing the liveness analysis, since we have added new variables x_i and changed the liveness of x. We may optimise this a bit by repeating the liveness analysis only for the affected variables (x_i and x), as the results will not change for the other variables.

It may happen that the subsequent new register allocation will generate additional spilled variables. There are several reasons why this may be:

- We have ignored spilled variables when selecting colours for a node in the **select** phase. When the spilled variables are replaced by new variables, these may use colours that would otherwise be available, so we may end up with no choices where we originally had one or more colours available.

- The choices of nodes to remove from the graph in the **simplify** phase and the colours to assign in the **select phase** can change, and we might be less lucky in our choices, so we get more spills.

If we have at least as many registers as the number of variables used in a single instruction, all variables can be loaded just before the instruction, and the result can be saved immediately afterwards, so we will eventually be able to find a colouring

Node	Neighbours	Colour
n		1
t	n	2
b	t,n	3
a	b,n,t	*spill*
z	a,b,n	2

Figure 9.6: Algorithm 9.3 applied to the graph in figure 9.5

by repeated spilling. If we ignore the `CALL` instruction, no instruction in the intermediate language uses more than two variables, so this is the minimum number of registers that we need. A `CALL` instruction can use an unbounded number of variables as arguments, possibly even more than the total number of registers available, so it is unrealistic to expect all arguments to function calls to be in registers. We will look at this issue in chapter 10.

If we take our example from figure 9.2, we can attempt to colour its interference graph (figure 9.5) with only three colours. The stack built by the **simplify** phase of algorithm 9.3 and the colours chosen for these nodes in the **select** phase are shown in figure 9.6. The stack grows upwards, so the first node chosen by **simplify** is at the bottom. The colours (numbers) are, conversely, chosen top-down as the stack is popped. We can choose no colour for a, as all three available colours are in use by the neighbours b,n and t. Hence, we mark a as spilled. Figure 9.7 shows the program after spill code has been inserted. Note that, since a is live at the end of the program, we have inserted a load instruction at the end of the program. Figure 9.8 shows the interference graph for the program in figure 9.7and figure 9.9 shows the stack used by algorithm 9.3 for colouring this graph, showing that colouring with three colours is now possible.

Suggested exercises: 9.1(e).

9.7 Heuristics

When the **simplify** phase of algorithm 9.3 is unable to find a node with less than N edges, some other node is chosen. So far, we have chosen arbitrarily, but we may apply some heuristics (qualified guessing) to the choice in order to make colouring more likely or reduce the number of spilled variables:

- We may choose a node with close to N neighbours, as this is likely to be colourable in the **select** phase anyway. For example, if a node has exactly N