15
Code Generation

In this chapter we will explore how intermediate forms, like JVM bytecodes, are translated into executable form. Collectively, this process is called *code generation*, though code generation actually involves a number of individual tasks that must be handled.

Translation has already been done by code generation subroutines associated with AST nodes (Chapters 12 to 14). Source level constructs have been transformed in bytecodes. Bytecodes may be interpreted by a byte code interpreter. Alternatively, we may wish to complete the translation process and produce machine instructions native to the computer we are using.

The first problem we will face is *instruction selection*. Instruction selection is highly target machine dependent; it involves choosing a particular instruction sequence to implement a given JVM bytecode. Even for a simple bytecode we may have a choice of possible implementations. For example, an `iinc` instruction, that adds a constant to a local variable, might be implemented by loading a variable into a register, loading the constant into a second register, doing a register to register add, and storing the result back into the variable. Alternatively, we might choose to keep the variable in a register, allowing implementation using only a single add immediate instruction.

Besides instruction selection, we must also deal with r*egister allocation* and *code scheduling*. Register allocation aims to use registers effectively, by minimizing *register spilling* (storing a value held in a register and reloading the register with

something else). Even a few unnecessary loads and stores can significantly reduce the speed of an instruction sequence. *Code scheduling* worries about the order in which generated instructions are executed. Not all valid instruction ordering are equally good—some incur unnecessary delays that must be avoided.

We shall first consider how bytecodes may be efficiently translated to machine-level instructions. Techniques that optimize the code we generate will be considered next, especially the translation of expressions in tree forms. A variety of techniques that support efficient use of registers, especially *graph coloring* will be discussed. Approaches to code scheduling will next be studied. Techniques that allow us to easily and automatically retarget a code generator to a new computer will then be discussed. Finally, a form of optimization that is particularly useful at the code generation level—peephole optimization—will be studied.

15.1 Translating Bytecodes

We will first consider how to translate the bytecodes generated by the translation phase of our compiler into conventional machine code. Many different machine instruction sets exist; one for each computer architecture. Examples include the Intel x86 architecture, the SPARC, the Alpha, the Power PC, and the MIPS.

In this chapter we'll use the MIPS R3000 instruction set. This architecture is clean and easy to use and yet represents well the current generation of RISC architectures. The MIPS R3000 also is supported by SPIM [Lar 93] a widely-available MIPS interpreter written in C.

Most bytecodes map directly into one or two MIPS instructions. Thus an `iadd` instruction corresponds directly to the MIPS `add` instruction. The biggest difference in the design of bytecodes and the MIPS (or any other modern architecture) is that bytecodes are stack-oriented whereas the MIPS is register-oriented.

The most obvious approach to handling stack based operands is to load top of stack values into registers when they are used, and to push registers onto the stack when values are computed. This unfortunately is also one of the *worst* approaches. The problem is that explicit pop and push operations imply memory load and store instructions which can be slow and bulky.

Instead, we'll make a few simple, but important, observations on how stack operands are used. First, note that no operands are left on the stack between source-level statements. If they were, a statement, placed in a loop, could cause stack overflow. Thus the stack is used only to hold operands while parts of a statement are executed. Moreover, each stack operand is "touched" twice—when it is created (pushed) and when it is used (popped).

These observations allow us to map stack operands directly into registers—no pushes or pops of the run-time stack are really needed. We can imagine the JVM operand stack as containing register names rather than values. When a particular value is at the top of the stack, we'll use the corresponding "top register" as the source of our operand. This may seem complex, but it really is quite simple. Consider the Java assignment statement `a = b + c - d;` (where `a`, `b`, `c` and `d` are integer locals). The corresponding bytecodes are

```
iload 2  ; Push int b onto stack
iload 3  ; Push int c onto stack
iadd     ; Add top two stack values
iload 4  ; Push int d onto stack
isub     ; Subtract top two stack values
istore 1 ; Store top stack value into a
```

Whenever a value is pushed, we will create a temporary location to hold it. This temporary location (usually just called a temporary) will normally be allocated to a register. We'll track the names of the temporaries associated with stack locations as bytecodes are processed. At any point, we'll know exactly what temporaries are logically on the stack. We say logically, because these values aren't pushed and popped at run time. Rather, values are directly accessed from the registers that hold them.

Continuing with our example, assume `a`, `b`, `c` and `d` are assigned frame offsets 12, 16, 20 and 24 respectively (we'll discuss memory allocation for locals and fields below). These four variables are given offsets because, as discussed in Chapter 11, local variables in a procedure or method are allocated as part of a frame—a block of memory allocated (on the run-time stack) whenever a call is made. Thus rather than push or pop individual data values, as bytecodes do, we prefer to push a single large block of memory once per call.

Let's assume the temporaries we allocate are MIPS registers, denoted `$t0`, `$t1`, ... Each time we generate code for a bytecode that pushes a value onto the stack, we'll call getReg (Section 15.3.1) to allocate a result register. Whenever we generate MIPS code for a bytecode that accesses stack values, we'll use the registers already allocated to hold stack values. The net effect is to use registers rather than stack locations to hold operands, which yields fast and compact instruction sequences. For our above example we might generate

```
lw    $t0,16($fp)    # Load b, at 16+$fp, into $t0
lw    $t1,20($fp)    # Load c, at 20+$fp, into $t1
add   $t2,$t0,$t1    # Add $t0 and $t1 into $t2
lw    $t3,24($fp)    # Load d, at 24+$fp, into $t3
sub   $t4,$t2,$t3    # Subtract $t3 from $t2 into $t4
sw    $t4,12($fp)    # Store result into a, at 12+$fp
```

The `lw` instruction loads a word of memory into a register. Addresses of locals are computed relative to `$fp`, the frame pointer, which always points to the currently active frame. Similarly, `sw` stores a register into a word of memory. `add` and `sub` add (or subtract) two registers, putting the result into a third register.

Bytecodes that push constants, like `bipush n`, can be implemented as a load immediate of a literal into a register. As an optimization, we can delay doing the actual load until we're sure it is needed. That is, we note that a stack location, associated with a MIPS register, will hold a known literal value. When that register is used, we may be able to replace the register with the value it must hold (for example, replacing a register to register add, with an add immediate instruction).

Allocating memory addresses. As we learned in Chapter 11, local variables and parameters are allocated in the frame associated with a procedure or method. Thus we must map each JVM local variable into a unique frame offset, used to address the variable in load and store instructions. Since a frame contains some fixed-size control information followed by local data, a simple formula like offset = const + size * index suffices, where index is the JVM index assigned to a variable, size is the size (in bytes) of each stack value, const is the size (in bytes) of the fixed-size control area in the frame, and offset is the frame offset used in generated MIPS code.

Static fields of classes are assigned fixed, static addresses when a class is compiled. These addresses are used whenever a static field is referenced. Instance fields are accessed as an offset relative to the beginning of a class. Thus if we had a class `Complex` defined as

```
class Complex { float re; float im;}
```

the two fields `re` and `im`, each one word in size, would be given offsets of 0 (for `re`) and 4 (for `im`) within instances of the class. The bytecode `getfield Complex/im` fetches field `im` of the `Complex` object referenced by the top of stack. Translation is easy. We first lookup the offset of field `im` in class `Complex`, which is 4. The reference to the referenced object is in the register corresponding to top-of-stack, say `$t0`. We could add 4 to `$t0`, but since the MIPS has an indexed addressing mode that adds a constant to a register automatically (denoted `const($reg)`), we need generate no code. We simply generate `lw $t1,4($t0)`. which loads the field into register `$t1`, which now corresponds to the top of stack.

Allocating Arrays and Objects. In Java, and hence in the JVM, all objects are allocated on the heap. To translate a `new` or `newarray` bytecode, we'll need to call a heap-allocation subroutine, like `malloc` in C. We pass the size of the object required and receive a pointer to a newly allocated block of heap memory. For a `new` bytecode the size required is determined by the number and size of the fields in the object. In addition, a fixed-size header (to store the size and type of the object) is required. The overall memory size needed can be computed when the class definition of the object is compiled. In our earlier example of a `Complex` object, the size required would be 8 bytes plus 2 or 4 bytes of header information.

For `newarray` we determine the size required by multiplying the number of elements requested (in the register corresponding to the top of stack) by size requirement for individual array elements (stored in the symbol table entry for the requested class). Again, space for a fixed-size object header must be included.

Default initialization of fields within objects must also be performed by clearing or copying appropriate bit patterns (based on type declarations).

In languages like C and C++, objects and arrays can be allocated "inline" in the current frame if their size is known at compile-time. We can do a similar allocation within the current frame if we know that no reference to the allocated object "escapes." That is, if no reference to the allocated object or array is assigned to a field or returned as a function value, then the object or array is no

longer accessible after the current method (and its corresponding frame) are terminated.

As a further optimization, string objects (which are immutable) are often defined to be a string literal. Space for such a string may be allocated statically, and accessed via a fixed static address.

In the JVM array elements are accessed or updated using a single bytecode (e.g., `iaload` and `iastore` for integer arrays). In conventional architectures, like the MIPS, several instructions are needed to access an array, especially if array bounds checking is included. Details of array indexing are discussed in Chapter 14. Here we'll just show the kind of code that is needed to implement a JVM array load or store instruction.

An `iaload` instruction expects an array index at the top of the stack, and an array reference at the top-1 position. In our implementation, both of these values will be loaded or evaluated into MIPS registers. Let's call these registers `$index` and `$array`. We will need to generate code to check that `$index` is a legal index, and then to actually fetch the desired integer value from the array. Since arrays are just objects, the size of the array, and the array elements, are at fixed offsets relative to the start of the array object. Assume the array size is at offset `SIZE` and that elements are at offset `OFFSET`. Then the following MIPS code can be used to implement `iaload`, leaving the array value in register `$val`. (For simplicity and efficiency, we'll assume a null reference is represented by an invalid address that will force a memory fault.)

```
bltz   $index,badIndex      # Branch to badIndex if $index<0
lw     $temp,SIZE($array)   # Load size of array into $temp
slt    $temp,$index,$temp   # $temp = $index < size of array
beqz   $temp,badIndex       # Branch to badIndex if
                            # $index >= size of array
sll    $temp,$index,2       # multiply $index by 4 (size of
                            # an int) using a left shift
add    $temp,$temp,$array   # Compute $array + 4*$index
lw     $val,OFFSET($temp)   # Load word at
                            #   $array + 4*$index + OFFSET
```

The register `$temp` is a work register, used within the code sequence to hold intermediate values (see Section 15.3.1). The `iastore` instruction is very similar. A value at top-2 (in `$val`) is stored in the array referenced at top-1 (`$array`) using the index at the top of the stack (`$index`). We can use the following code to implement an `iastore` instruction

```
bltz   $index,badIndex      # Branch to badIndex if $index<0
lw     $temp,SIZE($array)   # Load size of array into $temp
slt    $temp,$index,$temp   # $temp = $index < size of array
beqz   $temp,badIndex       # Branch to badIndex if
                            # $index >= size of array
sll    $temp,$index,2       # multiply $index by 4 (size of
                            # an int) using a left shift
```

```
add    $temp,$temp,$array  # Compute $array + 4*$index
sw     $val,OFFSET($temp)  # Load $val into word at
                           #   $array + 4*$index + OFFSET
```

The MIPS code we've chosen for array indexing looks rather complex and expensive, especially since arrays are a very commonly used data structure. Part of this complexity is due to the fact that we've included array bounds checking, as required in Java. In C and C++, array bounds are rarely checked at run-time, allowing for fast (and less secure!) code.

In many cases it is possible to optimize or entirely eliminate array bounds checks. On architectures that support unsigned arithmetic, the check for an index too large and the check for an index too small (less than zero) can be combined! The trick is to do an unsigned compare between the array index and the array size. A negative index will be equivalent to a *very* large unsigned value (since its leftmost bit will be one), making it greater than the array size.

In a for loop it is often possible to determine that a loop index is bounded by known lower and upper bounds. With this information, arrays indexed by the loop index may be known to be "in range," eliminating any need for explicit checking. Similarly, once an array bound is checked, subsequent checks of the same bound are unnecessary until the index is changed. Thus in `a[i] = 100-a[i]`, `a[i]` needs to be checked only once.

If array bounds checks are optimized away, or simply suppressed, array indexing is much more efficient—typically three (or fewer) instructions (a shift (or multiply), an add, and a load or store). In the case that the array index is a compile-time constant (e.g., `a[100]`), we can reduce this to a single instruction by doing the computation of size*index+offset at compile-time and using it directly in a load or store instruction.

Method Calls. The JVM makes method calls very simple. Many of the details of a call are hidden. In implementing an `invokestatic` or `invokevirtual` bytecode, we must make such hidden details explicit.

Let's look at `invokestatic` first. In the bytecode version of the call, parameters are pushed onto the stack, and a static method, qualified by its class and type (to support overloading) is accessed. In our MIPS translation, the parameters will be in registers, which is fine since that's how most current architectures pass scalar (word-sized) parameters.

We'll need to guarantee that parameters are placed in the right registers. On the MIPS, the first four scalar parameters are passed in registers `$a0` to `$a3`; remaining parameters and non-scalar parameters are pushed onto the run-time stack. In our translation, we can generate explicit register copy instructions to move argument values (already in registers) to the correct registers. If we wish to avoid these extra copy instructions, we can compute the parameters directly into the correct registers. This is called register targeting. Essentially, when the parameter is computed, we mark the target register into which the parameter will be computed to be the appropriate argument register. Graph coloring, as discussed in Section 15.3.2, makes targeting fairly easy to do. For parameters that are to be

passed on the run-time stack, we simply extend the stack (by adjusting the top-of-stack register, `$sp`) and then store excess parameters into the locations just added.

To transfer control to the subprogram, we issue a `jal` (jump and link) instruction. This instruction transfers control to the start of the method to be called (using an address recorded when the subprogram was translated) *and* stores a return address in the return address register, `$ra`.

Additional details must be handled to complete our translation of an `invokestatic` instruction. Since at the point of call variable and expression values may be held in registers, these registers must be saved prior to execution of the method. All registers that hold values that may be destroyed during the call (by the instructions in the method body) are saved on the stack, and restored after the method completes execution. Registers may be saved by the caller (these are caller-save registers) or by the method to be called (these are callee-save registers). It doesn't really matter if the caller or callee does the saving (often both save selected registers), but any register holding a program value needed after the call must be protected.

If a non-local or global variable is held in a register, it must be saved, prior to a call, in its assigned memory location. This guarantees that the subprogram will see the correct value during the call. Upon return, registers holding non-local or global variables must be reloaded since the subprogram may have updated their values.

As an example, consider this function call `a = f(i,2);` `a` is a static field, `f` is a static method and `i` is a local variable held in register `$t0`, a caller-save register. Assume that a storage temporary, assigned frame offset 32, is created to hold the value of `$t0` across the call. The following MIPS code is produced

```
move   $a0,$t0         # Copy $t0 to parm register 1
li     $a1,2           # Load 2 into parm register 2
sw     $t0,32($fp)     # Store $t0 across call
jal    f               # Call function f
                       # Function value is in $v0
lw     $t0,32($fp)     # Restore $t0
sw     $v0,a           # Store function value in a
```

When a method is called, space for its frame must be pushed, and the frame and stack pointers must be properly updated. This is normally done in the prologue of the called method, just before its body is executed. Similarly, after a method is finished, its frame must be popped, and frame and stack pointers properly reset. This is done in the method's epilogue, just before branching back to the caller's return address. The exact code sequences needed vary according to hardware, and operating system conventions, but the following MIPS instructions can be used to push, and later pop, a method's frame. (The size of the frame, `frameSz`, is determined when the method is compiled and all its local declarations are processed; on the MIPS the run-time stack grows downward).

```
subi  $sp,$sp,frameSz# Push frame on stack
```

```
 sw    $ra,0($sp)      # Save return address in frame
 sw    $fp,4($sp)      # Save old frame pointer in frame
 move  $fp,$sp         # Set $fp to access new frame
# Save callee-saved registers (if any) here
# Body of method is here
# Restore callee-saved registers (if any) here
 lw    $ra,0($fp)      # Reload return address register
 lw    $fp,4($fp)      # Reload old frame pointer
 addi  $sp,$sp,frameSz# Pop frame from stack
 jr    $ra             # Jump to return address
```

To translate an `invokevirtual` instruction, we must implement a dynamic dispatching mechanism. When a method `M` of class `C` is called using `invokevirtual`, we will be given, as the first parameter, an object of class `C` or any subclass of `C`. If `M` is redefined in the subclass (with exactly the same type as `M` in class `C`), the redefined version of `M` must be called. How do we know which version of `M` to execute?

The first parameter, compiled into a register, is a pointer to an object of class `C` or a subclass of `C`. To support garbage collection and heap management, each heap object has a type code as part of its header. This type code can be used to index into a dispatch table that contains the addresses of all methods the object contains. If we assign to each method a unique offset, we can use `M`'s offset, in the object's dispatch table, to choose the correct method to execute.

Fortunately, it is often the case that class `C` has no subclasses that redefine `M`. (e.g., if `C` or `M` is `private` or `final`). If dynamic loading of `C` is not possible, we can select `M`'s implementation at compile-time, and generate code to call it directly, without any table lookup overhead.

Example. As an example of the overall bytecode translation process, let us consider the following simple method, `stringSum`. This method sums integers from one to its parameter, `limit`, and returns a string representation of the sum:

```
public static String stringSum(int limit){
   int sum = 0;
   for (int i = 1; i <= limit; i++)
      sum += i;
   return Integer.toString(sum);
}
```

The following bytecodes implement `stringSum`:

```
      iconst_0        ; Push 0
      istore_1        ; Store into variable #1 (sum)
      iconst_1        ; Push 1
      istore_2        ; Store into variable #2 (i)
```

```
        goto L2         ; Go to end of loop test
  L1:   iload_1         ; Push var #1 (sum) onto stack
        iload_2         ; Push var #2 (i) onto stack
        iadd            ; Add sum + i
        istore_1        ; Store sum + i into var #1 (sum)
        iinc 2 1        ; Increment var #2 (i) by 1
   L2:  iload_2         ; Push var #2 (i)
        iload_0         ; Push var #0 (limit)
        if_icmple L1    ; Goto L1 if i <= limit
        iload_1         ; Push var #1 (sum) onto stack
                        ; Call toString
        invokestatic java/lang/Integer/toString(int)
        areturn         ; Return String reference to caller
```

In analyzing `stringSum`, we see references to three local variables (including the parameter). Adding in two words of control information, we conclude that a frame size of 5 words (20 bytes) is required. `limit` will be placed at offset 8, `sum` at offset 12, and `i` at offset 16.

In the code we generate below, we will follow the MIPS convention that one word function values, including object references, will be returned in register `$v0`. We will also exploit the fact that register `$0` always contains a zero value. The code we generate will begin with a method prologue (to push `stringSum`'s frame), then a line-by-line translation of its bytecodes, followed by an epilogue to pop `stringSum`'s frame and return to its caller.

```
        subi  $sp,$sp,20     # Push frame on stack
        sw    $ra,0($sp)     # Save return address
        sw    $fp,4($sp)     # Save old frame pointer
        move  $fp,$sp        # Set $fp to access new frame
        sw    $a0,8($fp)     # Store limit in frame
        sw    $0,12($fp)     # Store 0 ($0) into sum
        li    $t0,1          # Load 1 into $t0
        sw    $t0,16($fp)    # Store 1 into i
        j      L2            # Go to end of loop test
  L1:   lw    $t1,12($fp)    # Load sum into $t1
        lw    $t2,16($fp)    # Load i into $t2
        add   $t3,$t1,$t2    # Add sum + i into $t3
        sw    $t3,12($fp)    # Store sum + i into sum
        lw    $t4,16($fp)    # Load i into $t2
        addi  $t4,$t4,1      # Increment $t4 by 1
        sw    $t4,16($fp)    # Store $t4 into i
   L2:  lw    $t5,16($fp)    # Load i into $t5
```

```
lw    $t6,8($fp)      # Load limit into $t6
sle   $t7,$t5,$t6     # set $t7 = i <= limit
bnez  $t7,L1          # Goto L1 if i <= limit
lw    $t8,12($fp)     # Load sum into $t8
move  $a0,$t8         # Copy $t8 to parm register
jal   String_toString_int_ # Call toString
                      # String ref now is in $v0
lw    $ra,0($fp)      # Reload return address
lw    $fp,4($fp)      # Reload old frame pointer
addi  $sp,$sp,20      # Pop frame from stack
jr    $ra             # Jump to return address
```

15.2 Translating Expression Trees

So far, we have concentrated on generating code from ASTs. We now focus on generating code from expression trees. In an expression tree interior nodes represent operators and leaves represent variables and constants. Many ASTs use this format for expressions, so much of this discussion applies to ASTs too.

An expression tree may be traversed and translated in many different orders. Normally, a depth-first, left-to-right traversal is used when translating expressions. A depth-first, left-to-right traversal always produces a *valid* translation. However alternative traversals may lead to better code (if exceptions, which must be tested in source order, are not a concern).

Consider the expression `(a-b) + ((c+d)+(e*f))`. The normal depth-first traversal first translates `(a-b)`, leaving its result in a register. Then `(c+d)+(e*f)` is translated, requiring three registers (one to hold the first subexpression, and two more to evaluate the other subexpression). Thus a total of four registers is used. However, if the right subexpression, `(c+d)+(e*f)`, is evaluated first, only three registers are needed, because once this subexpression is computed its value can be held in one register, using the other two registers to compute `(a-b)`.

We now consider an algorithm that determines the minimum number of registers needed to evaluate any expression or subexpression. We ignore for the moment any special properties of operators, like associativity. The algorithm labels each node in a tree with the minimum number of registers needed to evaluate the subexpression rooted by that node. This labeling is called Sethi-Ullman numbering [Sethi Ullman 70]. Once the minimum number of registers needed for each expression and subexpression is known, we traverse the tree in a manner that generates *optimal* code (that is, code that minimizes register use and hence register spilling).

As we did in previous sections, we assume a MIPS-like machine model that requires that all operands be register-resident. The algorithm works in a bottom-up direction, first labeling leaves of the tree. All leaves are labeled with 1, since one

register is needed to hold a variable or constant. For interior nodes, which are assumed to be binary operators, the register requirements of the operands are considered. If both operands require *r* registers, the operator requires *r+1* registers because an operand, once computed, will be held in a register. If the two operands require a different number of registers, the whole expression requires the same number of registers as the more complex of the two operands. (Evaluate the more complex operand first and save it in a register. The simpler operand needs fewer registers, and hence the registers used for the previous, more complex operand can be reused.) This analysis leads to the algorithm of Figure 15.1.

```
   registerNeeds( T )
1.    if  T.kind = Identifier or T.kind = IntegerLiteral
2.          then  T.regCount ← 1
3.          else  registerNeeds(T.leftChild)
4.                registerNeeds(T.rightChild)
5.                if  T.leftChild.regCount = T.rightChild.regCount
6.                      then  T.regCount ← T.rightChild.regCount+1
7.                      else  T.regCount ← max(T.leftChild.regCount,
                                                 T.rightChild.regCount)
```

Figure 15.1 An Algorithm to Label Expression Trees with Register Needs

As an example of this algorithm, registerNeeds would label the expression tree for `(a-b) + ((c+d)+(e*f))` as shown in Figure 15.2 (regCount for each node is shown at its bottom).

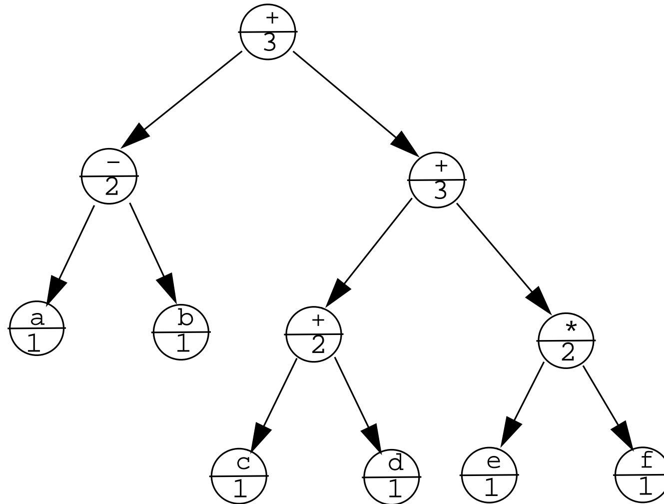


Figure 15.2 Expression Tree for `(a-b) + ((c+d)+(e*f))` with Register Needs.

We can use the regCount labeling to drive a simple, but optimal, code generator, treeCg, defined in Figure 15.3. treeCG takes a labeled expression tree and a

list of registers it may use. It generates code to evaluate the tree, leaving the result of the expression in the first register on the list. If treeCG is given too few registers, it will spill registers, as necessary, into storage temporaries. (We use standard list manipulation functions, like head and tail, without defining them.).

```
    treeCG( T, regList )
 1.   r1 ← head(regList)
 2.   r2 ← head(tail(regList))
 3.   if T.kind = Identifier    /* Load a variable */
 4.       then generate(lw, r1, T.IdentifierName)
 5.   elsif T.kind = IntegerLiteral    /* Load a literal */
 6.       then generate(li, r1, T.IntegerValue)
 7.       else /* T.kind must be a binary operator */
 8.            left ← T.leftChild
 9.            right ← T.rightChild
10.            if left.regCount ≥ length(regList) and
                  right.regCount ≥ length(regList)
11.                then /* Must spill a register into memory */
12.                     treeCG(left, regList)
13.                     temp ← getTemp() /* Get memory location */
14.                     generate(sw, r1, temp)
15.                     treeCG(right, regList)
16.                     generate(lw, r2, temp)
17.                     freeTemp(temp) /* Free memory location */
18.                     generate(T.operation, r1, r2, r1)
19.                else /* There are enough registers; no spilling is needed */
20.                     if left.regCount ≥ right.regCount
21.                         then treeCG(left, regList)
22.                              treeCG(right, tail(regList))
23.                              generate(T.operation, r1, r1, r2)
24.                         else treeCG(right, regList)
25.                              treeCG(left, tail(regList))
26.                              generate(T.operation, r1, r2, r1)
```

Figure 15.3 An Algorithm to Generate Optimal Code from Expression Trees

As an example, if we call treeCG with the labeled tree of Figure 15.2 and three registers, `($10, $11` and `$12)`, we obtain the following code sequence:

```
lw  $10,c        # Load c into register 10
lw  $11,d        # Load d into register 11
add $10,$10,$11  # Compute c + d into register 10
lw  $11,e        # Load e into register 11
lw  $12,f        # Load f into register 12
mul $11,$11,$12  # Compute e * f into register 11
add $10,$10,$11  # Compute (c + d) + (e * f) into reg 10
```

```
lw  $11,a         # Load a into register 11
lw  $12,b         # Load b into register 12
sub $11,$11,$12   # Compute a - b into register 11
add $10,$11,$10   # Compute (a-b)+((c+d)+(e*f)) into reg 10
```

treeCG illustrates nicely the principle of register targeting. Code is generated in such a way that the final result appears in the targeted register, without any unnecessary moves.

Because our simple machine model requires that all operands be loaded into registers, commutative operators (for which `exp1 op exp2` is identical to `exp2 op exp1`) can't be exploited to improve code quality. However, most computer architectures aren't entirely symmetric. Thus in the MIPS R3000 architecture, some operations (like add and subtract) allow the right operand to be immediate. Immediate operands are small literal values included directly into an instruction; they need not be explicitly loaded into registers (see Exercise 8). For commutative operators, a small literal used as a left operand can be treated as if it were a right operand.

Some operations, like addition and multiplication are associative. Operands of an associative operator may be processed in any order. Thus, mathematically, `a+b+c` and `c+b+a` are identical. Regrouping operands of an associative operators can reduce the number of registers needed to evaluate an expression (see Exercise 9). For example, using registerNeeds we can establish that `(a+b)+(c+d)` requires three registers whereas `a+b+c+d` requires only two registers. Unfortunately, because of overflow and rounding issues, computer arithmetic is often *not* associative. Thus if `a` and `b` equal 1, `c` equals `maxint`, and `d` equals -10, `(a+b)+(c+d)` will evaluate correctly, whereas `a+b+c+d` may overflow. Many compilers reorder operands only when it is absolutely safe to do so.

15.3 Register Allocation and Temporary Management

An essential component of any code generator is its register allocator. Machine registers must be assigned to program variables and expressions. Since registers are limited in number, they must be reclaimed (reused) throughout a program.

A register allocator may be a simple "on the fly" algorithm that assigns and reclaims registers as code is generated. We'll consider on the fly techniques first. More thorough register allocators, that consider the register needs of an entire subprogram or program, will be considered next.

15.3.1 On the Fly Register Allocation

Most computers have distinct integer (general purpose) and floating register sets. In organizing our register allocator, we'll divide each register set into a number of classes:

- Allocatable registers
- Reserved registers

- Work registers

Allocatable registers are explicitly allocated and freed by compile-time calls to register management routines. While allocated, registers are protected from use by any but the "owner" of the register. Thus it is possible to guarantee that a register containing a data value will not be incorrectly changed by another use of the same register.

Requests for allocatable registers are usually *generic*; that is, requests are for any member of a register class, not for a particular register in that class. Usually any member of a register class will do. Further, generic requests eliminate the problem that arises if a particular requested register is already in use but many other registers in the same class are available.

A register, once allocated, must be freed when its assignment to a particular temporary is completed. A register is usually freed in response to an explicit directive issued by a semantic routine. This directive also allows us to mark the last use of a register as dead. This is valuable information because better code may be possible if the contents of a register need not be preserved.

Reserved and work registers, on the other hand, are never explicitly allocated or freed. Reserved registers are assigned a fixed function throughout a program. Examples include display registers, stacktop registers, argument and return value registers, as well as return address registers. Since the function of reserved registers is set by the hardware or operating system, and they are in use for all of a program or procedure, it is unwise to use such registers for other than their designated purpose

Work registers may be used at any time by any code generation routine. Work registers may safely be used only in local code sequences, over which the code generator has complete control. That is, if we were generating code to do an indexing operation on an array (say, `a[i+j]`), it would be wrong to use a work register to hold the address of the array because computation of `i+j` might change the work register. An allocatable register would, of course, be protected.

Work registers are useful in several circumstances:

- Sometimes we need a register for a very brief time. (For example, in compiling `a=b` we load `b` into a register and then immediately store the register into `a`.) Using a work register saves the overhead of allocating and then immediately freeing a register.
- Many instructions require that their operands be in registers. What happens if *no* registers are free? Work registers are always free. If necessary, we can load values from memory into work registers, execute an instruction or two, then save needed values back into memory.
- We can pretend we have more registers than we really do. Such registers, sometimes called virtual registers or pseudo-registers, can be simulated by allocating them in memory and placing their values into work registers when they are used in instructions.

In general, reserved registers are identified in advance by hardware and operating systems conventions. Sometimes work registers are also established in

advance—if they aren't, choose three or four for this purpose. Remaining registers can be made allocatable. They will hold temporary values, and may also be used to hold frequently accessed variables and constants.

getReg and freeReg. To allocate and free registers, we'll create two subroutines, getReg and freeReg. getReg will allocate a single allocatable register and return its index. (If we have both integer and float registers, we will create getReg and getFloatReg.) A register allocated by a call to getReg is exclusively allocated to the caller until it is returned.

What happens if no more registers are available for allocation? In simple compilers we can simply terminate compilation with a message that the program requires more registers than are available. Modern computers routinely have at least 10–20 allocatable registers, so unless registers are used aggressively to hold program variables and constants, the chances of a "real life" program exhausting all allocatable registers is almost nil.

A more robust register allocator should not simply terminate when registers are exhausted. It can instead return pseudo-registers allocated in memory (in the frame of the procedure currently being translated). Pseudo-registers are encoded as integers greater than the indices of the real hardware registers. An array `regAddr`[] maps pseudo-registers into their memory addresses. Pseudo-registers are used exactly like real registers. The only difference is that when instructions using pseudo-registers are generated, they use work registers to move values back and forth from memory.

In some languages we may need to allocate temporaries in memory rather than registers. This may be because the temporary is too large to fit in a register (e.g., a struct returned by a function call) or because we need to be able to create a pointer to the temporary (most computers don't allow indirect references to registers). If we need storage based temporaries, we can create getTemp and freeTemp functions that essentially parallel getReg and freeReg. Temporaries allocated for a procedure are placed in the procedure's frame (essentially they are anonymous local declarations). Temporaries used by the main program may be allocated statically.

In some languages we may need to allocate temporaries whose size is not known at compile time. For example, if we use the + operator to concatenate C-style strings, then the size of `str1 + str2` will not in general be known until run-time. The temporary used to hold the result of such an expression can't be allocated statically or in a frame. Instead, we'd either push it onto the run-time stack or allocate space for it in the heap. Since pushing the stack is much more efficient (two or three instructions rather than hundreds), it is preferred. Note that when we return from the current procedure, the stack will be popped and the temporary *automatically* freed.

15.3.2 Register Allocation Using Graph Coloring

Using registers effectively is essential in generating efficient code for modern computers. We have already studied how to allocate registers in trees and "on the fly" as code is generated. In this section we address a greater challenge—how to allocate registers effectively throughout an entire procedure, function or main program. Since individual procedures and functions are the basic units of compilation in modern compilers, we have raised our sights from individual statements or expressions to entire procedure bodies.

Register allocation at the level of an entire subprogram is called global allocation, in contrast to allocation at the level of a single expression or basic block, which is termed local allocation. At the global level, a register allocator usually has many values that might profitably reside in registers (local and global variables, constants, temporaries containing available expressions, parameters and return values, etc.). Each value that might profitably reside in a register is called a register candidate; typically there are many more register candidates than there are registers.

Global register allocators do not usually allocate a register to a single value throughout the body of a subprogram. Rather, when possible, values that do not interfere with one another are assigned to the same register. Thus if variable `a` is used only at the top of a subprogram, and variable `b` is used only at the bottom on the subprogram, `a` and `b` may share the same register.

To enhance sharing, register candidates are divided into live ranges. A live range is the span of instructions in which a given value may be accessed, from its initial creation to its last use. For variables, a live range runs from its point of initialization or assignment to its last uses. For expressions and constants, a live range spans from their first to final use. Thus in Figure 15.4 variable `a` is broken into two separate and independent live ranges. Each is treated as a separate register candidate.

```
main() {
   a = f(x);          // Start of first live range
   print(a);          // End of first live range
   ....
   a = g(y)           // Start of second live range
   print(a);          // End of second live range
}
```

Figure 15.4 Example of Live Ranges

A live range can be readily computed using the SSA form described in Chapter 16, since each use of a variable is tied to *unique* assignment. Alternatively, one can avoid live range computation and simply treat each variable, parameter or constant as a distinct register candidate.

The Interference Graph. One of the central problems in global register allocation is deciding which live ranges may share the same register and which may not. A live range *l* is said to interfere with another live range *m* if *l*'s definition point (or beginning) is part of *m*'s range. This make sense—*l* and *m* cannot share the same register if at the point *l* is first computed or loaded *m* is also in use.

To represent all the interferences in a subprogram (there normally are many), an interference graph is built. Nodes of the graph are the live ranges of the subprogram. An arc exists between live ranges *l* and *m* if *l* interferes with *m* or *m* interferes with *l* (the arc is undirected). Consider the simple procedure shown in Figure 15.5. It has four register candidates, `a`, `b`, `c`, and `i`. `a`, `b` and `i` all interfere with one another; `c` interferes only with `a`. This interference information is concisely shown in the interference graph of Figure 15.6.

```
proc() {
   a = 100;
   b = 0;
   for (i=0;i<10;i++)
      b = b + i * i;
   print(a, b);
   c = 100;
   print(a*c);
}
```

Figure 15.5 A Simple Procedure with Candidates for Global Register Allocation.

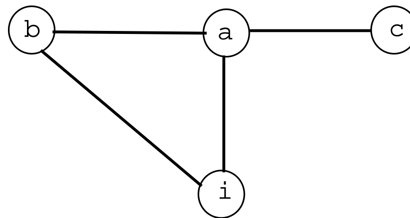


Figure 15.6 Interference Graph for procedure of Figure 15.5

With an interference graph, the problem of allocating registers is neatly reduced to a well-known problem—that of coloring the nodes of a graph. In the graph coloring problem the goal is to determine whether *n* colors suffice to color a graph given the rule that no two nodes linked by an arc may share the same color. This models exactly our problem of register allocation, where *n* is the number of registers we have available and each color represents a different register.

The problem of determining whether a graph is "*n*-colorable" is NP-complete [Garey Johnson 79]. This means the best known algorithms that solve the problem have a time bound that is exponential in the size of the graph. As a result, register

allocators based on graph coloring normally use heuristics to solve the coloring problem.

We'll first consider an approach to register allocation using coloring devised by Chaitin [CAC 81, Cha 82]. Initially, the algorithm assumes that all register candidates can be allocated registers. This is often an impossible goal, so the interference graph is tested to see if it is n-colorable, where n is the number of registers available for allocation. If the interference graph is n-colorable, a register allocation is produced from the colors assigned to the interference graph.

If the graph is not n-colorable, it is simplified. A node (corresponding to a live range) is selected and spilled. That is, the live range is denied a register. Rather, whenever it is assigned to or used, it is loaded from or stored into memory using work registers, like the pseudo-registers of the previous section.

Since the live range that was spilled is no longer a register candidate it is removed from the interference graph. The graph is simpler and may now be n-colorable. If it is, our register allocation is successful—all remaining candidates can be allocated registers. If the graph still isn't n-colorable, we select and spill another candidate, further simplifying the graph. This process continues until an n-colorable graph is obtained.

Two questions arise. How do we decide if a graph is n-colorable? (Recall this is a very hard problem). If a graph isn't n-colorable, how do we choose the "right" register candidate to spill?

In testing for n-colorability, Chaitin made the following simple but powerful observation. If a node in the interference graph has fewer than n neighbors, that node can always be colored (just choose any color not assigned to any of its neighbors). Such nodes (termed unconstrained) are removed from the interference graph. This simplifies the graph, often making further nodes unconstrained. Sometimes all nodes are removed, demonstrating that the graph *is* n-colorable.

When only nodes with n or more neighbors remain, a node is spilled to allow the graph to be simplified. Chaitin suggests that in choosing a node to spill, two criteria be considered. First, the cost of spilling a node should be considered. That is, we compute the extra loads and stores that will have to executed should a live range be spilled, weighted by the loop nesting level. Each level of loop nesting is arbitrarily assumed to add a factor of 10 to costs. Thus a live range in a single loop has its loads and stores multiplied by 10, a doubly nested loop multiplies loads and stores by 100, etc.

The second criterion Chaitin used is the number of neighbors a node has. The greater the number of neighbors a node has, the greater the number of interferences spilling the node removes. Chaitin suggests that the node with the smallest value of *cost/neighbors* is the best node to spill. That is, the "ideal" node to spill is one that has a low spill cost and many neighbors, yielding a very small *cost/neighbors* value.

Chaitin's algorithm is shown in Figure 15.7. As an example, consider the interference graph of Figure 15.6. Assume only two registers are available for allocation. Since `c` has only one neighbor, it is immediately removed from the graph and pushed on a stack for later register allocation. `a`, `b` and `i` all have two neigh-

```
    GCRegAlloc( proc, regCount )
 1.   ig ← buildInterferenceGraph(proc)
 2.   stack ← ϕ
 3.   while ig ≠ ϕ
 4.       do if ∃ d ∈ ig for which neighborCount(d) < regCount
 5.              then ig ← ig - d
 6.                   push(d,stack)
 7.              else Choose d such that
                          cost(d)/ neighborCount(d) is minimized
 8.                   ig ← ig - d
 9.                   Generate code to spill d's live range
10.   while stack ≠ ϕ
11.       do  d ← pop(stack)
12.           reg(d) ← any register not assigned to neighbors(d)
```

Figure 15.7 Chaitin's graph coloring register allocator.

bors. One will have to be spilled. `a` has a very low cost (3) because it is referenced only 3 times, all outside of the loop. `b` and `i` are used inside the loop and have much higher costs. Since all three nodes have the same number of neighbors, `a` is correctly chosen as the proper node to spill. After a is removed, `i` and `b` become unconstrained. When registers are assigned, `i` and `b` get different registers and `c` can be assigned either register. `a` gets no register. Rather, whenever it is used, it is loaded from or stored into a memory location, just like an ordinary variable.

Improvements to Graph Coloring Register Allocators. Briggs et al. [BCKT 89] suggest a number of useful improvements to Chaitin's approach. They point out that nodes with the smallest number of neighbors ought to be removed first from the interference graph. This is because nodes with few neighbors are the easiest to color and hence they ought to be processed last during the phase in which stacked nodes are popped and colored.

Another improvement follows from the observation that as nodes are removed to simplify the interference graph, they need not be spilled immediately. Rather, removed nodes should be stacked just like unconstrained nodes. When nodes are colored, constrained nodes *may* be colorable (because they happen to have neighbors that share the same color or happen to have neighbors that are also marked to be spilled). Constrained nodes that can't be colored are spilled only when we are sure they are uncolorable.

Register allocators need to handle two other problems. Assignments between register values are common. We would like to reduce register moves by assigning the source and target values in the assignment to the same register, making the assignment a trivial one. Moreover, architectural and operating system constraints sometimes force values to be assigned to specific registers. We'd like our allocator to try to choose register assignments that anticipate and adhere to predetermined register conventions.

To see how coloring allocators can handle register moves and preallocated registers, consider the following simple subprogram.

```
int doubleSum(int initVal, int limit({
    int sum = initVal;
    for (int i=1; i <= limit; i++)
       sum += i;
    return 2*sum; }
```

When this subprogram is translated, many short-lived temporary locations are created. Moreover, rules involving register allocation for parameters and return values are enforced. Prior to register allocation, `doubleSum` looks like the following

```
doubleSum(){
    initVal = $a0;     // First parm passed in $a0
    limit = $a1;       // Second parm passed in $a1
    sum = initVal;
    i = 1;
    temp1 = i <= limit;
    while (temp1) {
       temp2 = sum + i;
       sum = temp2;
       temp3 = i + 1;
       i = temp3;
       temp1 = i <= limit; }
    temp4 = 2 * sum;
    $v0 = temp4; // Return value register is $v0
 }
```

The explicit use of register names in `doubleSum` represents nodes that must be allocated a particular register; these nodes are said to be precolored. If `a` and `b` are both allocated to registers, and we have the assignment `a = b`, an explicit register copy can be avoided if `a` and `b` are allocated to the same register. Values `a` and `b` will be automatically assigned to the same register if we coalesce their live ranges. That is, if we combine the nodes for `a` and `b` in the interference graph, then `a` and `b` *must* receive the same register.

When is coalescing `a` and `b` safe? At a minimum, they must not interfere. If they do interfere, then they are live at the same time and will need distinct registers. Even if `a` and `b` do not interfere, coalescing them may be problematic. The difficulty is that combining the live ranges of `a` and `b` will in general create a larger live range that is harder to color. We certainly don't want to spill a combined range when the individual ranges might have been individually colored.

To avoid problems in coloring coalesced interference graph nodes we can adopt a conservative approach. We will say a node in an interference graph has

significant degree if it has *n* or more neighbors (where *n* is the number of colors available). A node of significant degree may have to be spilled. A node that is insignificant (i.e., not significant) is always colorable. We can conservatively coalesce nodes `a` and `b` if the combined interference graph node has fewer than *n* significant neighbors. Why? Well, insignificant neighbors are always removed because they are trivially colorable. If the combined node has fewer than *n* significant neighbors, then, after insignificant neighbors are removed, the combined node will have fewer than *n* neighbors, so it too will be trivially colorable.

In our above `doubleSum` example, we have three values that must be register-resident (the two parameter values at the start, and the return value at the end). We have eight local variables and temporaries (`initVal`, `limit`, `i`, `sum`, `temp1`, `temp2`, `temp3` and `temp4`). We'll aim for a 4-coloring (a register allocation that uses 4 registers). Temporary `temp1` interferes with `i`, `limit` and `sum`, so we know that we can't use fewer than 4 registers without spilling.

We can coalesce `temp4` and `$v0`, guaranteeing that `2*sum` is computed into the return value register. We can coalesce `$a0` and `initVal`, allowing `initVal` to be accessed directly from `$a0` throughout the subprogram. Even more interestingly, we can then coalesce `initVal` and `sum`, allowing `sum` to use `$a0` too. Temporary `temp2` can also be coalesced with `sum`, allowing it too to use `$a0`. `limit` can be coalesced with `$a1`, allowing it to use `$a1` throughout the subprogram.

Temporary `temp3` can be coalesced with `i` since the combined node has fewer than 4 neighbors. Since neither `temp1` nor the combined `i` and `temp3` interfere with `$v0`, either of these can be assigned `$v0` to use. The other is assigned an unused register, for example `$t0`. The resulting register allocation, with register names replacing variables and temporaries, is shown below. Note that all register to register copies have been removed, and that only one register, beyond the preassigned ones, is used.

```
doubleSum(){
     $v0 = 1;
     $t0 = $v0 <= $a1;
     while ($t0) {
        $a0 = $a0 + $v0;
        $v0 = $v0 + 1;
        $t0 = $v0 <= $a1;
     }
     $v0 = 2 * $a0;
  }
```

It is sometimes possible to coalesce interference graph nodes that have more than *n* significant neighbors. This is done by iterating between interference graph simplification and node coalescing [GA 96]. The resulting algorithm is very effective and is one of the simplest and most effective register allocators in current use.

15.3.3 Priority Based Register Allocation

Hennessey and Chow [90] and Larus and Hilfinger [86] suggest interesting alternatives to Chaitin's graph coloring approach. After unconstrained nodes (which are trivially colorable) are removed from the interference graph, a priority is computed for each remaining node. This priority is similar to Chaitin's cost estimate, except that it normalizes the cost using the size of the live range. That is, if two live ranges have the same cost, but one is smaller (in terms of the number of instructions it spans), the smaller live range ought to be given preference over the larger one. This makes sense—the smaller the live range is, the shorter is the span of instructions in which it "ties up" a register. The priority function recommended is *cost/size(live range)*. The greater the priority of a live range, the more likely it is to receive a register.

Another important difference is that when a node can't be colored (because its neighbors have been allocated all the available colors), the node is split rather than being spilled. That is, if possible, the live range is divided into two smaller live ranges. Loads and stores are placed at the boundary of the split ranges, but each split range may be allocated a (possibly different) register. Because split ranges usually have fewer interferences than the original range, split ranges are often colorable when the original range is not.

There are many ways a live range may be split into smaller ranges. The following simple heuristic is often used:

1. Remove the first instruction of the live range (usually a load or computation), putting it into a new live range, *NR*.
2. Move successors to instructions in *NR* from the original live range to *NR* as long as *NR* remains colorable.

The idea is to break off at least one instruction, and then add instructions as long as the split range appears colorable. Instructions not split off remain in what's left of the original live range, which may be split again. Single definitions or uses that can't be colored are spilled.

A priority-based register allocator, PriorityRegAlloc, is shown in Figure 15.8. Reconsider the interference graph of Figure 15.6, assuming two registers, `$r1` and `$r2`. Variable `c` is placed in unconstrained; it is trivial to color and will be handled after all other variables have been allocated registers. `a`, `b` and `i` are all placed in constrained. `i` has the highest priority for register allocation, since assigning it a register saves 51 loads and stores, and it spans only two statements. Assume it is assigned register `$r1`. Variable `b` has the next highest priority (22 loads and stores saved). It is given `$r2`. Variable `a` is the last candidate in constrained, but it can't be colored. We split it into two smaller live ranges, `a1` and `a2`. `a1` is the single assignment at the top of the procedure. Range `a2` spans the two print statements. `a1` is effectively spilled since its range is a single instruction. `a2` interferes with `b` but not `i`. Hence it receives `$r1`. Finally, `c` receives `$r2`.

```
   PriorityRegAlloc( proc, regCount )
1.    ig ← buildInterferenceGraph(proc)
2.    unconstrained ← { n ∈ nodes(ig) | neighborCount(n) < regCount }
3.    constrained ← { n ∈ nodes(ig) | neighborCount(n) ≥ regCount }
4.    while constrained ≠ ϕ
5.          do for c ∈ constrained such that not colorable(c) and canSplit(c)
6.                  do c1, c2 ← split(c)
7.                     constrained ← constrained - c
8.                     if neighborCount(c1) < regCount
9.                        then unconstrained ← unconstrained + c1
10.                       else constrained ← constrained + c1
11.                    if neighborCount(c2) < regCount
12.                       then unconstrained ← unconstrained + c2
13.                       else constrained ← constrained + c2
                       for d ∈ neighbors(c) such that d ∈ unconstrained
                               and neighborCount(d) ≥ regCount
14.                         do unconstrained ← unconstrained - d
15.                            constrained ← constrained + d
            /* At this point all nodes in constrained are colorable
                    or can't be split */
16.         Select p ∈ constrained such that priority(p) is maximized
17.         if colorable(p)
18.            then color p
               else spill p
19.   color all nodes in unconstrained
```

Figure 15.8 A priority-based graph coloring register allocator.

15.3.4 Interprocedural Register Allocation

The global register allocators we have considered are limited by the fact that they consider only one subprogram at a time. Interprocedural interactions are ignored. Thus when a subprogram is called, either the caller or callee must save and restore any registers that both might use. When registers are used aggressively to hold a large number of variables, constants and expressions, saving and restoring common registers can make calls costly. Similarly, if a number of subprograms access the same global variable, each must load and later save the value when it is used.

Interprocedural register allocation improves overall register allocation by identifying and removing register conflicts across calls. Wall [86] considers interprocedural register allocation for architectures with a large number of registers. His goal is to assign registers so that caller and callee never use the same register. This guarantees that no saves or restores are needed during a call, making the call very inexpensive.

First, a priority estimate, similar to that of the previous section, is computed for each local variable or constant that might be kept in a register. These priorities are weighted by estimates of the execution frequency of each procedure. That is,

variables used by frequently executed subroutines have a much higher priority than those of infrequently executed subroutines. This is reasonable—we want to use registers most effectively in those subprograms that are executed most often. If procedure `a` calls `b`, the register allocator places the locals of `a` and `b` in different registers. Otherwise, a local of `a` and a local of `b` can share a common register. Groups of locals, one from each of a set of subprograms that can never be simultaneously active, are grouped together. The priority of a group is the sum of the priorities of all its member locals.

Registers are then "auctioned." The group with the highest overall priority gets the first register. The next highest priority group gets the next register and so forth. Global variables are handled by placing them in singleton groups, with a priority equal to the total savings that result in all subprograms by having the global register-resident.

Wall found improvements of from 10% to 28% in execution speed, while from 83% to 99% of all dynamic memory references to data were removed. Since Wall's scheme eliminates all saving and restoring, it works best when a large number of registers are available for allocation (52 in his tests). When fewer registers are available, saving and restoring must be included. Now the cost of giving a subprogram an extra register is compared with the benefit of having that register available for local use. If save-restore costs are less than the benefits, save and restore code is added.

When interprocedural effects are accounted for, it is possible to assign registers and position save-restore code in such a way that optimal register allocation is obtained [Kurlander Fischer 96]. The improvements in execution speed that result can sometimes be dramatic.

Some architectures, most notably the SPARC, provide register windows. When a call is made, the callee is provided its own "window" of registers, distinct from the caller's register window. This reduces the cost of calls as saving and restoring of registers is done automatically. Register windows are allowed to partially overlap to facilitate parameter passing through registers.

15.4 Code Scheduling

We have already discussed the issues of instruction selection and register allocation in code generation. Modern computer architectures have introduced a new problem—that of ordering (or scheduling) the instructions that are generated. Most modern computer architectures are pipelined. This means that instructions are processed in stages, with an instruction progressing from stage to stage until it is completed. A number of instructions can be in different stages of execution at the same time. This is very important since instruction execution overlaps, allowing much faster execution speeds.

What happens if one instruction being executed needs a value produced by an earlier instruction that hasn't yet completed execution? Normally this isn't a problem—pipelines are designed to make results available as soon as possible. In a few cases however, a needed operand may not be available. Then the pipeline must be

stalled, delaying execution of an instruction (and its successors) until the needed value is available.

Most current pipelined architectures are delayed load. This means that a register value created by a load instruction is not available for use by the very next instruction. Rather it is *delayed* for one or more instructions. For example on a MIPS R3000 processor, loads are delayed by one instruction. This delay is needed to allow the processor's cache to be searched for the desired operand. Thus the following instruction sequence, though valid, would stall:

```
lw  $12,b        # Load b into register 12
add $10,$11,$12 # Add reg 11 and reg 12 into reg 10
```

Stalls are not inevitable after a load. If another instruction can be placed between a load and the instruction that uses the loaded value, instruction execution proceeds without delay. Thus the following instructions would be delay-free:

```
lw  $12,b        # Load b into register 12
li  $11,100      # Load 100 into register 11
add $10,$11,$12 # Add reg 11 and reg 12 into reg 10
```

The role of instruction scheduling is to order instructions so that stalls (and their delays) are minimized.

Code scheduling is normally done at the basic block level. A basic block is a linear sequence of instructions that contains no branches except at its very end. Instructions within a basic block are always executed sequentially, as a unit. During code scheduling all the instructions within a basic block are analyzed to determine an execution order that produces correct computations with a minimum of interlocks or delays. We'll consider a simple but effective postpass approach devised by Gibbons and Muchnick [1986].

Postpass code schedulers operate after code has been generated and registers have been chosen. They are very general because they can handle code generated by any compiler (or even hand-coded assembly language programs). However because instructions and registers have already been selected, they can't modify choices already made, even to avoid interlocks.

A code scheduler tries to move apart instructions that will interlock. However, instructions can't be reordered haphazardly—loads of a register must precede use of that register, and stores of a register must follow instructions that compute the register's value. We use a dependency dag to represent dependencies between instructions. Nodes of the dag are instructions that are to be scheduled. An arc exists between two instructions *i* and *j* if instruction *i must* be executed before instruction *j*. Thus arcs are added between instructions that load or compute a register and instructions that use or store that register. Similarly an arc is added between a load from memory location *A* and a subsequent store into location *A*. Also an arc is added between a store into location *B* and any subsequent load or store involving location *B*. In the case of aliasing, we make worst-case assumptions. Thus a load through a pointer *P* must precede a store into any location *P* might alias, and a store through *P* must precede any load or store involving a location *P* might alias.

As an example, assume we generate the MIPS code shown in Figure 15.9 for the expression `a=((a*b)*(c+d))+(d*(c+d))`.

```
1. lw    $10,a         6.  add  $10,$10,$12
2. lw    $11,b         7.  mul  $11,$11,$10
3. mul   $11,$10,$11   8.  mul  $12,$10,$12
4. lw    $10,c         9.  add  $12,$11,$12
5. lw    $12,d         10. sw   $12,a
```

Figure 15.9 MIPS code for `a=((a*b)*(c+d))+(d*(c+d))`

Figure 15.10 illustrates the corresponding dependency dag. Double-circled nodes are loads—the critical nodes in this example because they can stall.

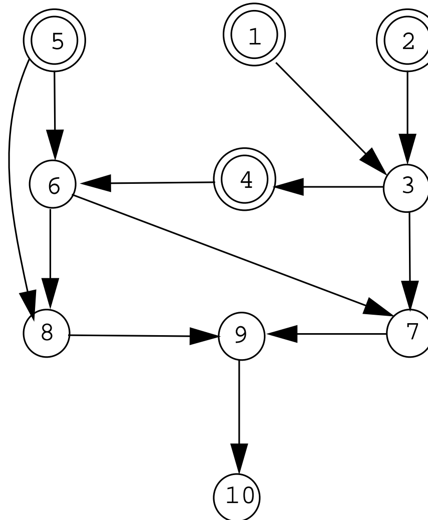


Figure 15.10 Dependency DAG for `a=((a*b)*(c+d))+(d*(c+d))`

Dependency dags have the property that any topological sort of the nodes represents a valid execution order. This is, as long as an instruction is scheduled before any of its successors in the dependency dag, it will execute properly. Any node that is a root of the dependency dag may be scheduled immediately. It is then removed from the dag, and again any root may be scheduled. Our goal in scheduling instructions will be to choose roots that avoid stalls. In fact the first rule in our scheduling algorithm is just that:

> When choosing a root to schedule, choose one that *won't* be stalled by the most recently scheduled node.

Sometimes we can't find a root that doesn't stall its predecessor—not all instruction sequences are stall-free.

If we find more than one root that doesn't stall its predecessor, secondary criteria apply. We try to select the "nastiest" root—the one most likely to cause future stalls, or to complicate the scheduling process. Three criteria are considered, in decreasing order of importance

1. Does the root stall any of its successors in the dependency dag?
2. How many new roots will scheduling this root uncover?
3. What is the longest path from this root to a leaf of the dependency dag?

If a root can stall a successor, we want to schedule it immediately so that other roots can be scheduled before the successor, avoiding a stall. If we schedule a root that exposes other new roots, we increase the range of choices available to the scheduler, simplifying its task. If we schedule a root with a long path to a leaf, we are attacking a "critical path," a long instruction sequence that allows the scheduler few choices in reordering instructions.

In our scheduling algorithm, scheduleDag, we'll use an operation select that takes a set of root nodes of the dependency dag, and a criterion. select will choose the nodes in the root set that meet the criterion, as long as the set selected in non-empty. That is, if no node in the set meets the criterion, select will return the entire input set. The reason for this is that a criterion that rejects all nodes is useless since our goal is to choose *some* node to schedule.

For example, select(nodeSet, "Has the longest path to a leaf") selects those nodes in nodeSet with the greatest distance to a leaf (several nodes may be selected if they all share the same maximum distance to a leaf). However, select(nodeSet, "Can stall some successor") would return all of nodeSet if no member of the set had a successor that it stalled. Once we have refined a nodeSet to a single node, further applications of select are unnecessary; they will have no effect.

```
   scheduleDag( dependencyDag )
1.    while candidates ← roots(dependencyDag) ≠ ϕ
2.          do select(candidates, “Is not stalled by last instruction generated”)
3.             select(candidates, “Can stall some successor”)
4.             select(candidates, “Exposes the most new roots if generated”)
5.             select(candidates, “Has the longest path to a leaf”)
6.             Let Inst ∈ candidates
7.             Schedule Inst as next instruction to be executed
8.             dependencyDag ← dependencyDag - Inst
```

Figure 15.11 An Algorithm to Schedule Code from a Dependency Dag.

The complete definition of scheduleDag is shown in Figure 15.11. An an example, consider the dependency dag of Figure 15.10. The code originally generated (Figure 15.9) contains two stalls (after instructions 2 and 5). The initial set of roots is 1, 2 and 5, all load instructions. All roots can stall a successor instruction and none expose a new root if scheduled. Both 1 and 2 have the longest path to a leaf, so 1 is arbitrarily chosen and scheduled. The root set is now 2 and 5. Instruction 2 is chosen because it exposes a new root, 3. Next 5 is chosen because it can

stall a successor. Instructions 3, 4 and 6 are chosen next, as they form, in turn, singleton root sets. Instructions 7 and 8 are the new root set; 7 is arbitrarily chosen, then 8, 9 and 10. The code we produce is shown in Figure 15.12.

```
1. lw    $10,a          6.  add  $10,$10,$12
2. lw    $11,b          7.  mul  $11,$11,$10
3. lw    $12,d          8.  mul  $12,$10,$12
4. mul   $11,$10,$11    9.  add  $12,$11,$12
5. lw    $10,c          10. sw   $12,a
```

Figure 15.12 Scheduled MIPS code for `a=((a*b)*(c+d))+(d*(c+d))`

15.4.1 Improving Code Scheduling

The code shown in Figure 15.12 is not prefect. A stall still occurs after the fifth instruction. In fact, using just three registers a stall *can't* be avoided. It is shown in [Kurlander et al. 95] that sometimes an additional register is needed to avoid all stalls. One way to improve the code produced by scheduleDag is to reallocate registers in the initial code sequence, using an extra register beyond the original allocation.

To do this, we find instructions that stall and try to move then "up" in the instruction sequence. If we can't move a stalling instruction earlier because it assigns to a register used by the preceding instruction, we reallocate the register assigned to by the stalling instruction to be a register unused by the preceding instruction. Because we've added an extra register, we can always find an unused register, and move the stalling instruction at least one position earlier in the execution sequence.

For example, reconsidering Figure 15.12, instruction 5 (a load) stalls because `$10` is used in instruction 6. We can't move instruction 5 up because instruction 4 uses a previous value of `$10`, loaded in instruction 1. If we add an additional register, `$13`, to our allocation, we can load it in instruction 5 (taking care to reference `$13` rather than `$10` in instruction 6.) Now instruction 5 can be moved earlier in the sequence, avoiding a stall. The resulting delay-free code is shown in Figure 15.13.

```
1. lw    $10,a          6.  add $10,$13,$12
2. lw    $11,b          7.  mul $11,$11,$10
3. lw    $12,d          8.  mul $12,$10,$12
4. lw    $13,c          9.  add $12,$11,$12
5. mul   $11,$10,$11    10. sw  $12,a
```

Figure 15.13 Delay-free MIPS code for `a=((a*b)*(c+d))+(d*(c+d))`

It is evident that there is a tension between code scheduling, which tries to increase the number of registers used (to avoid stalls), and code generation, which

seeks to reduce the number of registers used (to avoid spills and make registers available for other purposes). An alternative to postpass code scheduling is an *integrated approach* that intermixes register allocation and code scheduling.

The Goodman Hsu [1988] algorithm is a well-known and widely used integrated register allocator and code scheduler. As long as registers are available, it uses them to improve code scheduling by loading needed values into distinct registers. This allows loads to "float" to the beginning of the code sequence, eliminating stalls in later instructions that use the loaded values. When registers grow scarce, the algorithm switches emphasis, and begins to schedule code to free registers. When sufficient registers are available, it resumes scheduling to avoid stalls. Experience has shown that this approach balances nicely the need to use registers sparingly and yet avoid stalls whenever possible.

15.4.2 Global and Dynamic Code Scheduling

Although we have focused on code scheduling at the basic block level, it is possible to schedule code at the global level [Bernstein & Rodeh 1991]. Instructions may be moved upward, past the beginning of a basic block to predecessor blocks in the control flow graph. We may need to move instructions out of a basic block because basic blocks are often very small—sometimes only an instruction or two in size. Moreover, certain instructions, like loads and floating point multiplies and divides, can incur long latencies. For example, a load that misses in the primary cache may stall for 10 or more cycles; a miss in the secondary cache—to main memory—can cost 100 or more cycles.

As a result, code schedulers often seek to move loads as early as possible in an instruction sequence. There are several complicating factors, however. To what predecessor block should we move an instruction? Ideally, to a predecessor that is control equivalent; that is, a predecessor that will be executed if, and only if, the current block is. An example of this is moving an instruction that follows an `if` statement to a position that precedes the `if` (and thereby past both arms of the `if`). An alternative is to move an instruction to a block that dominates it (that is, to a block that is a necessary predecessor). Now however, the moved instruction may be speculative—it may be executed unnecessarily on some execution paths. Thus if an instruction is moved from a `then` part to a position above the `if`, the instruction will be executed even when the `else` part is selected. Speculative instructions may waste computational resources, by executing useless instructions. What's worse, if a speculative instruction faults (e.g., a load through a null or illegal pointer), a false run-time error may be created.

Even if we can move an instruction freely upward, how far should we move it? If we move the instruction too far forward, it will "tie up" a register for an extended period, making register allocation harder and less effective. Some architectures, like the DEC alpha, provide a prefetch instruction. This instruction allows data to be loaded into the primary cache in advance, reducing the chance that a register load will miss. Again, placement of preloads is a tricky scheduling issue. We want to preload early enough to hide the delays incurred in loading the

cache. But, if we preload too early, we may displace other useful cache data, causing cache misses when *these* data are used.

A number of recent computer architectures (MIPS R10000, Intel Pentium Pro) have included a sophisticated dynamic scheduling facility. These designs, sometimes called out of order architectures, delay instructions that aren't ready to execute, and dynamically choose successor instructions that *are* ready to execute. These designs are far less sensitive to compiler-generated code schedules. In fact, dynamically scheduled architectures are particularly effective in executing old programs ("dusty decks") that were created before code scheduling was even invented.

Even with dynamically scheduled architectures compiler-generated code scheduling is still an important issue. Loads, especially loads that frequently miss in the primary cache, must be moved early enough to hide the long delays a cache miss implies. Even the best current architectures can't look dozens or hundreds of instructions ahead for a load that might miss in the cache. Rather, compilers must identify those instructions that might incur the greatest delays and move them earlier in the instruction sequence.

15.5 Automatic Instruction Selection

An important aspect of code generation is instruction selection. After a translation for a particular construct is determined, the machine level instructions that implement the translation must be chosen. Thus if we decide to implement a `switch` statement using a jump table (see Section 13.1.5), instructions that index into the jump table and then do an indirect jump must be generated.

Often several different instruction sequences can implement a particular translation. Even something as simple as `a + 1` can be implemented by loading `1` into a register and generating an add instruction, or by generating an increment or add immediate instruction. We normally want the smallest or fastest instruction sequence. Thus an add immediate, because it avoids an explicit load, is preferred.

In simple RISC architectures, the choice of potential instruction sequences is limited because almost all operands must be loaded into registers before they can be used (immediate operands being a notable exception). Further, the variety of addressing modes provided is also spartan—often only absolute and indexed addresses are allowed.

Older architectures, like the Motorola 680x0 and Intel x86, are much more elaborate. Many different operation codes are provided, and a wide variety of addressing modes are available. Operands need not always be loaded into registers, addressing modes can fetch operands indirectly and can increment and decrement registers. Different register classes (e.g., address registers and data registers) are used in different instructions (in a non-interchangeable manner) and particular registers are sometimes "wired into" certain instructions.

For very complex architectures, a way of systematizing and automating instruction selection is vital. Even for simpler architectures, it may be necessary to "extend" a code generator when a successor architecture is introduced. Very

ambitious compilers may aim to compile into more than one target architecture, mandating alternative instruction sequences for different target machines.

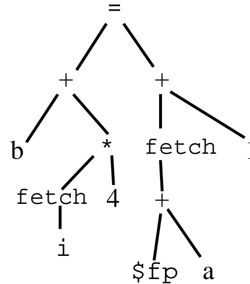


Figure 15.14 A Low-level IR Representation of `b[i]=a+1`

Instruction selection is often simplified by translating source language constructs into a very low-level tree-structured intermediate representation (IR). In this IR, leaves represent registers, memory locations, or literal values, and internal nodes represent basic operations on operand values. Detailed data access patterns and manipulations are exposed. (For an excellent example of a low-level tree-structured IR see [Appel 87]) Consider the statement `b[i]=a+1`, where `b` is an array of integers, `i` is a global integer variable, and `a` is a local variable accessed through the frame register, `$fp`. The statement's tree-structured IR is shown in Figure 15.14. Note that leaves corresponding to identifiers are their addresses (if globals) or offsets (if locals). Explicit memory fetches (using the `fetch` operator) are shown, as is the multiply by 4 needed to build a valid word address for an element of an array of integers.

A tree-structured IR may also be used to define the effect of each instruction of a computer. A tree defines the computation performed by the instruction as well as the kind of value it produces. This is illustrated in Figure 15.15 in which tree-structured patterns (or productions) are used to define valid IR trees.

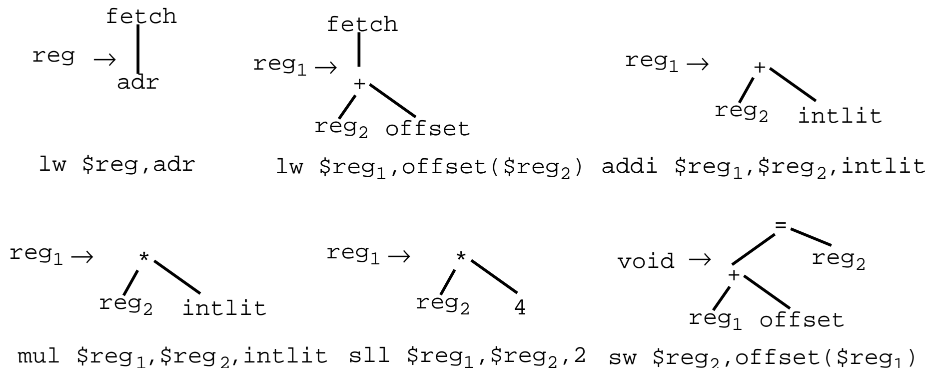


Figure 15.15 IR Tree Patterns for Various MIPS Instructions.

Now instruction selection for a given IR tree becomes a matter of matching instruction patterns against the generated IR such that the IR tree is covered (parsed) with adjacent patterns. That is, we find a subtree in the IR translation that matches exactly the pattern for some instruction. That subtree is then replaced with the pattern's left hand side. The process is repeated until the entire IR tree is reduced to a single node. This is very similar to ordinary bottom-up parsing (Chapter 6).

As instruction patterns are matched, their corresponding machine language instructions are generated. Registers can be allocated "on the fly" using the techniques of Section 15.3.1. Alternatively, pseudo-registers can be allocated as code is generated, and then later mapped to real registers using the graph coloring techniques of Section 15.3.2.

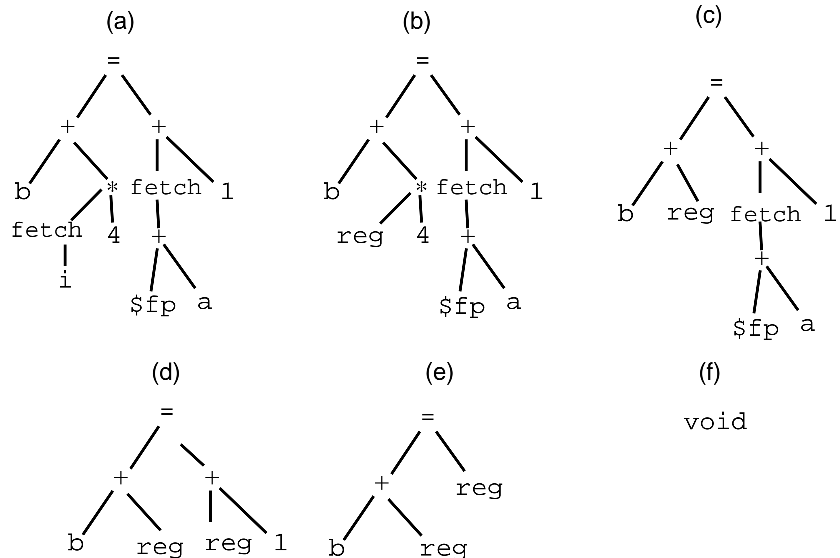


Figure 15.16 Instruction Selection Using Patterns.

As an example, reconsider the IR tree corresponding to `b[i]=a+1` (Figure 15.16 (a)). We first match a load of `i` (b). Next, a multiply by 4 is matched (c). Then an indexed load is generated for `a` (a local variable), (d). Finally, an add immediate (e) and a store instruction, (f), reduce the IR tree to `void`. The instructions generated (assuming calls to getReg and freeReg as code is generated) are shown in Figure 15.17.

15.5.1 Instruction Selection Using BURS

It is often the case that more than one instruction sequence can implement the same construct. In terms of IR trees, different reductions of the same tree, yielding

```
lw      $t1,i
mul     $t1,$t1,4
lw      $t2,a($fp)
addi    $t2,$t2,1
sw      $t2,b($t1)
```

Figure 15.17 MIPS code for `b[i]=a+1`

different instruction sequences, may be possible. How can we choose the instruction sequence to be generated?

A very elegant approach involves assigning costs to instruction patterns. The cost of an instruction is set when a code generator is built. This cost may be size of an instruction, or its execution speed, or the number of memory references the instruction makes, or any criterion that measures how "good" an instruction is. When given a choice, we'll prefer a cheaper instruction over a more expensive one.

Now matching of instruction patterns to an IR tree is generalized so that a least-cost cover is obtained. That is, the pattern matcher guarantees that the matches it selects have the lowest possible cost. Thus using the measure of quality selected when the code generator was built, the best possible instruction sequence is generated.

To guarantee that a least-cost cover of an IR tree is found, we use dynamic programming. Starting at the leaves of the tree, we mark each leaf with the lowest cost possible to reduce the leaf to each of the nonterminals. (Nonterminals, as in context-free productions, are the symbols that appear on the left-hand side of instruction patterns). Next interior nodes just above the leaves are considered. Each instruction pattern that correctly matches the interior node and has the correct number of children is considered. The cost of the pattern plus the costs of the node's children are considered. The node is marked with the cheapest cost possible to reduce it to each nonterminal. We continue traversing the IR tree, until the root node is reached. The lowest cost found to reduce it to any nonterminal is selected as the best (least-cost) cover.

IR trees for a large program or subroutine can easily comprise tens of thousands of nodes. The extensive processing needed for each node would appear to make least-cost instruction selection using patterns a very slow process. Happily this is not the case.

An approach based on BURS theory (Bottom Up Rewrite Systems) [PG 88] allows very fast instruction selectors (and code generators) to be built. Code generators built using BURS theory can be extremely fast because all dynamic programming is done in advance when a special BURS automaton is built. During compilation it is only necessary to make two traversals of the IR tree: one bottom-up traversal to label each node with a state that encodes all optimal matches and a second top-down traversal that uses these states to select and generate code. It has been reported that careful encodings can produce an automaton that executes fewer than 90 RISC instructions per node to do both traversals.

The automaton that labels the tree is a simple finite state machine, similar to that used in shift-reduce parsers (Chapter 6). A bottom-up walk of the tree is performed, and the label for any given node is determined by a table lookup given the operator at the node and the states that label each of its children. The automaton that emits code is equally simple in design. The code to be emitted is determined by the state that labels a node and by the nonterminal to which that node should be reduced—another table lookup.

As an example, the instruction patterns of Figure 15.15 would all be given a cost of 1 except for `mul`, which would be given a cost of 3. This is because `mul` is actually implemented by the MIPS assembler using three hardware instructions, whereas all the other instructions are implemented using a single instruction. Returning to the example of Figure 15.14, all the leaves would be labeled with a state indicating that no reductions of leaf nodes are possible—the leaves must all be matched directly. Visiting `i`'s parent with its state, the `fetch` would be labeled with a state indicting that application of an `lw` pattern is possible, at a cost of 1. Going to its parent (a `*` operator), the state reached would show that although two reductions are possible (patterns for both `mul` and `sll` match); `sll`, being cheaper, will apply. That is, the instruction selector has recognized a well-known trick—multiplication by a power of two can often be implemented more efficiently by doing a left shift rather than an explicit multiply.

Continuing, the rest of the nodes are labeled, with the remaining matches being identical to those illustrated in Figure 15.16. The state labeling the root tells us that the final instruction to be generated (to implement the assignment) will be a `sw`. The two subtrees are visited to generate the instructions needed to implement them. We therefore generate the root's instruction after returning from recursive visits to both children, guaranteeing that the store's operands are computed prior to its execution. We generate the code shown in Figure 15.18.

```
lw      $t1,i
sll     $t1,$t1,2
lw      $t2,a($fp)
addi    $t2,$t2,1
sw      $t2,b($t1)
```

Figure 15.18 Improved MIPS code for `b[i]=a+1`

Two difficulties arise in creating a BURS-style code generator: efficiently generating the states and state transition tables (because *all* potential dynamic programming decisions are done at table generation time, they must be done efficiently) and creating an efficient encoding of the automata for use in a compiler. Fraser and Henry discuss a solution to the encoding problem in [FH 91]. Proebsting created BURG, [Pro95], a simple and efficient tool for generating BURS-style code generators. Using a very clean implementation and ingenious state elimination techniques, least-cost code generators for a variety of architectures can be created in a few seconds.

15.5.2 Instruction Selection Using Twig

Other code generation systems based on tree pattern matching and dynamic programming have been developed. They differ primarily from BURS in how they do tree pattern matching and in the fact that they do dynamic programming at compile-time rather than compile-compile time.

Aho, Ganapathi, and Tjiang [AGT 89] created a tree manipulation language and system called Twig. Given a specification of tree patterns and associated costs, Twig generates a top-down tree automaton that will find the least-cost cover of a subject tree. Twig uses fast top-down Hoffmann-O'Donnell [HO 82] pattern matching in parallel with dynamic programming to find the least-cost cover.

Starting at the root of possible instruction trees, paths to each of the tree's children are traced. Whenever a such a path is correctly traced, a counter is incremented. When the counter equals the number of children a pattern tree has, a potential match is recognized. Using costs and dynamic programming, the least-cost cover for an entire IR tree can be found.

The costs associated with patterns in Twig are more general than those afforded by any BURS system. Twig may compute the cost of a pattern dynamically—depending on semantic information available at compile-time. This flexibility further allows Twig to abort certain matches if semantic predicates are not satisfied. Thus, the applicability of Twig's patterns is context sensitive. BURS does not have this flexibility since all costs must be fixed prior to compilation to allow precomputation of dynamic programming decisions. The great advantage of BURS is its speed. *All* possible matches are anticipated in advance and tabulated. Twig must recognize partial matches and update counters as instruction selection proceeds. Given the huge IR trees that often need to be translated, even a little extra processing at each node can represent a significant slowdown.

15.5.3 Other Approaches

One of the first instruction selection techniques based on tree rewriting was that of Cattell [Cat 80]. First the effect of each instruction was described, using a register-transfer notation. Then a code generator "discovered" appropriate code sequences by matching instructions against IR trees. That is, the code generator explored ways to decompose an IR tree into combinations of special primitive trees, using backtracking if necessary. Because this process could be very slow, a catalog of the tree patterns that are implemented was precomputed. At compile-time this catalog was searched to find available instruction sequences.

Glanville and Graham [GG 78] observed that the problem of matching code templates against an IR tree is very similar to the problem of matching productions against a token sequence during parsing. They cleverly reformulated the template-matching problem in context-free parsing terms. Using standard shift-reduce parsers, augmented to handle multiple template matches, instruction selection could be automated.

A limitation of the Graham-Glanville approach is that it is purely syntactic. It simply matches, in a context-free manner, sequences of symbols. Ganapathi and

Fischer [GF 85] suggested adding attributes to code templates. Attributes allow types, sizes and values to influence instruction selection.

The Back End Generator (BEG) [ESL 89] finds a least-cost cover of the tree using dynamic programming techniques that are essentially identical to Twig's. Like Twig, BEG can guard patterns with semantic predicates. A BEG specification, in addition to having instruction patterns, includes a description of the register set of the target machine. This specification automatically generates the register allocator. Experiments show code quality and code generation times to be comparable to handwritten code generators.

Fraser, Hanson, and Proebsting [FHP 92] developed a code-generator generator based on naive pattern matching and dynamic programming. This system, iburg, maintains the same interface as BURG. Although iburg code generators are slower than those generated by BURG, iburg presents a simple and efficient framework for the development of pattern-based code generators.

15.6 Peephole Optimization

To produce high-quality code, it is necessary to recognize a multitude of special cases. For example, it is clear we would like to avoid generating code for an addition of zero to an operand. But where should we check for this special case? In each semantic routine that might generate an add? In each code-generation routine that might emit an add instruction?

Rather than distribute knowledge of special cases throughout semantic or code-generation routines, it is often preferable to utilize a distinct peephole optimization phase that looks for special cases and replaces them with improved code. Peephole optimization may be performed on ASTs, IR trees [Tan 82] or generated code [McK 65]. As the term "peephole" suggests, a small window of two or three instructions or nodes is examined. If the instructions in the peephole match a particular pattern, they are replaced with a replacement sequence. After replacement, the new instructions are reconsidered for further optimization.

In general, we represent the collection of special cases that define a peephole optimizer as a list of pattern-replacement pairs. Thus, *pattern* ⇒ *replacement* means that if a instruction sequence or tree matching the pattern is seen, it is replaced with the replacement sequence. If no pattern applies, the code sequence is unchanged. Clearly the number of special cases that might be included is unlimited. We will illustrate where peephole optimization can be employed, and the kinds of optimizations that can be realized.

15.6.1 Levels of Peephole Optimization

In general there are three places where peephole optimization may profitably be employed. After parsing and typechecking, a program is represented in AST form. Here peephole optimization may be used to optimize the AST, recognizing special cases at the source level that are independent of how a construct is translated, or the code that is generated for it.

After translation, a program is represented in an IR or bytecode form. Here peephole optimization can recognize optimizations that simplify or restructure an IR tree or bytecode sequence. These optimizations are independent of the actual target machine or the exact code sequences used to implement an IR tree or bytecodes.

Finally, after code generation peephole optimization can replace pairs or triples of target machine instructions with shorter or simpler instruction sequences. At this level, the optimization is highly dependent on the details of a machine's instruction set.

AST Level Optimizations. In Figure 15.19 we illustrate optimizations that can simplify or improve an AST representation of a program. In (a) an `IF` whose condition is always true is replaced with the body of the conditional. In (b) and (c), expressions involving constant operands are "folded" (replaced with the value of the expression). This folding optimization can expose other optimizations (such as the conditional replacement optimization of (a)).

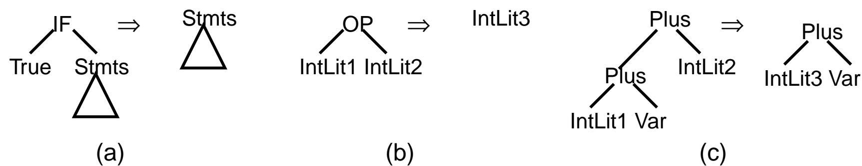


Figure 15.19 AST Level Peephole Optimization.

Optimizations at the AST level can conveniently be implemented using a tree rewriting tool like BURS. Source patterns are first recognized and labeled. Then during the "processing" traversal, trees can be rewritten into the target form. If necessary, an AST can be traversed several times, so that rewritten ASTs can be matched and transformed several times.

IR Level Optimizations. As illustrated in Figure 15.20, a variety of useful optimizations can be performed at the IR level. In (a) and (b), constant folding is specified. Since some arithmetic operations are exposed only after translation (e.g., indexing arithmetic), folding can be done at both the AST and IR levels. In (c), multiplication by a power of 2 is replaced with a left shift operation. In (d) and (e) identity operations are removed. In (f) the commutativity of addition is exposed, and in (g) addition of a negative value is transformed into subtraction.

Transformations on IR trees can be conveniently implemented using a tool like BURS.

As illustrated in Figure 15.21, optimizations corresponding to those of Figure 15.20 can be applied to a bytecode representation of a program. This level of optimization may be appropriate if bytecodes are later expanded into target machine code. Alternatively, the machine-level optimizations described in the next section

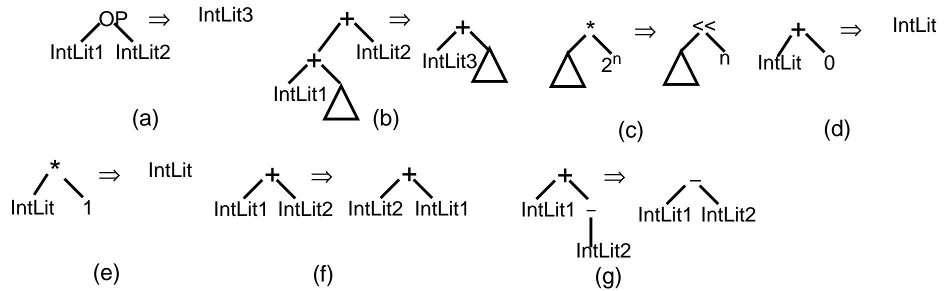


Figure 15.20 IR Level Peephole Optimizations.

may be applied to bytecodes, since bytecodes share much of the structure of conventional machine code.

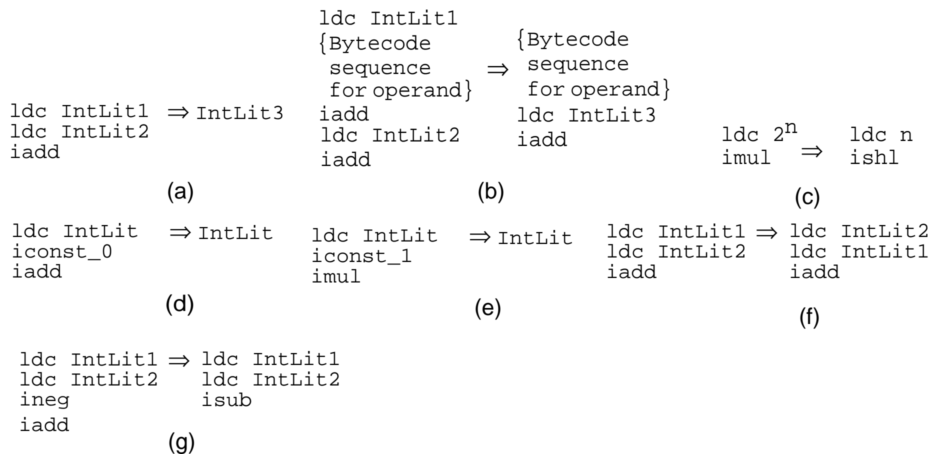


Figure 15.21 Bytecode Level Peephole Optimizations.

Code Level Optimizations. Figure 15.22 illustrates some simple peephole optimizations performed after code generation. In (a) a conditional branch around an unconditional branch is replaced with a single conditional branch (with the sense of the test inverted). In (b), a branch to the next instruction is removed (this is sometimes generated when a `then` or `else` part of an `if` is null). A branch to a second branch can be collapsed to a direct branch to the final target (c). In (d) a move from a register to itself is suppressed (this sometimes happens when a special register, like a parameter register, is loaded with a value that already is in the correct register). In (e) a register is stored into a location and then that same register is immediately reloaded from the same location; the load is unnecessary and may be deleted.

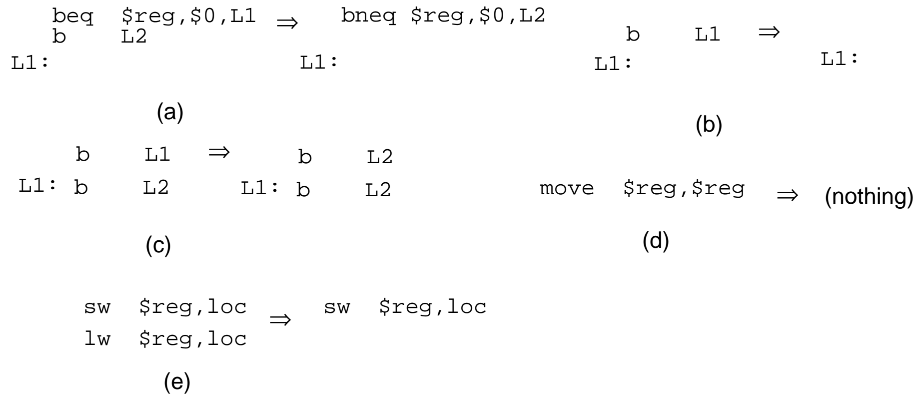


Figure 15.22 Code Level Peephole Optimizations.

More elaborate architectures present additional opportunities for peephole optimization. If a special increment or decrement instruction is available, it can replace an ordinary add immediate (which usually is longer and a bit slower). If auto-increment or auto-decrement addressing modes are available, these can be used to "hide" an explicit increment or decrement of an index. Some architectures have a special "loop control" instruction that decrements a register and conditionally branches if it is zero.

Recognizing replacement patterns must be done quickly, if peephole optimization is to be fast. Operator-operand combinations are hashed to applicable patterns. Also, the size of a peephole window is normally limited to two or three instructions. Using a careful hashed implementation, speeds of several thousand instructions per second have been achieved [DF 84].

The concept of analyzing physically adjacent instructions has been generalized to logically adjacent instructions [DF 82]. Two instructions are logically adjacent if they are linked by flow of control or if they are unaffected by intervening instructions. (The "branch chain" of Figure 15.22 (c) is a good example of this.) By analyzing logically adjacent instructions it is possible to remove jump chains (jumps to jump instructions) and redundant computations (for example, unnecessarily setting a condition code). Detecting logical adjacency can be costly, so care is required to keep peephole optimization fast.

15.6.2 Automatic Generation of Peephole Optimizers

In [FD 80] ways of automating the creation of peephole optimizers are discussed. The idea is first to define the effect of target machine instructions at the register-transfer level. At this level, instructions are seen to modify primitive hardware locations, including memory (represented as a vector *M*), registers (represented as a vector *R*), the *PC* (program counter), various condition codes, and so

on. A target machine instruction may have more than one effect, and its definition at the register-transfer level may include more than one assignment.

The peephole optimizer (PO) operates by considering pairs of instructions, expanding them to their register-transfer level definitions, simplifying the combined definitions, and then searching for a *single* instruction that has the same effect as the combined pair.

To be applicable, an instruction must perform all the register transfers of the combined instructions. It may also do other register transfers as long as these are dead (and therefore have no effect on subsequent computations). Thus an instruction may set a condition code, even if this is not wanted, as long as the updated condition code is not referenced by later instructions.

Instruction pairs that start with a conditional branch get special treatment. In particular, the second instruction is prefixed with a conditional representing the negation of the original condition (the only way the second instruction is executed is if the conditional branch fails). An unconditional branch is paired with its target instruction. This pairing often allows jump chains (a jump to another jump) to be collapsed. Note, however, that instruction pairs with the second instruction labeled are not optimized. This situation is needed to make jumps to such labels work correctly. However, if all references to a label are removed by the PO, then the label itself is also removed, possibly allowing new optimizations to be discovered.

The analysis and simplification of the instructions just described are not actually done during compilation because this would be far too slow. Rather, representative samples of actual programs are analyzed in advance, and the most common peephole optimizations are stored in a table. During compilation, this table is consulted to determine if the instructions currently in the peephole may be optimized.

Exercises

1. Consider the following Java function:

```
public static int fact(int n){
   if (n == 0)
      return 1;
   else return n*fact(n-1); }
```

   Show the JVM bytecodes that would be generated for this method. Explain how these bytescodes would be translated to target machine code using the techniques of Section 15.1.

2. Recall that in Section 13.1.5 switch statements were translated using either a `tableswitch` or `lookupswitch` bytecode. Using either the MIPS architec-

ture or your favorite processor architecture, explain how the `tableswitch` and `lookupswitch` can be efficiently translated into machine-level instructions.

3. On many processors certain registers *must* be used to hold a parameter to a subprogram or a return value from a function. Suggest how the techniques of Section 15.1 could be extended so that when bytecodes are translated, parameters and return values are computed directly into the required register (without any unnecessary register to register moves).
4. Recall that a key to generating efficient target-machine code from bytecodes is to avoid explicit stack manipulations for bytecode operands. Rather, machine registers are used.

   Assume we use the techniques of Section 15.3.1 to allocate registers "on the fly." Explain how we could tag each bytecode, prior to code generation, with the machine registers the bytecode will use for its operands and result value. (These tags would then be used to "fill in" register names when bytecodes are expanded to machine code.)
5. A common subprogram optimization is inlining. At the point of a method call, the body of the method is substituted for the call, with actual parameter values used to initialize local variables that represent parameters.

   Assume we have the bytecodes that represent the body of subprogram `P` that is marked `private` or `final` (and hence can't be redefined in a subclass). Assume further that `P` takes `n` parameters and uses `m` local variables. Explain how we could substitute the bytecodes representing `P`'s body for a call to `P`, prior to machine code generation. What changes in the body must be made to guarantee that the substituted bytecodes don't "clash" with other bytecodes in the context of call?
6. Show the expression tree, with registerNeeds labeling, that corresponds to the expression `a+(b+(c+((d+e)*(f/g))))`.

   Show the code that would be generated using the treeCG code generator.
7. Recall that registerNeeds gives the *minimum* number of register needed to evaluate an expression without spilling registers to memory, Show that there exist expressions of *unbounded* size that require only 2 registers for evaluations. Show that for any value of m there exist expressions that always require *at least* m registers.
8. Some computer architectures include an immediate operation of the form

   ```
   op $reg1,$reg2,val
   ```

   that computes `$reg1` = `$reg2` `op` `val`. In an immediate instruction `val` does not need to be loaded into a register; it is extracted directly from the instruction's bit pattern.

   Explain how to extend registerNeeds and treeCG to accommodate architectures that include immediate operations.
9. Sometimes the code generated for an expression tree can be improved if the associative property of operators like + and * is exploited. For example, if the

following expression is translated using treeCG, four registers will be needed:
`(a+b) * (c+d) * ((e+f) / (g-h))`

Even if the commutativity of + and * is exploited, four registers are still required. However, if the associativity of multiplication is exploited to evaluate multiplicands from right to left, then only three registers are needed. [First `((e+f) / (g-h))` is evaluated, then `(c+d) * ((e+f) / (g-h))`, and finally `(a+b) * (c+d) *((e+f) / (g-h))`.]

Write a routine associate that reorders the operands of associative operands to reduce register needs. (*Hint:* Allow associative operators to have more than two operands.)

10. In Section 15.4 we saw that many modern architectures are delayed load. That is, a value loaded into a register may not be used in the next instruction; a delay of one or more instructions is imposed (to allow time to access the cache).

    The treeCG routine of Section 15.2 is not designed to handle delayed loads. Hence, it almost always generates instruction sequences that stall at selected loads.

    Show that if an instruction sequence (of length 4 or more) generated by treeCG is given an additional register, it is possible to reorder the generated instructions to avoid *all* stalls for a processor with a one instruction load delay. (It will be necessary to reassign the register used by some operands to utilize the extra register).
11. Following the example of `doubleSum` in Section 15.3, convert the `stringSum` function of Section 15.1 into a form that makes explicit temporaries, live ranges, and parameter and return value register assignments. Then create the interference graph for `stringSum`. Use this interference graph and GCRegAlloc to assign registers to `stringSum`, assuming three registers are available (including `$a0`, the parameter register and `$vo`, the return value register).
12. Assume we have the following method

```
int f(int i) {
   g(1,i);
}
```

    At the point where the second parameter of `g` is loaded, we have a conflict if we require that parameters be passed in registers. In particular, `i` is passed in the first parameter register. But when the second parameter of `g` is loaded, the first parameter register is loaded with the value 1, possibly making `i` inaccessible. How can a register allocator deal with the problem of reuse of dedicated parameter registers? That is, what rules should be followed in determining where a parameter value is to be allocated throughout a program or subprogram?
13. In GCRegAlloc we spill a live range if we are unable to color it. An alternative to spilling a live range is to split it, as is done in PriorityRegAlloc. What

changes are needed in GCRegAlloc if we split an uncolorable live range rather than spill it?

14. At the site of a method call, we may need to save registers currently in use (lest they be overwritten by the method about to be executed). Assume we allocate registers using GCRegAlloc. Explain how to determine which registers are in use at a method call.
15. Assume we have n registers available to allocate to a subprogram. Explain how, using either GCRegAlloc or PriorityRegAlloc, we can estimate the total cost of register spills within the subprogram. How could this cost estimate be used in deciding how many registers to allocate to a subprogram?
16. In performing "on the fly" register allocation, some implementations store freed registers on a stack. Thus the most recently freed register will be the next register to be allocated. On the other hand, other implementations place freed registers at the back of a queue. Thus the least-recently freed register will be the next to be allocated.

    From the point of view of a postpass code scheduler, which of the register reallocation implementations (stack vs. queue) is preferable? Why?
17. The scheduleDag code scheduler of Section 15.4 assumes that instructions that can stall have unit delay. That is, one instruction must separate an instruction that can stall from the first use of the value it produces. It may happen that some instructions have **n** cycle delays, meaning n instructions must separate the instruction from the first use of the value it produces.

    How must scheduleDag be modified to handle instructions that have n cycle delays?
18. The scheduleDag code scheduler is a post-pass scheduler. That is, it schedules instructions *after* registers have been allocated. It is possible to create a dependency dag in terms of instructions that reference pseudo-registers. After instructions are scheduled, the pseudo-registers are mapped to real registers. Such a scheduler is a pre-pass scheduler, since it operates *before* register allocation.

    It is important to note that the order in which instructions are scheduled will affect the number of registers that later are needed. For example, scheduling all loads immediately will force each load to use a different register. Scheduling some loads after other operations may allow registers to be reused.

    If scheduleDag is used as a pre-pass code scheduler, how should it be modified so that the number of pseudo-registers in use is a criterion in selecting the next instructions to schedule? That is, scheduling an instruction that increases the number of registers that will be needed should be discouraged unless it serves to avoid stalls in the code schedule.
19. It is sometimes the case that we need to schedule a small block of code that forms the body of a frequently executed loop. For example

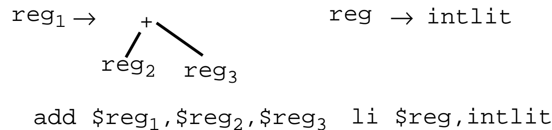
```
for (i=2; a <1000000; i++)
   a[i] = a[i-1]*a[i-2]/1000.0;
```

Operations like floating point multiplication and division often have significant delays (5 or more cycles). If a loop body is small, code scheduling can't do much—there aren't enough instructions to cover all the delays. In such a situation loop unrolling may help. The body of loop is replicated n times, with loop indices and loop limits suitably modified. For example, with n = 2, the above loop would become

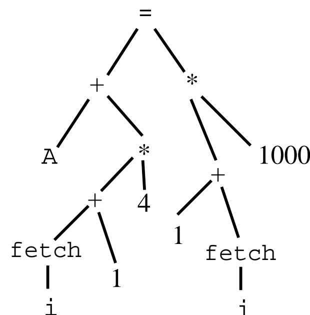
```
for (i=2; a <999999; i+=2){
    a[i] = a[i-1]*a[i-2]/1000.0;
    a[i+1] = a[i]*a[i-1]/1000.0;}
```

A larger loop body gives a code scheduler more instructions that can placed after instructions that may stall. How can we determine the value of n (the loop unrolling factor) necessary to cover all (or most) of the delays in a loop body? What factors limit how large n (or an unrolled loop body) should be allowed to become?

20. The scheduleDag code scheduler is very optimistic with respect to loads—it schedules them assuming that they *always* hit in the primary cache. Real loads are not always so co-operative. Assume we can identify load instructions most likely to miss. How should scheduleDag be modified to use "probability of cache miss" information in scheduling instructions?
21. Assume we extend the IR tree patterns defined in Figure 15.15 with the following patterns for the MIPS add and load immediate instructions:



Show how the following IR tree, corresponding to `A[i+1] = (1+i)*1000`, would be matched. What MIPS instructions would be generated?



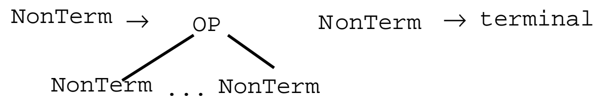
22. Code generators that use IR tree pattern matching still have the problem of allocating registers for the generated code. Suggest how an "on the fly" regis-

ter allocator can be integrated with pattern matching to form a complete code generator.

23. Instructions like the MIPS load immediate instruction are complicated by the fact that the immediate operand may be too big to fit in a single instruction. In fact immediate operands that are too big force two instructions to be generated—a "load upper immediate" that fills in the upper half of a word followed by an or immediate that fills in the lower half of a word.

    How can costs and IR tree patterns be used to specify to an instruction selector that two alternative translations are possible depending on the *size* of an immediate operand?

24. Assume we have tree-structured instruction patterns limited to the two forms shown below



    That is, a nonterminal may generate a single terminal symbol or it may generate an operator, all of whose children are nonterminals.

    Give an algorithm that can walk any IR tree and determine whether it can be covered (matched) using a set of productions limited to the two forms described above.

25. Assume that we now add cost values (integer literals greater than or equal to 0) to instruction patterns limited to the two forms described in Exercise 24. Extend the algorithm you proposed in Exercise 24 so that it now finds a least-cost cover. That is, your algorithm should choose productions that minimize the overall cost of matching a given IR tree.

26. The following instruction sequence often appears in Java programs:

```
a[i] = ...
... = a[i];
```

    That is, an element of an array is stored, then that same element is immediately reused. Suggest a peephole optimization rule, at the bytecode level, that would recognize this situation and optimize it using the `dup` bytecode.

27. Machines like the MIPS and SPARC have delayed branch instructions. That is, the instruction immediately following a branch is executed prior to transferring control to the target of the branch.

    Often, compilers simply generate a `nop` instruction after a branch, effectively hiding the effects of the delayed branch. Suggest a peephole optimization pattern for unconditional branches followed by a `nop` that swaps the instruction prior to the branch into the "delay slot" that follows it. Can this optimization always be done, or must some conditions be met to make the swap valid?

Now consider a delayed conditional branch in which the value of a register is tested. If the condition is met, the instruction following the conditional branch is executed, and then the branch is taken. Otherwise, instructions following the conditional branch are executed (as usual); no branch is taken. Suggest a peephole optimization pattern that allows the instruction preceding a conditional branch to be moved after it as long as the swapped instruction does not affect the register tested by the conditional branch.

28. Many architectures include a load negative instruction that loads the negation of a value into a register. That is, the value, while being loaded, is subtracted from zero, with the difference stored into the register. Suggest a variety of instruction-level peephole optimization patterns that can make use of a load negative instruction.
29. After a peephole optimization is preformed, the optimized instruction that is substituted for the original instructions is reconsidered for further peephole optimizations. Give examples of cases in which peephole optimizations may be profitably cascaded.
30. Assume we have a peephole optimizer that has n replacement patterns. The most obvious approach to implementing such an optimizer is to try each pattern in turn, leading to an optimizer whose speed is proportional to n.

    Suggest an alternative implementation, based on hashing, that is largely independent of n. That is, the number of patterns considered may be doubled without automatically doubling the optimizer's execution time.