Individual Exercises - Lecture 18

1. Read the following additional references.
   Can be found here:
   http://www.itu.dk/~sestoft/papers/numericperformance.pdf

2. Fischer et. al. exercises 1 (you may want to read Seftoft's note first), 6, 9 (sketch rather than implement - NOTE CG should be TreeCG), 10, 19, 21 538- 546 on pages (exercises 2 (you may want to read Seftoft's note first), 6, 8 (sketch rather than implement), 10, 20, 21 on pages 570-578 in GE)

   1. Consider the following Java method:

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

   Using your favorite Java compiler, show the JVM bytecodes that would be generated for this method. Explain what each of the generated bytecodes contributes to the execution of the methods.

   If the "public static" prefix is removed, the Java method becomes a valid C or C++ function. Compile it using your favorite compiler your favorite processor using no optimization. List the machine instructions generated and show which machine instructions correspond to each generated JVM bytecode.
   The bytecodes generated by the Java compiler supplied as part of the Java Development Kit, JDK (version 1.6.0:20) are:

```
0:   iload_0
1:   ifne    6
4:   iconst_1
5:   ireturn
6:   iload_0
7:   iload_0
8:   iconst_1
9:   isub
10:  invokestatic    #2; //Method fact:(I)I
13:  imul
14:  ireturn
```

The integers in the left column are byte offsets in the method (used as target addresses for branch instructions). The iload at offset 0 pushes the contents of local variable 0 (n) onto the operand stack. If it is not equal to 0 (ifne), control is transferred to offset 6. Otherwise, 1 is pushed onto the operand stack, and an integer return (ireturn) is executed.

If n was not equal to 0 execution continues at offset 6. n is pushed twice, followed by the integer 1. The isub at offset 9 computes n-1. The invokestatic calls the method whose signature is at offset 2 in the constant pool. This is the string fact:(I)I which represents a method named fact that takes an integer parameter and returns an integer result.

The return value from the call is left at the top of the operand stack. The value of n is below it. The imul at offset 13 computes n*fact(n-1). The ireturn at offset 14 returns this value to the caller.

The MIPS code generated by gcc version 3.4.4 is:

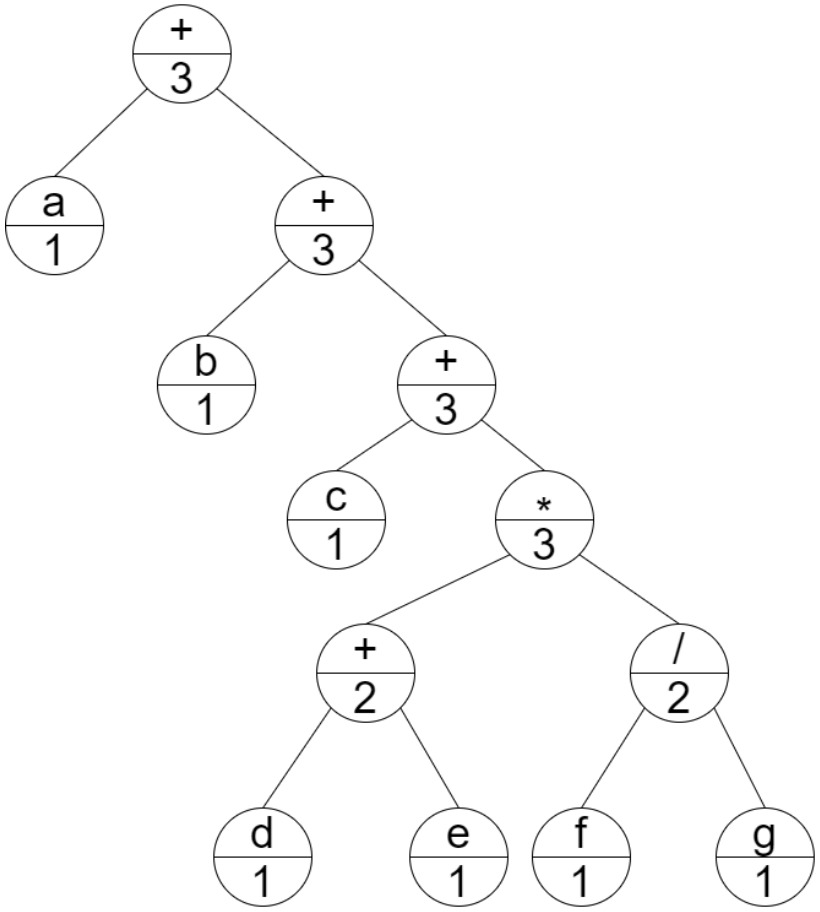
```
fact:
        addiu   $sp,$sp,-32
        sw      $31,28($sp)
        sw      $fp,24($sp)
        move    $fp,$sp
        sw      $4,32($fp)
        lw      $2,32($fp)
        nop
        bne     $2,$0,$L2
        nop

        li      $2,1
        sw      $2,16($fp)
        j       $L1
        nop

$L2:
        lw      $2,32($fp)
        nop
        addiu   $2,$2,-1
        move    $4,$2
        jal     fact
        nop

        lw      $3,32($fp)
        nop
        mult    $2,$3
        mflo    $2
        sw      $2,16($fp)
$L1:
        lw      $2,16($fp)
        move    $sp,$fp
        lw      $31,28($sp)
        lw      $fp,24($sp)
        addiu   $sp,$sp,32
        j       $31
        nop
```

The initial instructions, from addiu to the store of register 4 (the parameter register) set up and link fact's frame. Hence they are actually part of the invokestatic code executed by fact's caller. The load into register 2 corresponds to the bytecode at offset 0. The bne corresponds to the ifne at offset 1. The li (load immediate) of 1 into

register 2 corresponds to the iconst 1 at offset 4. The sw of register 2 into the frame, followed by the jump to $L1 and the code at $L1 implement the ireturn instruction at offset 5.

At $L2 the else part in fact is implemented. The lw loads n into register 2. The addiu computes n-1 into register 2, which is then moved into register 4 (the parameter register). The jal is a call instruction, corresponding to the invokestatic at offset 10. Next, n is loaded into register 3 (corresponding to the iload 0 at offset 6). The mult, mflo pair compute n*fact(n-1), corresponding to the imul at offset 13. Finally, the sw and the code at $L1 implement the ireturn instruction at offset 14.

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.

```
a+(b+(c+((d+e)*(f/g))))
```

```
lw $10, f          # Load f into register 10
lw $11, g          # Load g into register 11
div $10, $10, $11  # Compute f / g into register 10
lw $11, d          # Load d into register 11
lw $12, e          # Load e into register 12
add $11, $11, $12  # Compute d + e into register 11
mul $10, $10, $11  # Compute (d + e) * (f / g) into register 10
lw $11, c          # Load c into register 11
add $10, $10, $11  # Compute c + ... into register 10
lw $11, b          # Load b into register 11
add $10, $10, $11  # Compute b + ... into register 10
lw $11, a          # Load a into register 11
add $10, $10, $11  # Compute a + ... into register 10
```

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:

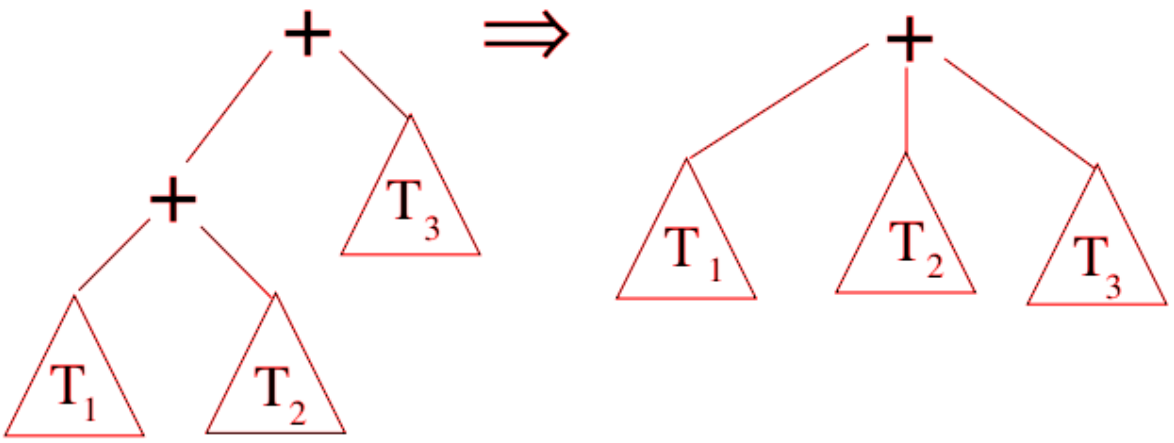
```
(a+b) * (c+d) * ((e+f) / (g-h))
```

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 operations to reduce register needs. (Hint: Allow associative operators to have more than two operands.)

We first restructure the expression tree so that all operands of an associative operator are visible. To do this, if an associative operator, op, has one or more

subtrees also rooted by op, we move the children of the subtrees rooted by op up to the level of the parent operator. This is illustrated below:



Now we must handle the case in which an operator has more than 2 children. Assume the greatest number of registers needed by any subtree is n. Obviously we will need at least n registers to evaluate the operator and its children. Moreover, we will never need more than n + 1 registers to do the evaluation. We can choose any subtree and evaluate it, using no more than n registers. Its result can be held in an additional register. The next subtree is evaluated, using no more than n registers. It is combined, with the previous result, reusing that result's register. This process is continued until all subtrees are evaluated and combined.

How do we decide if n or n + 1 registers are required for a particular expression tree? The key is to select the two subtrees that require the greatest number of registers to evaluate. Say these values are n1 and n2, where n1 ≥ n2. If n1 > n2 then only n1 registers are needed. We evaluate the subtree needing n1 registers first, leaving its results in a single register. The remaining n1 − 1 registers are sufficient to evaluate all remaining subtrees, with partial results always held in the same result register.

If n1 = n2 then n1 +1 registers are needed. One or the other of the two most expensive subtrees must be evaluated first. Its result, perhaps combined with other subexpression values, must be held in a register. Thus when the other of the two most expensive subtrees is evaluated n1 + 1 registers will be needed.

In our example, after restructuring the root operator * will have three subtrees. ((e+f)/(g-h)) requires 3 registers, while (a+b) and (c+d) each require 2 registers. Thus

n1 = 3 and n2 = 2, so we conclude 3 registers suffice if the associativity of * is exploited.

10. In Section 13.4 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 13.2 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.)

Example Expression: a + (b + c)

```
lw $10, c          # Load c into register 10

lw $11, b          # Load b into register 11

lw $12, a          # Load a into register 12

add $10, $10, $11   # Compute b + c into register 10

add $10, $10, $12   # Compute a + (b + c) into register 10
```

By using the extra register to preemptively load another value into a register that is needed in the computation directly following the current computation we ensure that the value is fully loaded into the register instead of having to wait for the load to finish between the two add instructions.

For a more detailed solution see page 278 to 286 in:
http://pages.cs.wisc.edu/~fischer/cs701.f14/lectures/L.All.pdf

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 sometimes have significant delays (5 or more cycles). If a loop body is small, code scheduling cannot do much; there are not 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 be 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?

Loop unrolling has much in common with **software pipelining**, which aims to schedule several loop iterations simultaneously. A popular variant of software pipelining first schedules only one iteration. If all pipeline stalls can't be covered, a second iteration is included, etc. As an upper limit to how many iterations to unroll, we can use the maximum delay possible for any instruction being scheduled. Thus if a multiply has a delay of 5 cycles, scheduling 5 iterations will certainly allow at least the first iteration's multiply to be fully covered. Note though that iterations near the bottom of the loop may still see some delays. That is, if we unroll 5 iterations of a loop, long delay instructions in the last of the 5 loop copies are helped least. They still can be moved upward, so some benefit can be expected.

In practice a limiting factor in unrolling is the availability of registers. If we schedule i iterations in an expanded loop body, we may see i copies of a long delay instruction (like a multiply or divide) executing concurrently. Each will require a different result register. In fact adding iterations until all delays are covered or until available registers run out is a reasonable approach when unrolling a loop.

21. Assume we extend the IR tree patterns defined in Figure 13.27 with the patterns for the MIPS add and load-immediate instructions shown in Figure 13.35.

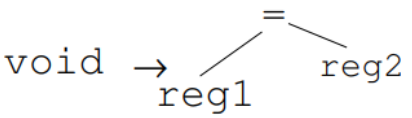
Show how the IR tree of Figure 13.36, corresponding to:

```
A[i+1] = (1+i)*1000;
```

Would be matched. What MIPS instructions would be generated?

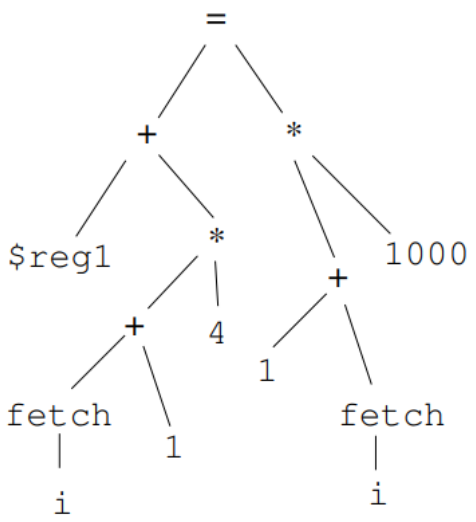
In the first printing of Crafting a Compiler the instruction patterns listed in Figure 13.35 on page 544 are incomplete. Patterns for loading an address and storing the contents of a register into a memory location addressed by a register are needed:

```
reg → adr

li $reg,adr
```




```
sw $reg2,0($reg1)
```

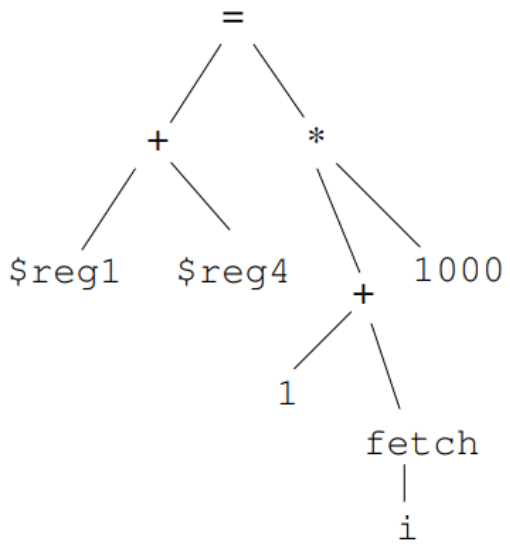
These patterns will be added to the figure in subsequent printings. The IR tree is matched bottom-up, so that code for operands is generated before the code for operators. In this example, register allocation is very simple—each time a register is assigned, a new register is allocated. A more realistic register assignment strategy is discussed in the next exercise. Our traversal will be bottom-up, left to right. So first the address of A is loaded into a register using a load immediate instruction. The IR tree is now:



Generated code:

```
li  $reg1,A
```

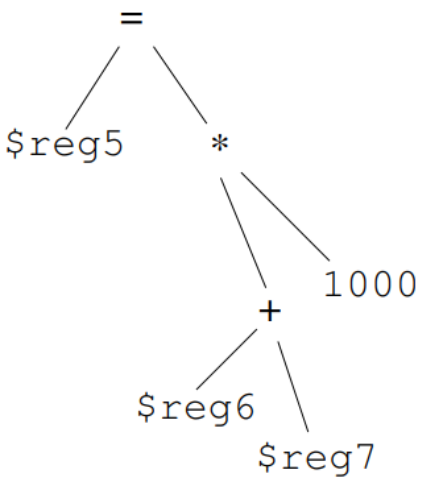
Next, the value held in location i is loaded into a register. One is added using a load immediate, and then i+1 is multiplied by 4 (using a shift left instruction) to compute the byte offset of i+1 in array A:



Generated code:

```
li   $reg1,A
lw   $reg2,i
addi   %reg3,$reg2,1
sll $reg4,$reg3,2
```

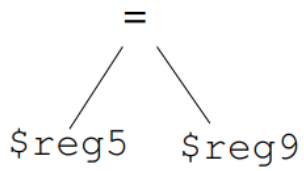
Adding $reg1 and $reg4 yields the address of A[i+1]. In translating 1+i we can't use an add immediate instruction because the pattern only allows an integer literal as the right operand. A richer set of instruction patterns might allow an integer literal as the left operand (since addition is commutative). We therefore load 1 in one register and load the value at location i into a second register:



Generated code:

```
li   $reg1,A
lw   $reg2,i
addi   %reg3,$reg2,1
sll $reg4,$reg3,2
add   %reg5,$reg1,$reg4
li   $reg6,1
lw   $reg7,i
```

We add registers $reg6 and $reg7 together to obtain 1+i, then multiply by 1000:



Generated code:

```
li   $reg1,A
lw   $reg2,i
addi   %reg3,$reg2,1
sll $reg4,$reg3,2
add   %reg5,$reg1,$reg4
li   $reg6,1
lw   $reg7,i
add   %reg8,$reg6,$reg7
mul   %reg9,$reg8,1000
```

Finally, we store the value of (1+i)*1000, held in $reg9, into the memory location A[i+1], whose address is in $reg5:

```
void
```

Generated code:

```
li   $reg1,A
lw   $reg2,i
addi   %reg3,$reg2,1
sll $reg4,$reg3,2
add   %reg5,$reg1,$reg4
li   $reg6,1
lw   $reg7,i
add   %reg8,$reg6,$reg7
mul   %reg9,$reg8,1000
sw   %reg9,0($reg5)
```

Group Exercises - Lecture 18

1. Discuss the outcome of the individual exercises
   Did you all agree on the answers?

2. Fischer et. al. exercises 5, 4, 15 (Note that GCR... skould be GCRegAlloc), 26, 28, 30 on pages 538-546 (exercises 4, 5, 14, 25, 29, 30 on pages 570-578 in GE)

   5. Array bounds checks are mandatory in Java and C#. They are very useful in catching errors, but are also fairly expensive, especially in loops. It is often the case that conditional branches provide information useful in optimizing or even eliminating unnecessary bounds checks. For example, in:

   ```
   while (i < 10) {
      print(a[i++]); }
   ```

   We know i must be less than 10 whenever array a is indexed. Moreover, since i is never decreased in the loop, a single check that i is non-negative at loop entrance suffices.

   Suggest ways in which information provided by conditional branches (in conditionals and loops) can be exploited when code to index arrays is generated.

   This analysis can be implemented as a variant of constant propagation (Section 14.6 on page 623). Rather than track variables known to hold a constant value, we track the range of values a variable may hold. Thus in our example, at the start of each iteration, we know i must be in the range [-∞..9] (-∞ and ∞ represent the minimum and maximum possible integer values). Range information must be updated across assignments. Thus after i++ we conclude i must be in the range [(-∞+1)..10].

   At the end of conditionals and loops, range information must be merged. Thus if we know j is in the range [0..10] at the end of the then part of an if, and j is in the range [-5..5] at the end of the else part, then j is in the range [-5..10] after the if statement.

   In Java the size of an array is part of its value, but it is very often the case that an array variable is allocated only once, with an easily identifiable size. Note too that conditional constant propagation, in which propagated constant values can be used to evaluate and simplify boolean expressions, can be generalized to range analysis. Thus if we know k is in the range [0..10], then an expression like (k<0) can be simplified to false.

4. A common subprogram optimization is *inlining*. At the point of call, the body of the called method is substituted for the call, with actual parameter values used to initialize local variables that represent formal parameters.

Assume we have the bytecodes that represent the body of subprogram P that is marked private or final (and hence cannot 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 do not "clash" with other bytecodes in the context of call?

All references to the n parameters in P must be updated so P references the arguments given at the call site. Also, care must be taken to rename the m local variables so they do not conflict with the parameters and local variables currently in scope at the call site.

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?

The key is getting reasonable estimates of how often each basic block in a subprogram will execute. Profiling is commonly used. An important issue is how closely test data mimics actual user data. Lacking dynamic data provided by profiling, static estimates may be used. We already scale spill cost estimates by a factor of 10 per loop nesting level. With a bit more effort actual loop counts can be extracted for the common case of constant loop bounds. We reduce execution counts in conditionals, perhaps assuming each leg of an if has a 50% chance of executing.

With estimates of execution frequencies, we can compute added execution time whenever GCRegAlloc or PriorityRegAlloc elects to do a spill. We may total added instruction counts or perhaps focus on added memory reads (since cache misses and page faults can be very costly).

Given accurate spill costs estimates, we can now ask a key question — how much benefit do we obtain by giving a subprogram an extra register. Interprocedural register allocators ask this question repeatedly, deciding which of a number of subprograms can best use an additional register.

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.

In the first printing of Crafting a Compiler this exercise incorrectly suggested using a dup instruction instead of a dup x2. This error will be corrected in subsequent printings.

This optimization is akin to a machine-level optimization that removes a register load if the desired value is already a register resident. We will aim to make a copy of the value assigned to a[i] and use that value in the second assignment, avoiding a reload of a[i]. The patterns we use will depend on where a and i are stored. We'll cover the common case that both are local variables within a method. Related patterns will be needed in the cases that a or i are fields.

We must also consider the possibility that the instructions on the right hand side of the first assignment involve assignment or transfer of control. (We don't want i changed while evaluating the right-hand side.) Call an IR instruction that is not a label and does not transfer control or potentially change a local variable to be "pure." How many IR instructions should we expect in the right-hand side of the first assignment? We can write a number of related rules, each for a particular IR instruction count.

Alternatively, we can allow a pattern to match a range of instructions, with a fixed upper limit (so that pattern matching doesn't waste time trying to match unrealistically large right-hand sides). Let the pattern pure[1:n] match from 1 to n pure instructions. The pattern listed below will look for an array store (*iastore*) immediately followed by an array load (*iaload*) of the same element in the same array. If this is found, a dup x2 instruction duplicates the value being stored and places it below the three operands to the array store instruction. This makes the array load instruction unnecessary, as the desired value will be found at the top of the operand stack:

```
aload Ar
iload Ind
pure[1:n]
iastore
aload Ar
iload Ind
iaload
```

⇒

```
aload Ar
iload Ind
pure[1:n]
dup_x2
iastore
```

28. Many architectures include *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 two instruction-level peephole optimization patterns that can make use of a load-negative instruction.

The obvious pattern corresponds to ...=-b which will load b then negate it. A load negative of b is a clear improvement. For uniformity, we'll use the MIPS instruction set and we'll assume *lwn* is a load word negative:

```
lw $reg,loc
negu $reg,$reg
```

⇒

```
lwn $reg,loc
```

A less obvious (and probably less common) replacement involves an expression like ...=-(a+b). This will generate two loads, an add, and a negation.

But this expression is equivalent to ...=(-a)-b which can be implemented using a load negative, a load and a subtraction:

```
lw $reg1,loc1
lw $reg2,loc2
add $reg1,$reg1,$reg2
negu $reg1,$reg1
```

⇒

```
lwn $reg1,loc1
lw $reg2,loc2
sub $reg1,$reg1,$reg2
```

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.

The basic idea is to represent patterns in a way that allows us to hash them. We can then construct a map that will map those pattern hashes to an optimization.

For example: optimizationMap["constant + constant"]=constantFoldingOptmization. We then traverse the tree and plug the node combinations we find into the map. If a node combination maps to an optimization, we apply it.