A couple of final issues must be dealt with:

- Does the iteration count need to be changed?
  In this case no, since the final valid value of `i`, 999, is used to compute `%g4` in cycle 5, before the loop exits.
- What instructions do we keep as the loop's epilogue?
  None! Instructions past the kernel aren't needed since they are part of future iterations (past `i==999`)which aren't needed or wanted.
- Note that `b[1000]` and `b[1001]` are "touched" even though they are never used. This is probably OK as long as arrays aren't placed at the very end of a page or segment.

Our final loop is:

```
cycle       instruction
1.          ld    [%o1], %g2      !N0
1.          add   %o1, 4, %o1     !N0
3.          add   %g3, %g2, %g4   !N0
3.          ld    [%o1], %g2      !N1
3.          add   %o1, 4, %o1     !N1
3.          add   %g3, 1, %g3     !N0
4.     L:   st    %g4, [%o0]      !N0
4.          add   %o0, 4, %o0     !N0
4.          cmp   %g3, 999        !N0
5.          add   %g3, %g2, %g4   !N1
5.          ld    [%o1], %g2      !N2
5.          add   %o1, 4, %o1     !N2
5.          ble   L               !N0
5.          add   %g3, 1, %g3     !N1
```

This is very efficient code—we use the full parallelism of the processor, executing 5 instructions in cycle 5 and 8 instructions in just 2 cycles. All resource limitations are respected.

False Dependencies & Loop Unrolling

A limiting factor in how "tightly" we can software pipeline a loop is reuse of registers and the false dependencies reuse induces.

Consider the following simple function that copies array elements:

```
void f (int a[],int b[], int lim) {
  for (i=0;i<lim;i++)
     a[i]=b[i];
}
```

The loop that is generated takes 3 cycles:

```
cycle       instruction
1.     L:    ld    [%g3+%o1], %g2
1.           addcc %o2, -1, %o2
3.           st    %g2, [%g3+%o0]
3.           bne   L
3.           add   %g3, 4, %g3
```


We'd like to tighten the iteration interval to 2 or less. One cycle is unlikely, since doing a load and a store in the same cycle is problematic (due to a possible dependence through memory).

If we try to use modulo scheduling, we can't put a second copy of the load in cycle 2 because it would overwrite the contents of the first load. A load in cycle 3 will clash with the store.

The solution is to unroll the loop into two copies, using different registers to hold the contents of the load and the current offset into the arrays.

The use of a "count down" register to test for loop termination is

helpful, since it allows an easy exit from the middle of the loop.

With the renaming of the registers used in the two expanded iterations, scheduling to "tighten" the loop is effective.

After expansion we have:

```
cycle       instruction
1.     L:    ld    [%g3+%o1], %g2
1.           addcc %o2, -1, %o2
3.           st    %g2, [%g3+%o0]
3.           beq   L2
3.           add   %g3, 4, %g4
4.           ld    [%g4+%o1], %g5
4.           addcc %o2, -1, %o2
6.           st    %g5, [%g4+%o0]
6.           bne   L
6.           add   %g4, 4, %g3
       L2:
```

We still have 3 cycles per iteration, because we haven't scheduled yet.

Now we can move the increment of `%g3` (into `%g4`) above other uses of `%g3`. Moreover, we can move the load into `%g5` *above* the store from `%g2` (*if the load and store are independent*):

```
cycle       instruction
1.     L:    ld    [%g3+%o1], %g2
1.           addcc %o2, -1, %o2
1.           add   %g3, 4, %g4
2.           ld    [%g4+%o1], %g5
3.           st    %g2, [%g3+%o0]
3.           beq   L2
3.           addcc %o2, -1, %o2
4.           st    %g5, [%g4+%o0]
4.           bne   L
4.           add   %g4, 4, %g3
       L2:
```

We can normally test whether `%g4+%o1` and `%g3+%o0` can be equal at compile-time, by looking at the actual array parameters.
(Can `&a[0] == &b[1]`?)

Predication

We have seen that conditional execution complicates code scheduling by creating small basic blocks and limiting code movement across conditional branches.

However, the problems conditionals introduce are even more fundamental.

Consider the following code fragment:

```
if (a<b)
     a++;
else b++;
if (c<d)
     c++;
else d++;
```


The two conditionals are completely independent, but they can't be evaluated concurrently in a single thread.

Why?

Look at the Sparc code generated:

```
  cmp     %o0, %g1
  bge,a   L1
  add     %g1, 1, %g1
  add     %o0, 1, %o0
L1:
  cmp     %o5, %o4
  bge,a   L2
  add     %o4, 1, %o4
  add     %o5, 1, %o5
L2:
```

The two compares can't be executed concurrently (because there is only one condition code register).

We can't do two conditional branches to two different places simultaneously.

And we must select the correct combination of two of the four adds to execute.

We could restructure this code into a four-way switch, but this far beyond what a code scheduler is expected to do.

The problem is that while values can easily be computed in parallel, flow of control can't.

The solution?

Convert flow of control computations into value computations.

Our first step is to generalize a single condition code register into a set of predicate registers. The Itanium, for example, includes 64 predicate registers that hold a single boolean value. For our purposes, let's denote a predicate register as `%p0` to `%p63`.

Predicate registers are set by doing compare or test instructions.

Thus

```
cmpeq  %o0, %g1,%p1
```

sets `%p1` true if the two operands are equal and false otherwise.

The real power of predication is that most instructions can be controlled (predicated) by a predicate register.

Thus

```
add(%p1) %r1,%r2,%r3
```

does an ordinary add but only commits the result (into `%r3`) if `%p1` is true.

A negated form is often included too:

```
add(~%p1) %r1,%r2,%r3
```

In this form, the add is completed only if `%p1` is false.

Using predication, we can eliminate many conditional branches. Now both legs of a conditional can be evaluated, with only one leg allowed to commit.
Returning to our earlier example,


```
if (a<b)
     a++;
else b++;
if (c<d)
     c++;
else d++;
```

we now generate

```
1. cmplt      %o0, %g1, %p1
1. cmplt      %o5, %o4, %p2
2. add(%p1)   %g1, 1, %g1
2. add(~p1)   %o0, 1, %o0
2. add(%p2)   %o4, 1, %o4
2. add(%~p2)  %o5, 1, %o5
```

This entire code fragment can now execute in two cycles, since the two compares and four adds are independent of each other.

Predication Enhances Software Pipelining

Conditionals in a loop body greatly complicate software pipelining since we usually won't know exactly what instructions future iterations will execute.

Consider this minor variant of our earlier example:

```
void f (int a[],int b[]) {
  t1 = &a[0];
  t2 = &b[0];
  for (i=0;i<1000;i++,t1++,t2++)
     if (i%2)
           *t1 = *t2 + i;
     else  *t1 = *t2 - i;

}
```



```
1.   f:   mov     0, %g3
2.   L:   andcc   %g3, 1, %g0
3.        bne     L1
4.        ld      [%o1], %g2
5.        b       L2
6.        sub     %g3, %g2, %g4
7.  L1:   add     %g3, %g2, %g4
8.  L2:   st      %g4, [%o0]
9.        add     %g3, 1, %g3
10.       add     %o0, 4, %o0
11.       cmp     %g3, 999
12.       ble     L
13.       add     %o1, 4, %o1
14.       retl
15.       nop
```

We've added an `andcc` (to do the `i%2` computation) as well as a conditional and unconditional branch. Each iteration will do an add or a subtract.

A two cycle per iteration schedule seems most unlikely.

But predication helps immensely!

The generated code becomes much cleaner:

```
1.   f:   mov        0, %g3
2.   L:   and        %g3, 1, %p1
3.        ld         [%o1], %g2
4.        sub(~%p1)  %g3, %g2, %g4
5.        add(%p1)   %g3, %g2, %g4
6.        st         %g4, [%o0]
7.        add        %g3, 1, %g3
8.        add        %o0, 4, %o0
9.        cmp        %g3, 999
10.       ble        L
11.       add        %o1, 4, %o1
12.       retl
13.       nop
```

And guess what? We can still software pipeline this into 2 cycles per iteration:


```
cycle     instruction
1.         ld     [%o1], %g2
1.         add    %o1, 4, %o1
2.          and   %g3, 1, %p1
3.         add(%p1)  %g3, %g2, %g4
3.         sub(~%p1) %g3, %g2, %g4
3.         ld     [%o1], %g2
3.         add    %o1, 4, %o1
3.         add    %g3, 1, %g3
4.      L: st     %g4, [%o0]
4.         add    %o0, 4, %o0
4.          and   %g3, 1, %p1
4.         cmp    %g3, 999
5.         add(%p1)  %g3, %g2, %g4
5.         sub(~%p1) %g3, %g2, %g4
5.         ld     [%o1], %g2
5.         add    %o1, 4, %o1
5.         ble    L
5.         add    %g3, 1, %g3
```

We now do need to be able to issue four ALU operations per cycle (since we issue both the add and subtract in the same cycle).

Reading Assignment

- **Read Section 13.5 (Automatic Instruction Selection) of *Crafting a Compiler.***

Automatic Instruction Selection

Besides register allocation and code scheduling, a code generator must also do *Instruction Selection*.

For CISC (Complex Instruction Set Computer) Architectures, like the Intel x86, DEC Vax, and many special purpose processors (like Digital Signal Processors), instruction selection is often *challenging* **because so many choices exist.**

In the Vax, for example, one, two and three address instructions exist. Each address may be a register, memory location (with or without indexing), or an immediate operand.

For RISC (Reduced Instruction Set Computer) Processors, instruction formats and addressing modes are far more limited.

Still, it is necessary to handle immediate operands, commutative operands and special case null operands (add of 0 or multiply of 1).

Moreover, automatic instruction selection supports *automatic retargeting* **of a compiler to a new or extended instruction set.**

Tree-Structured Intermediate Representations

For purposes of automatic code generation, it is convenient to translate a source program into a *Low-level, Tree-Structured IR*.

This representation exposes translation details (how locals are accessed, how conditionals are translated, etc.) without assuming a particular instruction set.

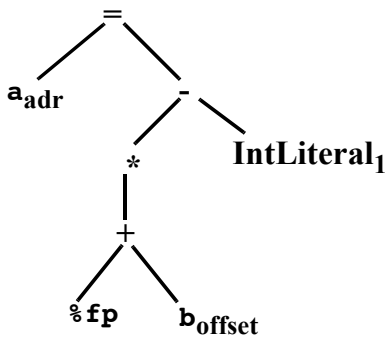
In a low-level, tree-structured IR, leaves are registers or bit-patterns and internal nodes are machine-level primitives, like load, store, add, etc.

Example

Let's look at how

```
a = b - 1;
```

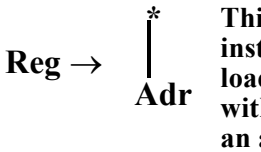
is represented, where `a` is a global integer variable and `b` is a local (frame allocated) integer variable.



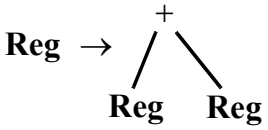
Representation of Instructions

Individual instructions can be represented as trees, rooted by the operation they implement.

For example:

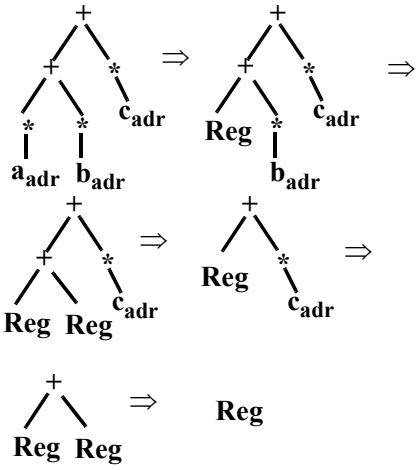


This is an instruction that loads a register with the value at an absolute address.



This is an instruction that adds the contents of two registers and stores the sum into a third register.

Using the above pair of instruction definitions, we can repeatedly match instructions in the following program IR:



Each match of an instruction pattern can have the side-effect of generating an instruction:

```
ld   [a],%R1
ld   [b],%R2
add  %R1,%R2,%R3
ld   [c],%R4
add  %R3,%R4,%R5
```

Registers can be allocated on-the-fly as Instructions are generated or instructions can be generated using pseudo-registers, with a subsequent register allocation phase.

Using this view of instruction selection, choosing instructions involves finding a *cover* for an IR tree using Instruction Patterns.

Any cover is a *valid* translation.

Tree Parsing vs. String Parsing

This process of selecting instructions by matching instruction patterns is very similar to how strings are parsed using Context-free Grammars.

We repeatedly identify a sub-tree that corresponds to an instruction, and simplify the IR-tree by replacing the instruction sub-tree with a nonterminal symbol. The process is repeated until the IR-tree is reduced to a single nonterminal.

The theory of reducing an IR-tree using rewrite rules has been studied as part of BURS (Bottom-Up Rewrite Systems) Theory by Pelegri-Llopart and Graham.

Automatic Instruction Selection Tools

Just as tools like Yacc and Bison automatically generate a string parser from a specification of a Context-free Grammar, there exist tools that will automatically generate a tree-parser from a specification of tree productions.

Two such tools are BURG (Bottom Up Rewrite Generator) and IBURG (Interpreted BURG). Both automatically generate parsers for tree grammars using BURS theory.

Least-Cost Tree Parsing

BURG (and IBURG) *guarantee* to find a cover for an input tree (if one exists).

But tree grammars are usually *very* ambiguous.

Why?—Because there is usually more than one code sequence that can correctly implement a given IR-tree.

To deal with ambiguity, BURG and IBURG allow each instruction pattern (tree production) to have a *cost*.

This cost is typically the size or execution time for the corresponding target-machine instructions.

Using costs, BURG (and IBURG) not only guarantee to find a cover, but also a *least-cost cover.*

This means that when a generated tree-parser is used to cover (and thereby translate) an IR-Tree, the *best possible* code sequence is guaranteed.

If more than one least-cost cover exists, an arbitrary choice is made.

Using BURG to Specify Instruction Selection

We'll need a tree grammar to specify possible partial covers of a tree.

For simplicity, BURG requires that all tree productions be of the form

$A \rightarrow b$

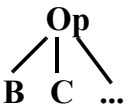
(where b is a single terminal symbol)

or

$A \rightarrow Op(B,C, ...)$

(where Op is a terminal that is a subtree root and B,C, ... are non-terminals)

$A \rightarrow Op(B,C, ...)$ denotes



All tree grammars can be put into this form by adding new nonterminals and productions as needed.

We must specify terminal symbols (leaves and operators in the IR-Tree) and nonterminals that are used in tree productions.

Example

A subset of a SPARC instruction selector.

Terminals

Leaf Nodes

`int32`	(32 bit integer)
`s13`	(13 bit signed integer)
`r`	(0-31, a register name)

Operator Nodes

`*`	(unary indirection)
`-`	(binary minus)
`+`	(binary addition)
`=`	(binary assignment)

Nonterminals

`UInt`	(32 bit unsigned integer)
`Reg`	(Loaded register value)
`Imm`	(Immediate operand)
`Adr`	(Address expression)
`Void`	(Null value)

Productions

Rule #	Production	Cost	SPARC Code
R0	UInt → Int32	0	
R1	Reg → r	0	
R2	Adr → r	0	
R3	Adr → + (Reg, Imm)	0	
R4	Imm → s13	0	
R5	Reg → s13	1	`mov s13,Reg`
R6	Reg → int32	2	`sethi %hi(int32),%g1` `or %g1, %lo(int32),Reg`
R7	Reg → − (Reg, Reg)	1	`sub Reg,Reg,Reg`

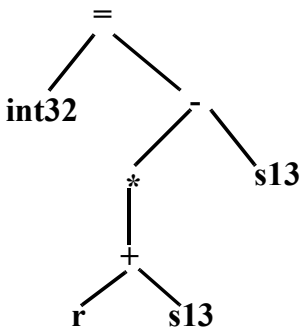
Rule #	Production	Cost	SPARC Code
R8	Reg → − (Reg, Imm)	1	`sub Reg,Imm,Reg`
R9	Reg → * (Adr)	1	`ld [Adr],Reg`
R10	Void → = (UInt, Reg)	2	`sethi %hi(UInt),%g1` `st Reg, [%g1+%lo(Uint)]`

Example

Let's look at instruction selection for

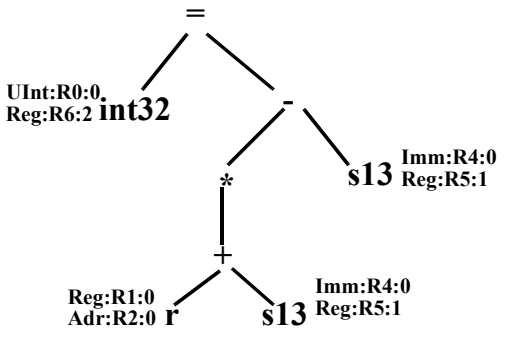
```
a = b - 1;
```

where `a` is a global int, accessed with a 32 bit address and `b` is a local int, accessed as an offset from the frame pointer.



We match tree nodes *bottom-up*. Each node is labeled with the nonterminals it can be reduced to, the production used to produce the nonterminal, and the cost to generate the node (and its children) from the nonterminal.

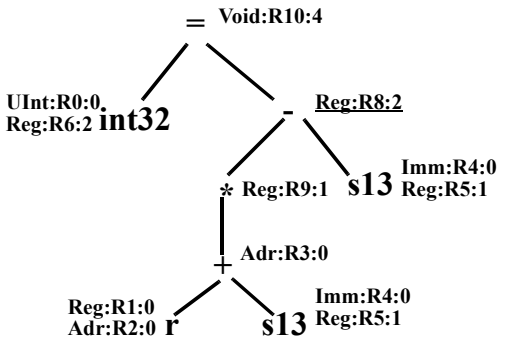
We match leaves first:



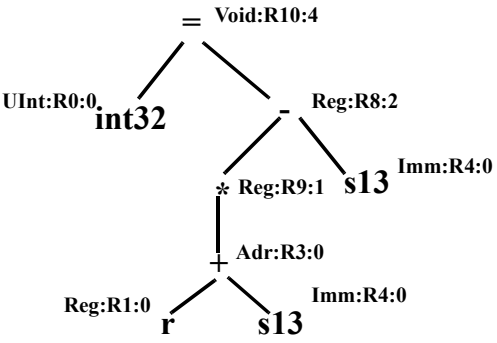
We now work *upward*, considering operators whose children have been labeled. Again, if an operator can be generated by a nonterminal, we mark the operator with the nonterminal, the production used to generate the operator, and the total cost (including the cost to generate all children).

If a nonterminal can generate the operator using more than one production, the *least-cost* derivation is chosen.

When we reach the root, the nonterminal with the lowest overall cost is used to generate the tree.

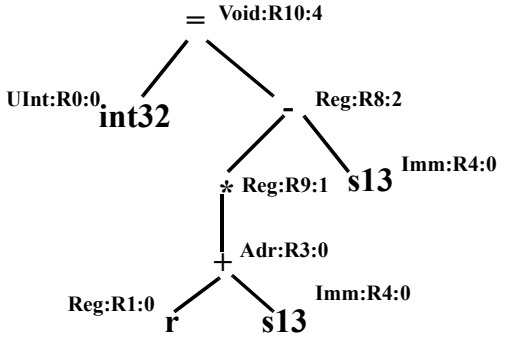


Note that once we know the production used to generate the root of the tree, we know the productions used to generate each subtree too:



We generate code by doing a depth-first traversal, generating code for a production after all the production's children have been processed.

We need to do register allocation too; for our example, a simple on-the-fly generator will suffice.




```
ld    [%fp+b],%l0
sub   %l0,1,%l0
sethi %hi(a),%g1
st    %l0,[%g1+%lo(a)]
```

Had we translated a slightly difference expression,

```
a = b - 1000000;
```

we would *automatically* get a different code sequence (because 1000000 is an int32 rather than an s13):

```
ld    [%fp+b],%l0
sethi %hi(1000000),%g1
or    %g1,%lo(1000000),%l1
sub   %l0,%l1,%l0
sethi %hi(a),%g1
st    %l0,[%g1+%lo(a)]
```


Adding New Rules

Since instruction selectors can be automatically generated, it's easy to add "extra" rules that handle optimizations or special cases.

For example, we might add the following to handle addition of a left immediate operand or subtraction of 0 from a register:

Rule #	Production	Cost	SPARC Code
R11	Reg → + (Imm Reg)	1	`add Reg,Imm,Reg`
R12	Reg → − (Reg Zero)	0	

Improving the Speed of Instruction Selection

As we have presented it, instruction selection looks rather slow—for each node in the IR tree, we must match productions, compare costs, and select least-cost productions.

Since compilers routinely generate program with tens or hundreds of thousands of instructions, doing a lot of computation to select one instruction (even if it's the *best* instruction) could be too slow.

Fortunately, this need not be the case.

Instruction selection using BURS can be made *very* fast.

Adding States to BURG

We can *precompute* a set of *states* that represent possible labelings on IR tree nodes. A table of node names and subtree states then is used to select a node's state. Thus labeling becomes nothing more than repeated table lookup.

For example, we might create a state s0 that corresponds to the labeling {Reg:R1:0, Adr:R2:0}.

A state selection function, *label*, defines label(r) = s0. That is, whenever r is matched as a leaf, it is to be labeled with s0.

If a node is an operator, label uses the name of the operator and the

labeling assigned to its children to choose the operator's label. For example,

label(+,s0,s1)=s2

says that a + with children labeled as s0 and s1 is to be labeled as s2.

In theory, that's all there is to building a fast instruction selector.

We generate possible labelings, encode them as states, and table all combinations of labelings.

But,

how do we know the set of possible labelings is even finite?

In fact, it isn't!

Normalizing Costs

It is possible to generate states that are identical except for their costs.

For example, we might have

s1 = {Reg:R1:0, Adr:R2:0},

s2 = {Reg:R1:1, Adr:R2:1},

s3 = {Reg:R1:2, Adr:R2:2}, etc.

Here an important insight is needed—the *absolute* costs included in states aren't really essential. Rather *relative* costs are what is important. In s1, s2, and s3, Reg and Adr have the same cost. Hence the same decision in choosing between Reg and Adr will be made in all three states.

We can limit the number of states needed by *normalizing* costs within states so that the lowest cost choice is always 0, and other costs are differences (deltas) from the lowest cost choice.

This observation keeps costs bounded within states (except for pathologic cases).

Using additional techniques to further reduce the number of states needed, and the time needed to generate them, fast and compact BURS instruction selectors are achievable. See

"Simple and Efficient BURS Table Generation," T. Proebsting, 1992 PLDI Conference.

Example

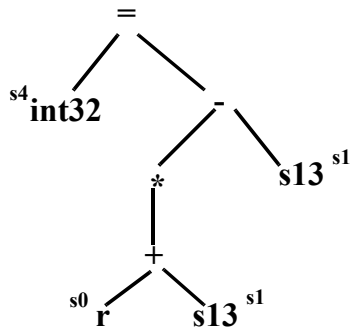
State	Meaning
s0	{Reg:R1:0, Adr:R2:0}
s1	{Imm:R4:0, Reg:R5:1}
s2	{adr:R3:0}
s3	{Reg:R9:0}
s4	{UInt:R0:0}
s5	{Reg:R8:0}
s6	{Void:R10:0}
s7	{Reg:R7:0}

Node	Left Child	Right Child	Result
r			s0
s13			s1
int32			s4
+	s0	s1	s2
*	s2		s3
-	s3	s1	s5
-	s1	s3	s7
=	s4	s5	s6

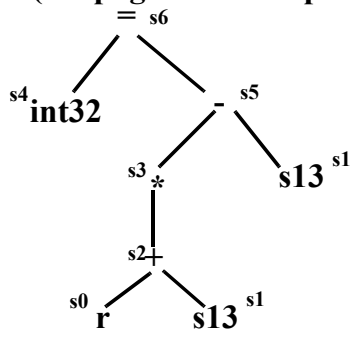
We start by looking up the state assigned to each leaf. We then work upward, choosing the state of a parent based on the parent's kind and the states assigned to the children. These are all table lookups, and hence very fast.

At the root, we select the nonterminal and production based on the state assigned to the root (any entry with 0 cost). Knowing the production used at the root tells us the nonterminal used at each child. Each state has only one entry per nonterminal, so knowing a node's state and the nonterminal used to generate it immediately tells us the production used. Hence identifying the production used for each node is again very fast.

Step 1 (Label leaves with states):



Step 2 (Propagate states upward):



Step 3 (Choose production used at root): R10.

Step 4 (Propagate productions used downward to children):