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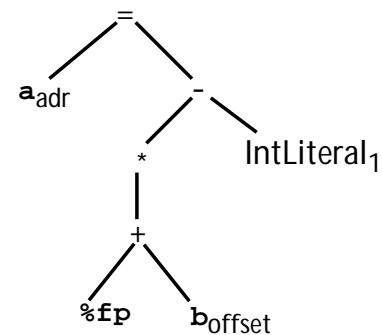
In a low-level, tree-structured IR, leaves are registers or bit-patterns and internal nodes are machine-level primitives, like load, store, add, etc.
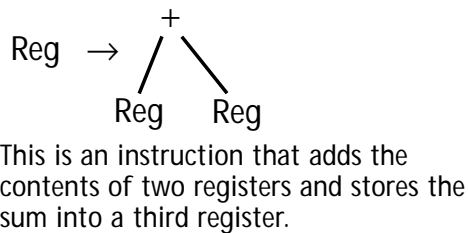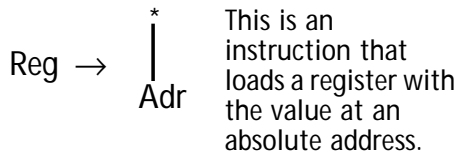
Example

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:

Reg →

*
|
Adr

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

Reg →

$+$
/ \
Reg   Reg

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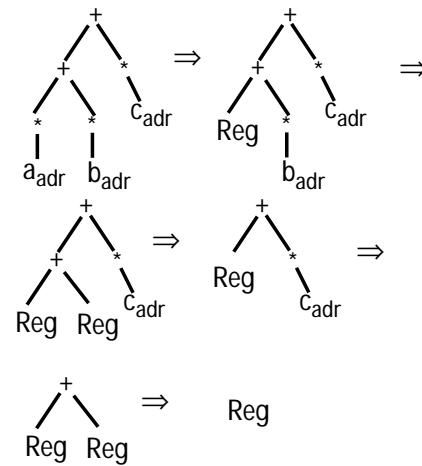
Each match of an instruction pattern can have the side-effect of generating an instruction:

```
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.

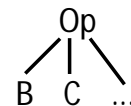For simplicity, BURG requires that all tree productions be of the form

$A \rightarrow b$

(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

```
      Op
     /|  \
    B  C  ...
```

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 → $\overset{-}{\diagup \diagdown}$ Reg Imm	1	`sub Reg,Imm,Reg`
R9	Reg → $\overset{*}{|}$ Adr	1	`ld [Adr],Reg`

R10	Void → $\overset{=}{\diagup \diagdown}$ UInt Reg	2	`sethi`
			`  %hi(UInt),%g1`
			`st Reg,`
			`  [%g1+%lo(Uint)]`

Reading Assignment

- Read "Optimal Spilling for CISC Machines with Few Registers," by Appel and George. (Linked from the class Web page.)

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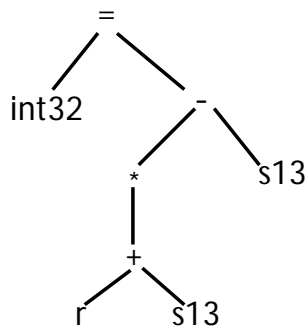
We match tree nodes *bottom-up.* Each node is labeled with the nonterminals it can be reduced to, the production used to produce the nonterminal, and the cost to generate the node (and its children) from the nonterminal.

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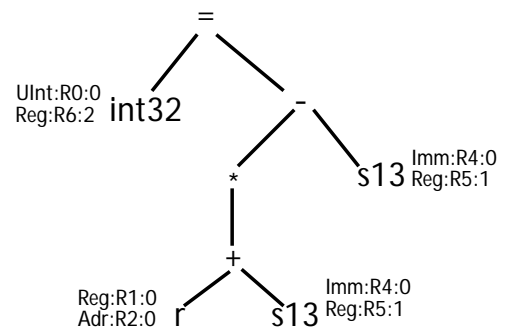
We now work *upward*, considering operators whose children have been labeled. Again, if an operator can be generated by a nonterminal, we mark the operator with the nonterminal, the production used to generate the operator, and the total cost (including the cost to generate all children).

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.
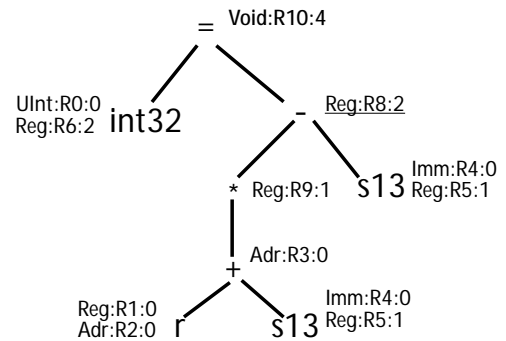
Note that once we know the production used to generate the root of the tree, we know the productions used to generate each subtree too:
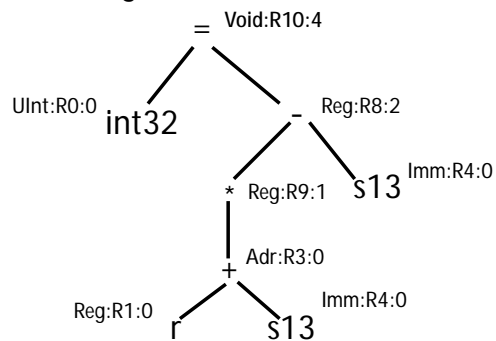
We generate code by doing a depth-first traversal, generating code for a production after all the production's children have been processed.

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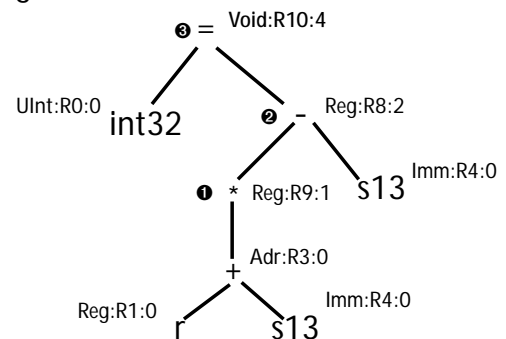

```
❶  ld     [%fp+b],%l0
❷  sub    %l0,1,%l0
❸  sethi  %hi(a),%g1
   st     %l0,[%g1+%lo(a)]
```

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):
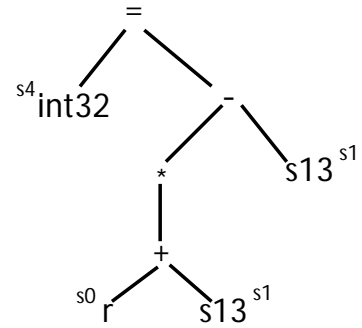
Step 2 (Propagate states upward):
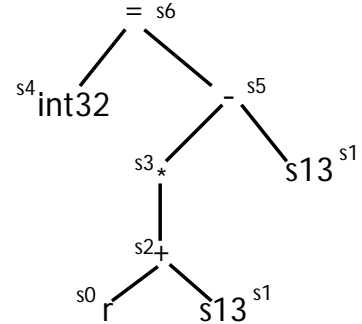
Step 3 (Choose production used at root): R10.

Step 4 (Propagate productions used downward to children):