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]**?)

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.

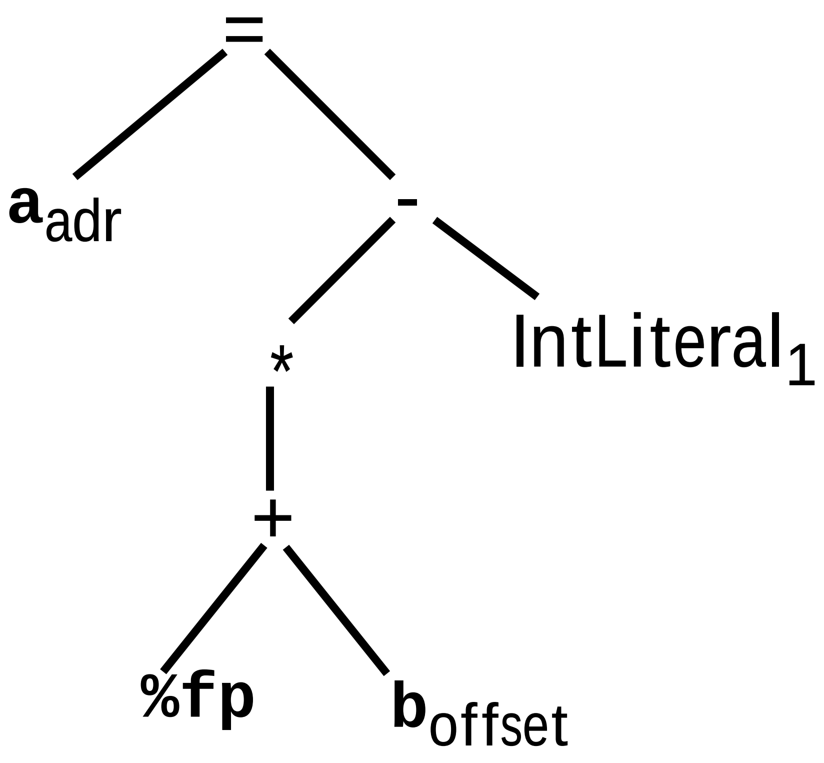
In a low-level, tree-structured IR, leaves are registers or bit-patterns and internal nodes are machine-level primitives, like load, store, add, etc.

Example

Let’s look at how

`a = b - 1`;

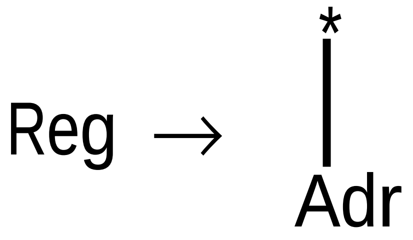
is represented, where **`a`** is a global integer variable and **`b`** is a local (frame allocated) integer variable.



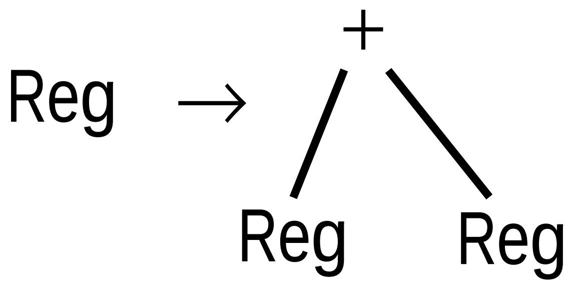
Representation of Instructions

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

For example:

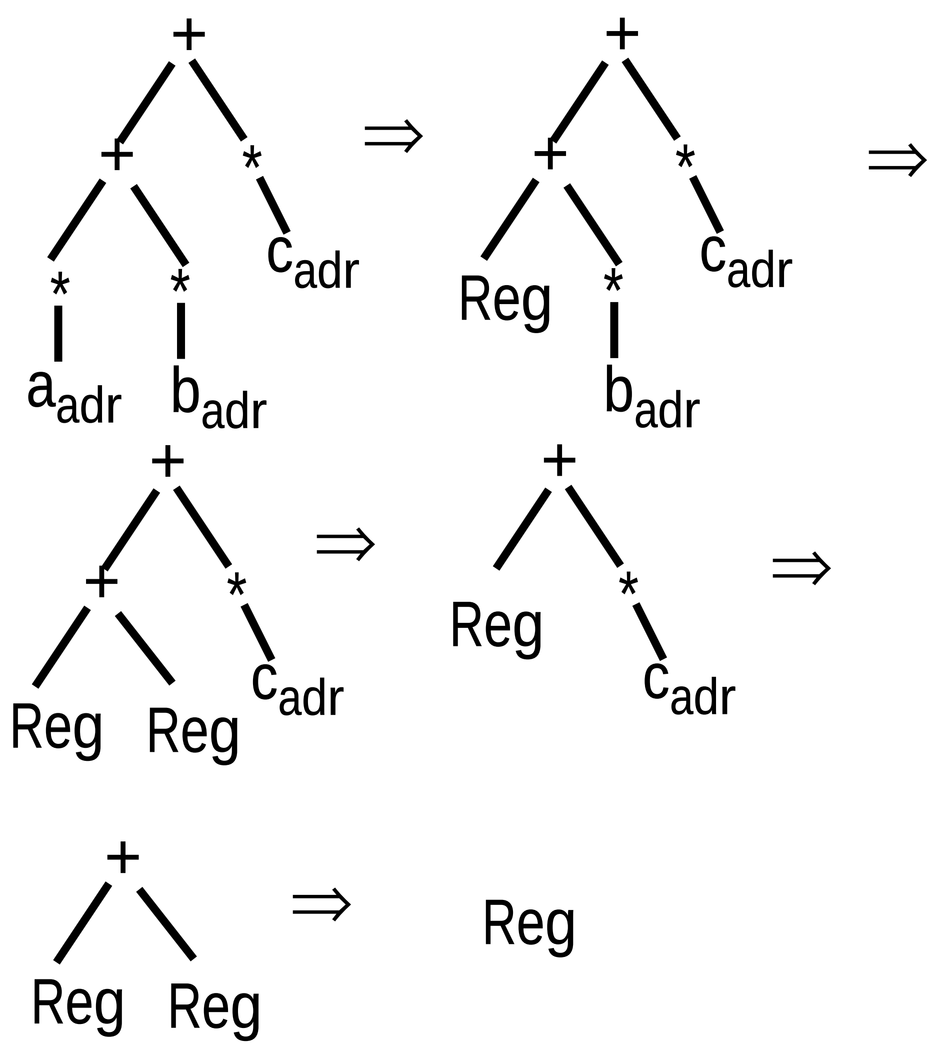


This is an instruction that loads a register with the value at an absolute address.



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

$A \rightarrow b$

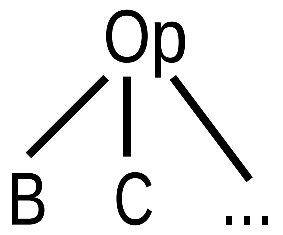
(where b is a single terminal symbol)

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)

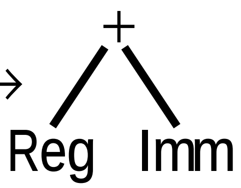
Operator Nodes

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

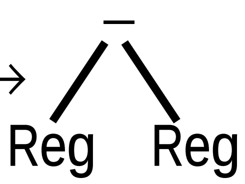
Nonterminals

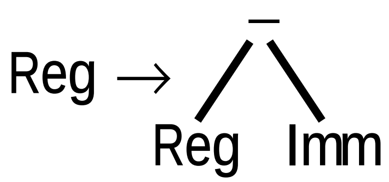
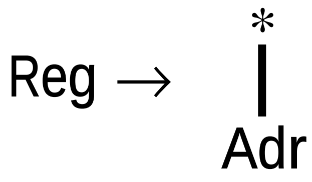
`UInt`	(32 bit unsigned integer)
`Reg`	(Loaded register value)
`Imm`	(Immediate operand)
`Adr`	(Address expression)
`Void`	(Null value)

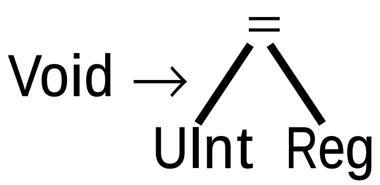
Productions

Rule #	Production	Cost	SPARC Code
R0	UInt → Int32	**0**	
R1	Reg → r	**0**	
R2	Adr → r	**0**	
R3	Adr → 	**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 → 	**1**	**`sub Reg,Reg,Reg`**

Rule #	Production	Cost	SPARC Code
R8		**1**	**`sub Reg,Imm,Reg`**
R9		**1**	**`ld [Adr],Reg`**

R10		**2**	**`sethi`**
			` %hi(UInt),%g1`
			`st Reg,`
			` [%g1+%lo(Uint)]`