# The RTL System

Carl McConnell
J. David Roberts
Charles B. Schoening
Department of Computer Science
University of Illinois at Urbana-Champaign

*revised October 8, 1990*

# Contents

# Part I

# Overall Design and Optimization

# Chapter 1

# Introduction

Object-oriented programming is an excellent way to develop modular and reusable code. A case in point is the Smalltalk-80[1] programming environment [Gol84], which uses the object-oriented language Smalltalk [GR83] to great advantage in the implementation of the class libraries, graphical user interface, and symbolic debugger that are the hallmarks of this powerful environment.

However, Smalltalk has one major shortcoming: its slowness. The fundamental reason for this is that Smalltalk is typeless. As a consequence, the code to be executed for a given procedure invocation must be determined at run-time. This late binding in itself is not the cause of the slowness. Rather, the cause is that late binding prevents any significant optimization: every operation in Smalltalk is a procedure call, and the procedure to be invoked cannot be determined at compile time. Therefore, it is impossible to generate long sequences of code that are uninterrupted by calls to unknown procedures and hence amenable to optimization.

If Smalltalk is to rival traditional programming languages in speed, it is clear that it needs to be implemented differently. The Typed Smalltalk compiler [JGZ88] uses a type system to bind most procedure calls at compile time. It performs a number of optimizations on the resulting program, such as in-line substitution of procedures. It is the in-line substitution of primitives, which are written in the intermediate language described in the next chapter, that accounts for most of the intermediate code fed to the optimizer. In the ideal case, the intermediate code for a program consists of the intermediate code for the primitives it invokes, along with a bit of linkage code.

This thesis describes the source language-independent, retargetable code optimizer in the Typed Smalltalk compiler. It shows how valuable an object-oriented programming language such as Smalltalk can be in creating an extensible framework for performing a given task, in this case code optimization. It also describes the use of a new and elegant representation of the data flow properties of a program, *static single assignment form* [CFR*89].

The current version of the optimizer owes a great deal to an earlier version described in [Wie87]. That version, in turn, was based on work described in [DF80] [Dav81] [DF84] [Dav85].

[1]Smalltalk-80 is a trademark of PARC Place Systems

# Chapter 2

# The Register Transfer Language RTL

The RTL System is based on an intermediate register transfer language (RTL). RTL is a compact language with operations so elementary that every machine has them, thus ensuring that every RTL program corresponds to some machine program. The simplicity of RTL means that a relatively large quantity of RTL code is sometimes needed to represent high-level operations. However, this is exactly the idea: provide verbose but simple code to the RTL System, thus enabling it to uncover all possible optimizations.

The sample program in Figure 2.1 illustrates the constructs available in RTL. RTL provides the usual assortment of unary and binary operators, as well as an infinite number of logical registers. There are also bit registers, which represent conditions such as `c` (carry), `n` (negative), `z` (zero), and `v` (overflow).

The RTL System works best when the RTL code it receives is as simple as possible. Computations that could be expressed using one RTL statement are instead broken up into several, with temporary registers holding intermediate results. This is because the RTL System forms machine instructions by combining as many RTL statements together as it can; it never breaks them down into simpler ones. Thus, a complex RTL statement may keep the RTL System from discovering an optimization. That is why the statements

```
r4 ← *r1.
r2 ← r2 + r4.
```

are used instead of

```
r2 ← r2 + *r1.
```

in Figure 2.1, even though the latter is equivalent and shorter.

One construct not shown in Figure 2.1 is the concurrency operator ";". For example,

```
r2 ← *(r1 - 4) ; r1 ← r1 - 4
```

expresses a pre-decrement memory access that might be written as

```
r2 ← *--r1
```

in other languages. The concurrency operator is used to represent complex instructions and addressing modes that are discovered by the RTL System during optimization and instruction selection, and does not usually appear in the RTL System's input.

*Compute the sum of 10 integers*
*starting at memory location 1000,*
*and put the result in memory location 500.*

| | |
|---|---|
| `r1 ← 1000.` | *Initialize pointer to address 1000.* |
| `r2 ← 0.` | *Initialize sum to 0.* |
| `r3 ← 0.` | *Initialize counter to 0.* |
| | |
| `L1:` | |
| `n ← r3 < 10.` | *If counter is less than 10, set negative bit.* |
| `~n ↑ L2` | *If negative bit is clear, jump to L2.* |
| `r4 ← *r1.` | *Fetch integer pointed to ...* |
| `r2 ← r2 + r4.` | *... and add it to sum.* |
| `r1 ← r1 + 4.` | *Increment pointer.* |
| `r3 ← r3 + 1.` | *Increment counter.* |
| `↑ L1.` | *Jump to L1.* |
| | |
| `L2:` | |
| `r5 ← 500.` | *Initialize a pointer to address 500 ...* |
| `*r5 ← r2.` | *... and store sum there.* |
| `@.` | *Return.* |

Figure 2.1: An RTL program.

# Chapter 3

# RTL Program Representation

An RTL program is represented by objects of four kinds of classes. Class **RTLProgram** represents the flow graph of a program. The subclasses of **FlowNode** represent the flow nodes in the flow graph. The subclasses of **RegisterTransfer** represent the register transfers in each flow node. The subclasses of **RTLExpression** represent the expressions in each register transfer.

Figure 3.1 illustrates how a program is represented in terms of objects in these classes by showing the flow graph for the RTL program in Figure 2.1. Notice that memory references are represented as functions taking an explicitly named memory object as an argument. This streamlines the treatment of memory operations, and is discussed fully below. The text version of a typical RTL program is shown in Appendix A.

## 3.1 Static Single Assignment Form

Global optimizations require information about the global data flow properties of a program: which transfers create the values used by a given transfer? This question is easy to answer if each register is assigned a value exactly once, and this is guaranteed by static single assignment (SSA) form, a program representation developed by researchers at IBM. The RTL System uses SSA form to create algorithms for common subexpression elimination [AWZ88], code motion out of loops [RWZ88], and other optimizations.

"Single assignment" languages are functional languages in which each variable is assigned a value exactly once at run-time, not merely in the program text. In contrast, any program can be put into static single assignment form; it is merely an alternative representation.

It is simple to rename registers in a basic block so that there is only one assignment to (i.e., definition of) each register. However, this isn't possible in the presence of conditionals or loops, since a given point in the program may be reached by conflicting definitions. In SSA form, the first node reached by a set of conflicting definitions starts with a *$\phi$-assignment* that merges them. A $\phi$-assignment has the form $U \leftarrow \phi(V, W, \ldots)$, where $U, V, W, \ldots$ are registers, and the number of operands $V, W, \ldots$ is the number of predecessors of the flow node in which the $\phi$-assignment appears. If control reaches the $\phi$-assignment through the $j$-th predecessor, then $U$ is assigned the value of the $j$-th operand. Each execution of a $\phi$-assignment uses only one of the operands, but which one depends on how control reached the flow node containing the $\phi$-assignment.

SSA form makes computing data flow information easy. It also makes thinking about optimizations easy, since the case where different definitions reach the same point no longer occurs.

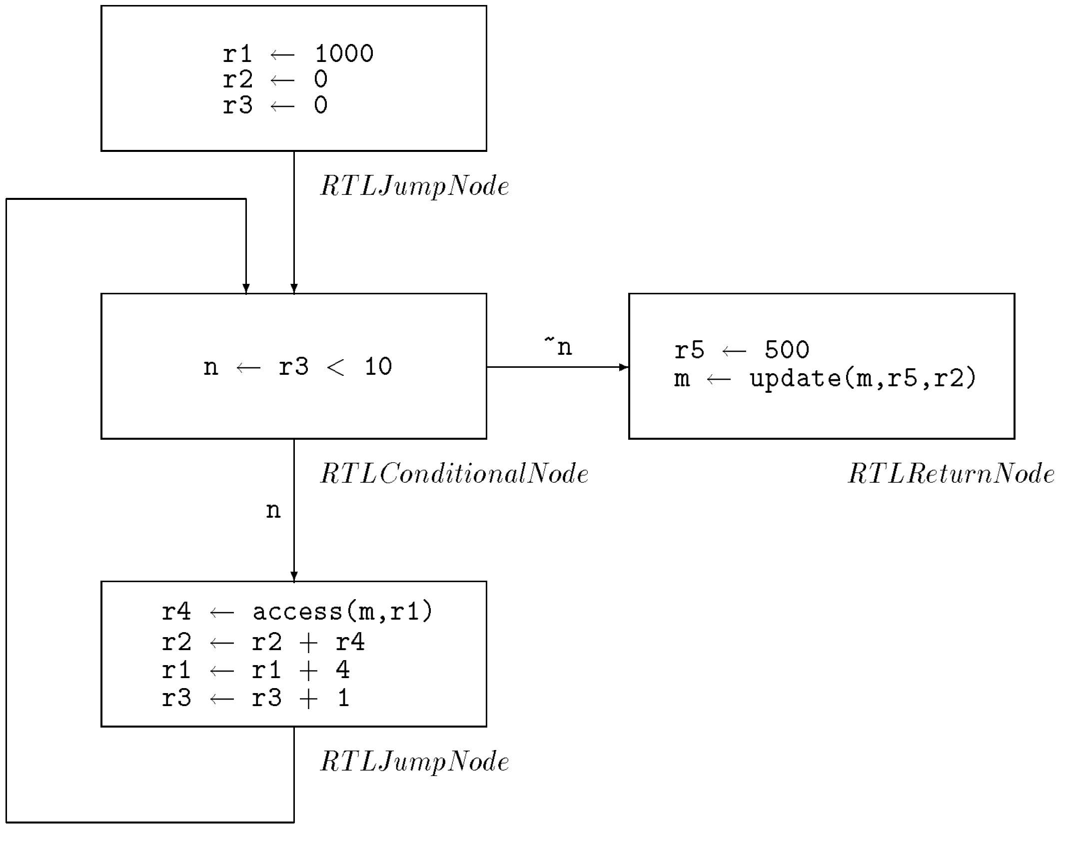



Figure 3.1: Flow graph for an RTL program.

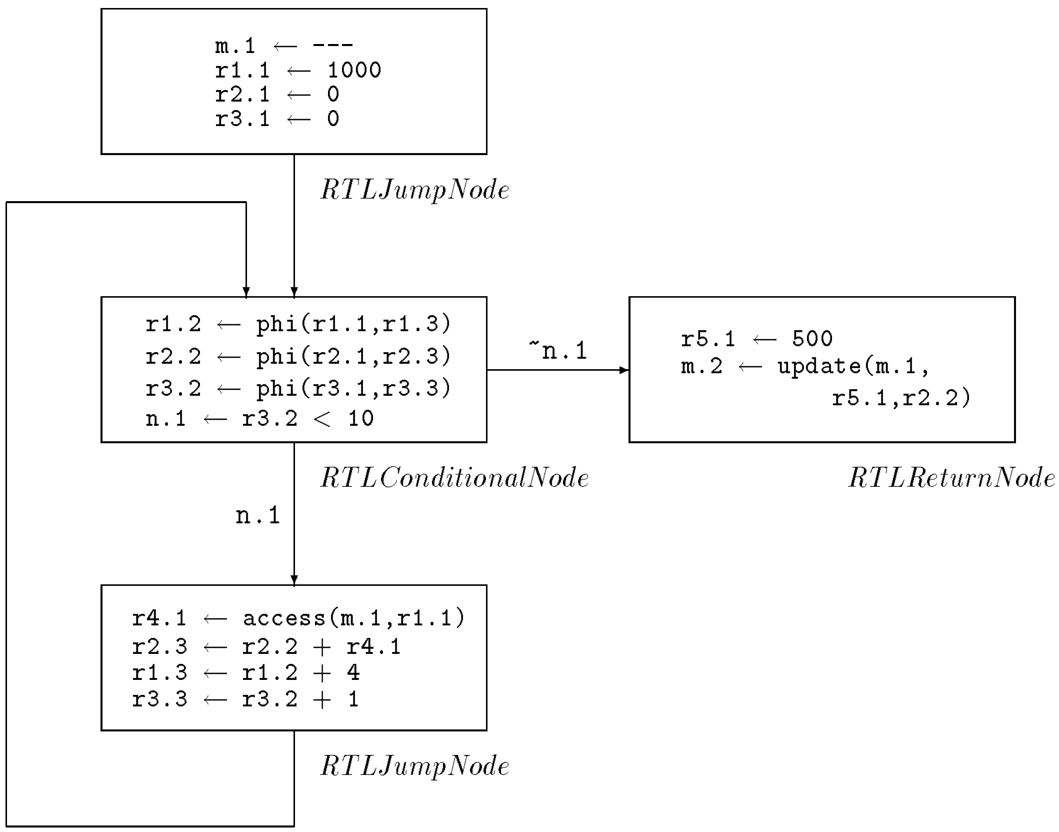



Figure 3.2: Flow graph for an RTL program after SSA conversion.

Figure 3.2 shows how the flow graph in Figure 3.1 looks after SSA conversion. Note how conflicting definitions of loop variables are resolved by $\phi$-assignments, which are shown as assignments with `phi(...)` on their right-hand sides in the figure. Also note the addition of *implicit assignments*, shown as assignments with `---` on their right-hand sides; these indicate that the destination does have a definite value, but one that is unknown at compile-time.

## 3.2 The Flow Graph Level

**RTLProgram** represents the flow graph of a program. It acts as a container for the flow nodes, and for anything that is global with respect to the optimization process. One particularly important global object is the instance of **MachineDescription** describing the target machine: all machine-dependent operations consult this object. **RTLProgram** also has methods for optimizing the RTL program, and for generating machine code, as well as for traversing the flow nodes in various ways.

Loops are structures worthy of special attention, since a program spends most of its time executing them. In the RTL System, a loop is referred to as an *interval* [Tar74], and is defined to be a single-entry, strongly connected region of the flow graph. Sometimes it is convenient to treat an interval itself as a flow node. For this reason, an interval is represented by an instance of **RTLIntervalNode**, which is a kind of flow node that may act on behalf of the nodes inside it. Intervals may of course be nested, so **RTLIntervalNode** has a pointer to the surrounding interval. The program is by definition the outermost interval. Since this makes an **RTLProgram** merely a special kind of interval, it is a subclass of **RTLIntervalNode**. Thus, the entire program may be viewed as a single, albeit special, flow node.

**RTLProgram** has the following instance variables:

machineDescription: a **MachineDescription** describing the target machine. This contains all machine-specific information, such as the number and kind of registers that are available.

expressions: an **ExpressionDictionary** (see below) in which all expressions are recorded.

flowNodes: a **SortedCollection** of the flow nodes in the flow graph.

rootNode: the root node of the flow graph.

## 3.3 The Flow Node Level

**FlowNode** and its subclasses represent the flow nodes in the flow graph, and act as containers for the register transfers. **FlowNode**s also capture the structure of the flow graph by maintaining pointers to the nodes which precede and succeed them, as well as to their dominators and intervals. Any flow-of-control optimization, such as branch elimination, takes place at the flow node level.

The **FlowNode** protocol includes many of the same kind of optimization and code generation methods as **RTLProgram**; in fact, many **RTLProgram** methods do little more than invoke the corresponding **FlowNode** methods. There are methods for adding, removing, and iterating over register transfers, and for changing the structure of the flow graph by giving a **FlowNode** different predecessors and successors.

In order to easily determine the relative order of flow nodes within an interval, each is assigned two topological numbers. One represents the node's position in a topological traversal of the nodes when every node's successors are traversed from left to right, and the other when successors are traversed from right to left. Thus, if each of the topological numbers in node $A$ is less than its counterpart in node $B$, then node $A$ is always executed before node $B$. This property is used to help determine the lifetime of registers during register allocation.

**FlowNode** is abstract, and has the following instance variables:

number: an integer used for printing and debugging.

inEdges: a collection representing the flow nodes that jump to this one.

hasBeenVisited: a boolean indicating whether this flow node has been visited in a traversal of whatever kind.

interval: the **RTLIntervalNode** containing this flow node.

**topNumber**: the number this flow node is assigned in a left-to-right topological traversal of the flow graph.

**topNumberAlternate**: the number this flow node is assigned in a right-to-left topological traversal of the flow graph.

There are also several instance variables that describe what nodes are related to this one by dominance.

The flow node hierarchy is

- **FlowNode**
  - **RTFlowNode**
    - **RTLJumpNode**
      - **RTLCallNode**
      - **RTLConditionalNode**
    - **RTLReturnNode**
  - **RTLIntervalNode**
    - **RTLProgram**

The reason for having a class hierarchy for flow nodes rather than just a single class is that *the class of a flow node implicitly represents how control changes after the basic block in that flow node finishes executing.* In other words, rather than ending basic blocks with a register transfer indicating how control changes, the flow node itself indicates this. This principle has three corollaries:

- Basic blocks always implicitly end (via the flow node that contains them) with a change of control. They never just "fall through" to a successor.
- Basic blocks consist only of assignments. They never contain a register transfer representing a change of control.
- The targets of jumps are always flow nodes. Flow nodes play the role that labels do in conventional representations.

This scheme simplifies the implementation of the RTL System, since methods that need to do different things depending on how control changes are simply implemented differently in each flow node class.

Each subclass of **RTFlowNode** has an instance variable for the register transfer with which the block implicitly ends. This implicit transfer allows the flow node to "act" like a register transfer when it needs to during, among other things, dependency computation and machine code generation.

The subclasses play the following roles:

**RTFlowNode**: the abstract superclass of flow nodes containing register transfers. Its instance variables are block, a doubly-linked list of the register transfers in this node; codePosition, the starting address of the machine code for block, used only during machine code generation; referencePositions, the positions in the machine code where the value of codePosition needs to be inserted, also used only during machine code generation; phiAssignments, an **OrderedCollection** of the $\phi$-assignments (see below) that begin block; and rtlProgram, the **RTLProgram** containing this flow node.

**RTLJumpNode**: represents nodes containing blocks that implicitly end with a jump. Its instance variables are outEdge, representing the flow node succeeding this one (to which this one "jumps"); and myJump, the implicit **Jump** transfer.

**RTLCallNode**: represents nodes containing blocks that implicitly end with a call followed by a jump, which is why it is a subclass of **RTLJumpNode**. Its only instance variable is myCall, the implicit **Call** transfer.

**RTLConditionalNode**: represents nodes containing blocks that implicitly end with a conditional jump followed by a jump, which is why it is a subclass of **RTLJumpNode**. Its instance variables are conditionalOutEdge, the flow node to which this node branches if the condition in myCondJump is true (otherwise branching to outEdge); and myCondJump, the implicit **CondJump** transfer.

**RTLReturnNode**: represents nodes containing blocks that implicitly end with a return. Its only instance variable is myReturn, the implicit **Return** transfer.

## 3.4 The Register Transfer Level

**RegisterTransfer** and its subclasses represent the register transfers that make up the program. A **RegisterTransfer** describes either a transfer of data—in other words, an assignment—or a transfer of control, such as a jump. However, as discussed in the previous section, control flow is represented at the flow node level. Control **RegisterTransfer**s are used only as a convenience in dependency analysis and machine code generation.

**RegisterTransfer** is abstract, and has the following instance variables:

number: an integer used for printing and debugging.

flowNode: the flow node containing this transfer.

prev: the transfer before this one in the flow node's block.

next: the transfer after this one in the flow node's block.

instruction: a **MachineInstruction** representing the machine instruction corresponding to this transfer.

topNumber: the number this transfer is assigned in a topological traversal of the flow nodes and a sequential traversal of the transfers in those flow nodes.

flowDependents: a **Set** of the register transfers that use registers set by this transfer.

flowSupporters: a **Set** of the register transfers that set registers used by this transfer.

The register transfer hierarchy is

**RegisterTransfer**
- **EmptyRegisterTransfer**
- **RegisterTransferSet**
- **Call**
- **AbstractAssignment**
  - **ImplicitAssignment**
  - **Assignment**
  - **PhiAssignment**
- **Jump**
  - **CondJump**
- **Return**

The subclasses play the following roles:

**EmptyRegisterTransfer**: used in the instruction builder to represent transfers that only set flags. See Chapter 8.

**RegisterTransferSet**: represents sets of **RegisterTransfer**s to be performed concurrently. Theoretically, this is a recursive structure, since it could contain an instance of itself; however, this is never allowed. Its only instance variable is transfers, an **OrderedCollection** of the component transfers.

**Call**: represents procedure calls. Its instance variables are method, the **Register** containing the callee's address; returnValueRegister, the **Register** in which the result of the call will be found; and argumentLogicalRegisters, an **OrderedCollection** of the **Register**s that contain the receiver and the first two arguments.

**AbstractAssignment**: the abstract superclass of all classes representing assignments to some storage. Its only instance variable is destination, the destination **Storage**.

**ImplicitAssignment**: represents assignments having an unknown source. These are currently used only to indicate which registers contain arguments. Examples of **ImplicitAssignment**s appear at the beginning of the program in Appendix A.

**Assignment**: represents assignments having a destination and a source. These are the most common kind of **RegisterTransfer**. Its only instance variable is source, the source **RTLExpression**.

**PhiAssignment**: represents assignments whose source is a $\phi$-assignment. Its only instance variable is nodeToRegisterMap, a **Dictionary** from flow nodes to the corresponding **Register** arguments of the $\phi$-assignment.

**Jump**: represents unconditional jumps. Its only instance variable is label, the target **FlowNode** of the jump.

**CondJump**: represents conditional jumps. Its only instance variable is test, the **RTLExpression** to evaluate to determine if a jump should be made.

**Return**: represents returns. Its instance variables are resultRegister, the **Register** that contains the return value; and registersUsed, an **OrderedCollection** of the **Register**s that the return uses.

## 3.5 The Expression Level

**RTLExpression** and its subclasses represent the expressions that are the basic components of a program. **RTLExpression** is abstract, and like most of its subclasses has no instance variables. The expression hierarchy is

- **RTLExpression**
  - **Storage**
    - **Register**
      - **SpecialRegister**
      - **BitRegister**
    - **Memory**
  - **MemoryAccess**
  - **AbstractUpdate**
    - **RegisterUpdate**
    - **MemoryUpdate**
  - **BinaryExpression**
  - **Constant**
  - **UnaryExpression**

**Storage** is the abstract superclass of all classes representing expressions that can be the target of an assignment. Its instance variables are

name: an integer denoting the name of this storage.

version: an integer denoting which version of the original storage named name this storage is. The original storage is version 0. New versions are created during the computation of SSA form.

definition: the **RegisterTransfer** that defines this storage. Thanks to SSA form, there will only be one.

counterForSSA: an integer indicating how many times this storage has been renamed during SSA computation.

registersForSSA: an **OrderedCollection** of the other storage units currently being used as renames for this storage during SSA computation.

One subclass of **Storage** is **Register**. A **Register** can represent a logical register or a machine register. Its instance variables are

physicalRegister: the physical machine **Register** assigned to this register.

lifetime: a **Lifetime** indicating the live range of this register.

The subclasses of **Register** are **BitRegister**, which represents condition codes; and **SpecialRegister**, which represents special registers such as the stack pointer.

The other subclass of **Storage** is **Memory**. As mentioned earlier, memory references are represented explicitly as functions in order to allow a unified treatment of operations on memory and operations on registers where dependency analysis and optimization are concerned; **Memory**s serve as one argument to these functions. The functions themselves are

```
access(m, x)
```

which returns the item at address `x` in memory `m`, and is represented by **MemoryAccess**; and

```
update(m, x, v)
```

which returns a fresh **Memory** after storing value `v` at address `x` in memory `m`, and is represented by **MemoryUpdate**[1].

The other subclasses play the following roles:

**BinaryExpression**: represents binary expressions. Its instance variables are first, the first **RTLExpression** operand; second, the second **RTLExpression** operand; and operation, a string indicating the binary operation to be performed. **BinaryExpression** has subclasses **MinusExpression** and **PlusExpression** representing the special cases where the operation is subtraction or addition, respectively.

**Constant**: represents constant expressions. Its only instance variable is name, the integer value of the constant expression.

**UnaryExpression**: represents unary expressions. Its instance variables are operand, the **RTLExpression** operand; and operation, a string indicating the unary operation to be performed.

**Variable**: used as a wildcard in expression matching. Whenever it is necessary to refer to all instances of a *particular* **RTLExpression** class (in describing a machine instruction, for example), an instance of that class is created with a **Variable** name. Whenever it is necessary to refer to all instances of *any* **RTLExpression** class, a **Variable** alone suffices. Its only instance variable is name, an integer.

## 3.6 Creating an Expression

Testing the equality of **RTLExpression**s is made efficient by ensuring that only one version of each **RTLExpression** exists. Thus, two **RTLExpression**s are equal if and only if they are the same object. One consequence of this scheme is that an **RTLExpression** can never be altered, since changing one occurrence of an expression would change all of them. Therefore, expressions are "altered" by creating altered copies of them.

Expressions are guaranteed to be unique by creating them using the **ExpressionDictionary** stored in the expressions instance variable of the **RTLProgram**. **ExpressionDictionary** implements a method for each kind of **RTLExpression**. Each of these methods take whatever arguments are necessary to define the expression, and returns an existing **RTLExpression** if possible, or a new one if not. For example, a **BinaryExpression** adding exp1 and exp2 would be created by sending BinaryExpressionFirst: exp1 operation: '+' second: exp2.

An **ExpressionDictionary** has a group of dictionaries, one for each kind of **RTLExpression**. Each dictionary is an instance of **OneKeyDictionary**, **TwoKeyDictionary**, or **ThreeKeyDictionary**. As the names imply, these dictionaries allow one, two, and three keys, and therefore

[1]**RegisterUpdate** is relevant only to data dependence computation, and so is discussed later in the section on that topic.

respectively implement at:, at:at:, and at:at:at: methods. The kind of dictionary used depends on the number of components the corresponding **RTLExpression** has. For example, a **ThreeKeyDictionary** is used for **BinaryExpression**s, and the method mentioned in the previous paragraph would ultimately send the message at: exp1 at: '+' at: exp2 to it to obtain a previously existing **BinaryExpression**, if one exists.

# Chapter 4

# RTL Program Generation

An **RTLProgram** is created by sending the generateRTL message to a Typed Smalltalk parse tree. This causes the parse tree to traverse itself and tell each of its parse nodes to generate code.

A parse node does not generate RTL code directly. Instead, it makes use of an interface object passed to it that acts as an intermediary between the parse node and the object that actually generates RTL code, the RTL *builder* (an instance of **RTLBuilder**). This *builder interface* is an instance of a subclass of **SmalltalkInterfaceForRTLBuilder** tailored to the run-time system being used. In the Typed Smalltalk compiler, the subclass is **TektronixSmalltalkInterfaceForRTLBuilder**. The builder interface acts as an expert on the run-time environment, and knows how to generate the RTL code for high-level operations such as loading an argument, returning a value, and so on. Thus, a parse node simply tells the builder interface what high-level operation it wishes to perform, and the builder interface in turn tells the builder what RTL code to generate. This design conveniently isolates run-time system dependencies. As an example, Figure 4.1 shows how parse nodes representing Smalltalk method arguments emit code for themselves.

The builder appends the RTL code it generates to a basic block [ASU86] it maintains. When the block is complete, the builder resolves label references, puts the block in a flow node, and adds the flow node to the **RTLProgram**. Figure 4.2 shows the method that does this.

Parse nodes representing calls to Smalltalk primitives generate code in a slightly different way. The compiler represents a Smalltalk primitive by an instance of **RTLProgram** created from RTL code describing what the primitive does. (As a pleasant consequence, primitives in the Typed Smalltalk compiler are not just "black boxes.") As a primitive is patched into the **RTLProgram** under construction, its formal arguments are replaced by actual arguments and registers are renamed so that they do not conflict with those already in use.

---

```
emitRTLForValueOn: rtlBuilder in: method

    ↑ rtlBuilder newRegisterWihArgumentNumber: self offset
```

Figure 4.1: Generating RTL code for a **TypedVarArgNode**.

---

```
flowGraphAdd: aNodeClass
    | node |
    node ← aNodeClass ofProgram: rtlProgram.
    "Convert all forward references to the current label(s) to
    references to the node that will contain the current block."
    currentLabel copy do:
        [:label |
        (self nodeForLabel: label)
            givePredecessorsNewSuccessor: node.
        nodeLabels at: label put: node.
        currentLabel remove: label].
    "Initialize the node with the current block."
    node block: currentBlock.
    currentBlock ← DLinkedList new.
    "Add the node to the program."
    rtlProgram addNode: node.
    "And if the last node simply fell through to this one rather
    than jumping to it, I have to tell that last node who its
    successor is."
    lastNode isNil ifFalse: [lastNode successor: node].
    lastNode ← node
```

Figure 4.2: Adding a flow node to an **RTLProgram**.

# Chapter 5

# RTL Program Optimization

This chapter discusses the optimizations carried out by the **RTLProgram** method optimize. Appendix B shows how the RTL program in Appendix A looks after optimization.

## 5.1 Overview

The **RTLProgram** method optimize in Figure 5.1 orchestrates the optimization of an RTL program by sending messages to self. The **RTLProgram** methods typically delegate most of the work involved to the flow nodes, often doing no more than telling each flow node to perform the optimization. Each flow node in turn may delegate the optimization to the register transfers it contains. Thus, an optimization passes control down through the representation levels, with objects at one level coordinating those at the next.

As a comment in the optimize method suggests, an optimization should not undo the results of earlier optimizations: once an optimization establishes some condition (for example, that no unreachable code exists), that condition should be maintained. The RTL System accomplishes this using several techniques:

- It performs some optimizations in a certain order. For example, copy propagation can create dead assignments, and so precedes dead assignment elimination.
- It retries some optimizations. For example, eliminating jumps on a constant condition can create non-maximal basic blocks, and so before returning, the method that eliminates such jumps invokes the method that maximizes basic block sizes.
- It relies on the self-optimizing nature of the RTL program representation. One example of this is unreachable code elimination: the RTL System explicitly eliminates unreachable code once, and thereafter relies on the RTL program representation itself to remove unreachable code as it arises.

The remaining sections of this chapter trace optimizations through the various representation levels, and discuss other relevant computations such as flow analysis.

```
optimize
    self eliminateUnreachableCode.
    self eliminateJumpChains.
    self setNodeTopologicalNumbers.
    "The node list is now in the topological order necessary for
    basic block maximizing and CSE."
    self maximizeBasicBlockSizes.
    "Three conditions now hold: there is no unreachable code,
    there are no jump chains, and basic blocks are maximal.
    The first two of these will be automatically
    maintained by the data structures;
    the last one must be explicitly maintained by retrying the
    corresponding optimization when appropriate."
    self eliminateCommonSubexpressions.
    self eliminateConstantConditionals.
    "Expression caches must now be assumed incorrect."
    self convertToSSAForm.
    self computeRegisterDefinitions.
    self computeDependencies.
    self propagateCopies.
    self eliminateDeadAssignments
```

Figure 5.1: Optimizing an RTL program.

## 5.2 Eliminating Unreachable Code

Unreachable code is code to which there is no control path. In the RTL program representation, this corresponds to code in a flow node that is not the root node of the flow graph, and that has no predecessors. Although it might seem wrong for unreachable code to be generated in the first place, it is often convenient for the front end of the compiler to do so rather than check for some special case.

The **RTLProgram** method eliminateUnreachableCode detects and removes unreachable code:

```
eliminateUnreachableCode
    "Eliminate code that can't be reached."
    self nodesDo: [:node | node isReachable ifFalse: [node removeFromSuccessors]]
```

This, in turn, calls on the isReachable method in **FlowNode**:

```
isReachable
    ↑predecessors size > 0 | (rtlProgram rootNode == self)
```

As mentioned earlier, it is never necessary to retry unreachable code elimination, since whenever a flow node is removed, it recursively removes any of its successors that consequently become unreachable.

## 5.3 Eliminating Jump Chains

A jump chain is a sequence of "jumps to jumps." In the RTL program representation, this corresponds to a chain of empty **RTLJumpNode**s. The **RTLIntervalNode** method eliminateJumpChains eliminates such chains:

```
eliminateJumpChains
    "Eliminate jumps to other jumps."

    self nodesDo: [:node | node eliminateJumpChain]
```

Most subclasses of **RTFlowNode** do nothing in response to the eliminateJumpChain message, since the optimization does not apply to them. The exception is **RTLJumpNode**:

```
eliminateJumpChain
    "Remove myself from the program and splice my predecessors
    and successor together if I do nothing but an unconditional jump.
    However, I can't remove myself if my successor contains phi
    assignments because the register allocator might need to put
    transfers in me if it fails to handle one of them. I also can't
    remove myself if I jump to myself, since then I'm an infinite loop."

    (block isEmpty
        and: [successor phiAssignments isEmpty
        and: [successor ~~ self]])
            ifTrue: [self givePredecessorsNewSuccessor: successor]
```

It is never necessary to retry this optimization, since whenever a flow node removes a register transfer from itself, it sends itself the eliminateJumpChain message.

## 5.4 Maximizing Basic Block Sizes

Code generators often needlessly divide basic blocks among several flow nodes. Moreover, some optimizations delete flow nodes, often creating an opportunity to form larger basic blocks through merging. A pair of nodes is eligible for merging if the first has the second as its sole successor, and the second has the first as its sole predecessor.

Maximizing basic block sizes means merging all eligible nodes. The **RTLIntervalNode** method maximizeBasicBlockSizes does this:

```
maximizeBasicBlockSizes
    "Make basic blocks as big as possible by merging a node
    and its successor if that successor has only one predecessor.
    It is important to note that merging two nodes results with
    the successor. Thus, as long as we iterate over the nodes in
    topological order, we won't miss possible merges."

    self nodesDo: [:node | node maximizeBasicBlockSize]
```

Most **RTFlowNode**s return self in response to the maximizeBasicBlockSize message, since they cannot merge with their successor and still comply with the rule that control only changes after a basic block. The exception is again **RTLJumpNode**:

```
maximizeBasicBlockSize
    "Ask my successor to merge with me."

    successor mergeWithPredecessor
```

An **RTFlowNode** method finally performs the merge, if appropriate:

```
mergeWithPredecessor
    "Sent only by a predecessor of mine, which is always a
    node with just one successor. I merge if there are
    no other predecessors."

    | predecessor |
    predecessors size == 1 ifFalse: [↑self].
    predecessor ← predecessors first.
    predecessor prependCopyToSuccessors.
    predecessor givePredecessorsNewSuccessor: self
```

The technique of bouncing messages between two objects in order to coordinate an operation is known as *double dispatching.*

## 5.5 Eliminating Common Subexpressions

Common subexpression elimination replaces expressions by equivalent but simpler expressions. The RTL System uses the algorithm described in [AWZ88], which detects when two computations produce equivalent values. Two variables are said to be *equivalent* at a point $P$ if those variables contain the same values whenever control reaches $P$ during any possible execution of the program. The algorithm works by initially assuming that all values are equal and then separating them into more and more sets of possibly equal variables. It is conservative in that any variables found to be equivalent are in fact equivalent, but not all equivalences are found. The SSA representation makes determining equivalence of variables simple.

## 5.6 Eliminating Constant Conditionals

A constant conditional is a conditional jump in which the condition can be evaluated at compile time, thus making the jump unnecessary. In the RTL System, the notion of a constant jump condition is broadened to embrace the situation shown in Figure 5.2, which arises as a result of compiling the ifTrue:ifFalse: message. Here, the value of the condition is not constant in the flow node containing it. However, it *is* constant if propagated up into the preceding flow nodes. In such a case, the conditional node is replaced by two jump nodes, each containing a copy of the conditional node's code. Any unnecessary code will be removed during dead assignment elimination. Figure 5.3 shows the result.

The **RTLProgram** method eliminateConstantConditionals eliminates constant conditionals:

```
eliminateConstantConditionals
    "Eliminate branches on a condition whose value is known."
    "This optimization may invalidate the expression caches."

    self nodesDo: [:node | node eliminateConstantConditional].
    self maximizeBasicBlockSizes
```

Most subclasses of **RTFlowNode** return self in response to the eliminateConstantConditional message, since the optimization does not apply to them. The exception is **RTLConditionalNode**:

```
eliminateConstantConditional
    "Eliminate this node if it does a superfluous equality check."

    | result newNode replacement |
    result ← myCondJump test evaluate: expressionCache.
    result isNil ifTrue: [↑self tryReroutingPredecessors].
    newNode ← RTLJumpNode ofProgram: rtlProgram.
    rtlProgram addNode: newNode.
    replacement ← result
                ifTrue: [conditionalSuccessor]
                ifFalse: [successor].
    newNode successor: replacement.
    self prependCopyTo: newNode.
    self givePredecessorsNewSuccessor: newNode
```

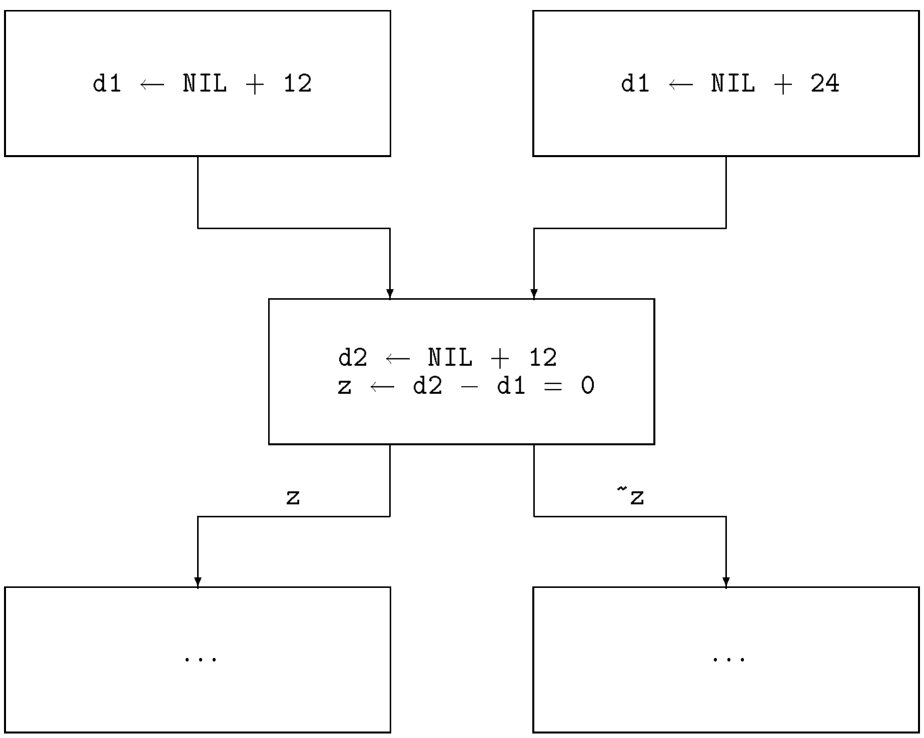



Figure 5.2: Flow graph fragment before conditional elimination.

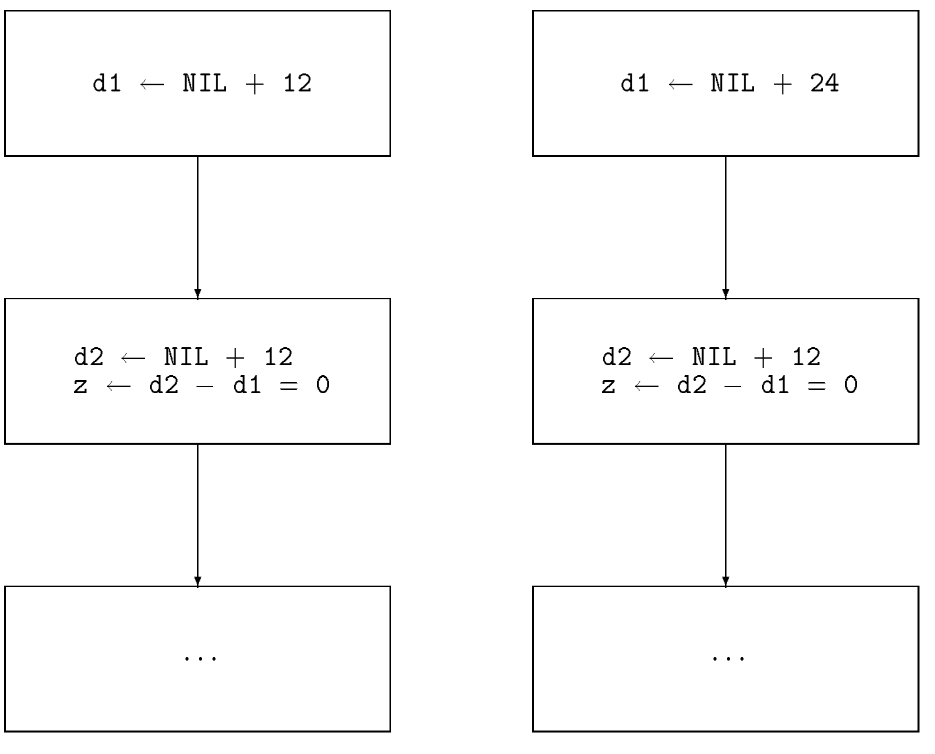



Figure 5.3: Flow graph fragment after conditional elimination.

```
tryReroutingPredecessors
    "See if my predecessors have enough information to
    evaluate the condition; if so, they can be rerouted around me."

    | replacement result newNode |
    predecessors copy do:
        [:node |
        "The idea is to evaluate the jump condition in my
        predecessor's expression cache. However, the bit
        register in the jump condition will be undefined
        there, so I want it looked up in MY
        expression cache. To accomplish this, I implement
        part of the ExpressionCache protocol."
        self primaryCache: node expressionCache.
        result ← myCondJump test evaluate: self.
        result isNil
            ifFalse:
                [replacement ← result
                            ifTrue: [conditionalSuccessor]
                            ifFalse: [successor].
                newNode ← RTLJumpNode ofProgram: rtlProgram.
                rtlProgram addNode: newNode.
                newNode successor: replacement.
                self prependCopyTo: newNode.
                node replaceSuccessor: self with: newNode]].
    self primaryCache: nil
```

Figure 5.4: Rerouting around a constant conditional.

This method attempts to evaluate the jump condition using only the expression cache of the **RTLConditionalNode**. If it succeeds, it replaces the conditional node with a single jump node. Otherwise, the tryReroutingPredecessors method shown in Figure 5.4 is invoked. This method evaluates the condition using the expression caches of the **RTLConditionalNode**'s predecessors, and applies the optimization shown in Figures 5.2 and 5.3 if appropriate.

## 5.7 Computing Global Data Dependencies

A data dependence exists between two register transfers if one writes to storage—either a register or memory—that the other uses. In traditional representations, there are three kinds of data dependency [Kuc78], but in the SSA representation there are only two: *flow dependency* and *anti dependency*. If transfer $A$ defines storage that transfer $B$ uses, then $A$ is a *flow supporter* of

$B$, and $B$ is a *flow dependent* of $A$. Similarly, if $A$ uses storage that $B$ (re)defines, then $A$ is an *anti supporter* of $B$, and $B$ is an *anti dependent* of $A$. Computing global data dependencies means determining the supporters and dependents of each register transfer in the program. This information is subsequently used in most optimizations—dead assignment elimination and common subexpression elimination being two examples—as well as in scheduling.

The conversion to SSA form is accomplished in a straightforward manner by the **RTLProgram** method convertToSSAForm using the algorithms in [CFR*89], and will not be discussed further here. After SSA conversion, flow dependencies are easily computed in two steps. The first step, performed by the **RTLProgram** method computeRegisterDefinitions, associates each storage unit with the register transfer that defines it:

```
computeRegisterDefinitions
    "Tell each register in the program which transfer defines it,
    and return an OrderedCollection of the registers. (Thanks
    to SSA form, each register is set in exactly one transfer.)"

    | registers |
    registers ← OrderedCollection new: 5 * flowNodes size.
    flowNodes do: [:node | node assignmentsDo: [:rt | rt registersSet do:
                    [:register |
                    registers add: register.
                    register definition: rt]]].
    ↑registers
```

The second step computes flow dependencies, and is performed by the computeDependencies method of **RTLProgram**:

```
computeDependencies
    "Compute the dependencies between register transfers."

    flowNodes do: [:node | node transfersDo: [:rt | rt computeDependencies]]
```

This method, in turn, calls upon the **RegisterTransfer** method of the same name to do the actual work of linking each transfer to its flow supporters:

```
computeDependencies
    "Determine which transfers I depend on.
    Since for convenience we don't add implicit
    assignments to every register to the beginning of
    the program, it is possible for an argument to a
    phi assignment to be undefined. This only happens
    when the phi assignment is dead, and is why we
    have to check for a nil definition."

    self registersUsed do: [:register |
        register definition isNil ifFalse: [register definition addFlowDependent: self]]
```

Anti dependencies are not stored explicitly, but instead are computed whenever they are needed from the flow dependencies as described below. First, however, anti dependence merits more discussion.

### 5.7.1 SSA Form and Anti Dependence

SSA conversion removes anti dependencies involving registers. For example, the anti dependence that exists between the statements

```
r2 ← r1 + 1.
r1 ← 2.
```

due to `r1` is broken when the statements are put into SSA form

```
r2.1 ← r1.1 + 1.
r1.2 ← 2.
```

thus making it permissible to reorder the two statements, for example. The register assigner (described later) is free to assign whatever physical registers it wishes to `r1.1` and `r1.2`.

However, SSA conversion does not remove anti dependencies involving memory or *special registers*, which are logical registers that must be assigned a particular physical register. (Memory and special registers will be collectively referred to as *special storage*.) For example, assuming that `SP` represents the stack pointer, the anti dependence in

```
r1 ← SP.
SP ← SP + 4.
```

continues to exist in

```
r1.1 ← SP.1.
SP.2 ← SP.1 + 4.
```

despite SSA conversion, since both `SP.1` and `SP.2` *must* be assigned the same physical register, the stack pointer. Similarly, there is an anti dependence in

```
r2.1 ← access(m.1, r1.1).
m.2 ← update(m.1, r3.1, 0).
```

involving `m.1` and `m.2`, because the second transfer might write to the same memory location that the first transfer read.

As previously mentioned, anti dependencies are computed from flow dependencies dynamically as needed, since they occur relatively infrequently. This is done using the relationships[1]

$$\begin{aligned}
\text{antiCandidates}(T) &= \left( \bigcup_{x \,\in\, \text{flowSupporters}(T)} \text{flowDependents}(x) \right) - T \\
\text{antiSupporters}(T) &= \{x \mid x \in \text{antiCandidates}(T) \wedge \text{trashes}(x, T)\} \\
\text{antiDependents}(T) &= \{x \mid x \in \text{antiCandidates}(T) \wedge \text{trashes}(T, x)\}
\end{aligned}$$

[1]Technically, these relationships do not yield all anti dependents and supporters, but only the nearest ones.

where the predicate trashes$(a, b)$ is true iff $a$ writes to physical storage that $b$ uses, regardless of the logical names involved; again, even though `SP.1` and `SP.2` (for example) are logically distinct, by necessity they represent the same physical storage. These relationships hold only because *any special storage that a transfer defines, it also uses*: as demonstrated in the previous examples, any transfer that sets a version of `SP` or `m` also uses a version of them[2]. Since antiCandidates$(T)$ usually has few members, the computation of anti dependencies takes little time.

## 5.8 Propagating Copies

Copy propagation is a special kind of common subexpression elimination in which the destination of a register-to-register assignment is replaced by the source throughout the program. This optimization is done separately because the current CSE algorithm has not yet been replaced with one that can take advantage of the SSA representation [AWZ88].

The **RTLProgram** method propagateCopies propagates copies:

```
propagateCopies
    flowNodes do: [:node | node assignmentsDo: [:rt | rt propagateIfCopy: expressions]]
```

Since this optimization only applies to assignments, only subclasses of **AbstractAssignment** need understand the propagateIfCopy message. The only such subclass that does not simply return self in response to this message is **Assignment**; Figure 5.5 shows what it does.

Copy propagation usually produces dead assignments, which the phase described next eliminates.

## 5.9 Eliminating Dead Assignments

Dead assignments are assignments whose destination is never used; in other words, assignments with no flow dependents. Thus, once dependency information has been computed it is quite easy to remove dead assignments.

The **RTLProgram** method eliminateDeadAssignments eliminates dead assignments:

```
eliminateDeadAssignments
    "Eliminate dead assignments."

    self nodesDo: [:node | node assignmentsDo: [:rt | rt removeIfDead]]
```

Since this optimization only applies to assignments, the removeIfDead method is only implemented

[2]A transfer such as `SP ← r1` is converted to `SP ← registerUpdate(SP, r1)` in order to conform to this rule. This is the only situtation in which **RegisterUpdate**, mentioned earlier, is used.

```
propagateIfCopy: expressions
    | supporter |
    (destination isRegister and: [source isRegister])
        ifFalse: [↑self].
    "If my source is a constant register like NIL, I have no
    supporters. Otherwise, I can have only one supporter,
    thanks to SSA form and the fact that my source is a register."
    flowSupporters isEmpty
        ifTrue: [supporter ← nil]
        ifFalse: [supporter ← flowSupporters first].
    flowDependents copy do: [:rt | rt isUpdatable
                ifTrue:
                    [rt
                        replace: destination
                        with: source
                        expressions: expressions.
                    self removeFlowDependent: rt.
                    supporter isNil ifFalse: [supporter addFlowDependent: rt]]]
```

Figure 5.5: Propagating copies.

in **AbstractAssignment**, where it recursively removes dead assignments:

```
removeIfDead
    "Remove myself if I'm dead, and recursively remove
    anybody who becomes dead as a result."

    | supporters |
    self isDead ifFalse: [↑self].
    supporters ← flowSupporters copy.
    self removeFromProgram.
    supporters do: [:rt | rt removeIfDead]
```

Finally, an assignment is dead if it sets a register but has no dependents:

```
isDead
    "Return true if this transfer is dead."

    destination isRegister ifFalse: [↑false].
    flowDependents isEmpty ifFalse: [↑false].
    ↑true
```

# Chapter 6

# Conclusion

This section has shown how an object-oriented programming language such as Smalltalk can be used to create a framework—a collection of abstract classes and their interfaces—for performing a given task, in this case code optimization. By providing what is essentially a metalanguage for optimization, this framework simplifies the implementation of the RTL System.

Also simplifying the implementation of the RTL System is the symbolic nature of Smalltalk. Garbage collection makes it easy to create the appropriate data structures. First-class functions make it possible to define appropriate control constructs, such as depth-first iteration. Since code optimization is a symbolic process, it is only natural to use a symbolic language to implement it.

Using SSA form to represent the data flow properties of a program not only makes the RTL System easier to implement, it also makes it easier to think about optimizations. In considering any data optimization, the difficult case is always the one in which multiple definitions of a variable reach the same point in the program. Since in SSA form every variable in the program is set exactly once, this case never arises. SSA form deserves to become one of the standard techniques in code optimization.

As the appendices show, the RTL System is quite effective, and thanks to its clean design, it is relatively easy to make it even more effective.

# Part II

# Scheduling and Machine Code Generation

# Chapter 7

# Introduction

All compilers are machine-dependent; they translate programs into the machine language of a particular machine. Usually the first stages of a compiler are machine-independent and depend only on the language being compiled, while later stages are progressively more machine-dependent.

The last decade has seen a great deal of work on automating the construction of the machine-dependent parts of compilers [DF84], [FW88], [Hen87],[HD89]; it is now much easier to port a compiler from one machine to another machine. This portability is achieved by using the technique of matching intermediate instruction trees against a grammar that describes the machine. Thus, porting to a new machine requires only writing a grammar for the new machine, not rewriting any algorithms in the compiler. However, this technique works best for similar machines, and none of the portable code generators produce high quality code for both RISC and CISC machines without rewriting quite a bit of the code generator.

This section[1] describes scheduling and machine code generation in the RTL System. Although it is language independent and could be used as a code generator for a compiler for any language, the fact that it was written in Smalltalk has had a major impact on its design. It not only uses tree matching techniques like the other portable code generators, but it also provides an object-oriented framework for customizing algorithms and data structures. This framework provides a machine-independent foundation for instruction selection and instruction scheduling.

Like some of the other code generators [Dav81], [DF84], [Sta89], the RTL System is based on a register transfer language, hence its name. The RTL System is unique in using static single assignment form to represent dependencies between register transfers. One effect of this is that programs are represented as directed acyclic graphs. A major part of code generation is converting the program from a DAG to a sequence of instructions.

The register transfer language has proven to be a very good representation for optimizations such as common subexpression elimination and constant propagation. However, this section describes how the RTL System performs code generation, not optimization. This includes pipeline scheduling and peep-hole optimization, since these are extremely machine-dependent. Thus, they are considered to be part of code generation rather than part of code optimization.

[1]This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

# Chapter 8

# Machine Code Generation

There are three kinds of objects that are involved in code generation. The first kind are the objects that represent the RTL program. The most important of these are register transfers. The second kind are machine instructions, which represent the machine language instructions for the target machine. The third kind is the machine description, which has a number of responsibilities. One of the most important is determining whether a register transfer matches a machine instruction and if so, which machine instruction.

The process of assigning instructions to register transfers is triggered by the scheduler. When a register transfer needs to report its instruction to the scheduler, it requests an instruction from the machine description.

Most machine descriptions forward the message to an instance of **InstructionBuilder**. The instruction builder contains one or more tables that represent the instruction set. It is responsible for choosing a machine instruction for the transfer and instantiating it with the appropriate values.

## 8.1 MachineInstruction

Machine instructions are responsible for binding themselves to register transfers and for generating code for themselves. On CISC machines, instructions contain operands that represent the various addressing modes of the machine. For a table-driven implementation of an instruction set, there are three "events" that happen to machine instructions.

First, an instruction template is created for each register transfer that can be implemented by the machine. These templates contain pattern variables that are instantiated when the instruction is bound to a register transfer. All templates are converted into three-address form before being passed to the instruction builder.

Second, once the instruction builder has matched a register transfer with an instruction template, the template is bound to the transfer. The template usually does this by instantiating the pattern variables or requesting operands from the instruction builder. It is possible that binding will fail. For example, an add-quick on the Motorola 68000 will bind successfully only when the addend is between 1 and 8.

Third, the instruction is sent a message to generate a binary representation of itself. Since machine instructions produce binary directly, compilers written with the RTL System do not need assemblers. A CISC instruction will usually generate code by sending messages to its operands.

Implementing a new architecture requires implementing a set of classes for the various kinds of machine instructions. The number depends on the machine. There are four for the National 32000, eight for the Motorola 68000, and six for the SPARC.

The following is a good general guideline for designing the instruction hierarchy for an instruction set. Each field in the byte code for an instruction should be an instance variable of the instruction. Following this guideline will make generating code trivial since the values necessary to generate code will be available in the instance variables.

This guideline suggests a subclass of **MachineInstruction** for each instruction format. For most machines, quite a bit of code sharing will be possible between the subclasses. Keeping the hierarchy in close correlation with the instruction formats will make the instruction set implementation clean and straightforward.

A good example of the importance of maintaining a close correlation between the instruction formats and the implementation is the use of operands. Originally, the Motorola instruction set was implemented without operands. It worked well, but some elements of the design were lacking. For example, there was machine dependent code in the machine-independent classes that represent RTL expressions.

When the National 32000 instruction set was implemented, the necessity of operands was recognized. When the Motorola was reimplemented with operands, the design was improved. Thus the close correlation between the design and the real-world entity was important. This is also a good example of how a design can be improved by reimplementing and "stretching" it.

## 8.2 InstructionBuilder

In the RTL System, a register transfer is associated with a machine instruction by sending the **rtAsInstruction:** message to the machine description, which returns a machine instruction. This message is the one and only way to associate register transfers with instructions. One could imagine a machine that performs all computation with only four or five instructions. This machine could implement the **rtAsInstruction:** message as a case statement.

However, most conventional machines (including RISCs) have an instruction set large enough to make matching instructions with a case statement unfeasible. Instead, most machine descriptions use a table-driven approach to match register transfers with instructions. Instances of the class **InstructionBuilder** are responsible for implementing the table-driven approach.

The instruction table is a good example of how tables are implemented in Smalltalk. The table is initialized through methods of the instruction class for the target machine. The messages sent by these methods have several arguments, and the arguments can be considered to be columns in the instruction table. The table initialization methods for the National 32000 are shown in the Appendix.

The instruction builder was originally a subclass of **Dictionary**. It is now an object that contains an instance of **Dictionary**. There are several reasons for having the dictionary be an instance variable. First, storing and retrieving instructions require calling special matching methods that are more complicated than = and hash. Second, simplification of register transfers can be cleanly encapsulated into the instruction builder. Lastly, it turns out that different types of instruction builders are necessary for different types of target machines, indicating that a hierarchy of instruction builders is necessary.

There is not always a one-to-one correlation between register transfers and machine instructions. For example, the SPARC has different versions of LOAD and STORE for different operand sizes (byte, half-word, word, or double-word). Thus, the instruction table maps register transfers to lists of instructions. A portion of the instruction table for the SPARC is shown below:

| Register Transfer | Instructions |
|---|---|
| $r\$1 \leftarrow *(r\$2 + r\$3)$ | LDSB, LDSH, LD, LDD |
| $*(r\$2 + r\$3) \leftarrow r\$1$ | STB, STH, ST, STD |

When the instruction builder is asked to find an instruction for a transfer, the following sequence of events occurs. The instruction builder hashes the transfer and retrieves the collection of instructions that might match. It then iterates through the collection. It tries to instantiate the instruction using the bindings of the pattern variables that are required in order for the template to match the register transfer. When an instruction is found that binds successfully, it is returned. The **InstructionBuilder** method **rtAsInstruction:expressions:** (Figure 8.1) implements this process.

The flag portion of the register transfer is matched separately from the rest of the transfer because a transfer that sets only one flag will need to match an instruction that sets that flag and possibly many others. The **matchFlags:expressions:** message insures that the flag portion of the register transfer is a subset of the flag portion of the instruction.

```
rtAsInstruction: anRT expressions: expressions
    "Return an instruction that implements the given RT. If
    none could be found, return nil."

    | instructions bindings newRT theInstruction |
    newRT ← anRT simplify: expressions.
    newRT ← newRT canonicalizeFor: machineDescription expressions: expressions.
    instructions ← instructionDictionary at: (self instructionHashStringFor: newRT) ifAbsent: [↑nil].
    instructions do:
        [:poss |
        bindings ← OneKeyDictionary new.
        ((poss rtl
            matchExceptFlags: newRT
            with: bindings
            expressions: expressions)
            and: [poss flags
                    matchFlags: newRT
                    with: bindings
                    expressions: expressions])
            ifTrue:
                [theInstruction ← poss copy.
                (theInstruction bind: bindings expressions: expressions)
                    ifTrue: [↑theInstruction]]].
    ↑nil
```

Figure 8.1: Associating a register transfer with an instruction.

Notice that the instructions in the collection are implicitly prioritized by their order. This can be advantageous. For example, a machine that implements an add-quick instruction should have the add-quick instruction placed first in the collection of instructions that matches an add register transfer. This way, any add register transfers that can be implemented as an add-quick will bind successfully and use the add-quick instruction. Other adds will fail to bind successfully with the add-quick instruction, but will bind with a conventional add later in the collection.

## 8.3 CISCInstructionBuilder

The complicated instruction sets of CISC machines require a sophisticated instruction builder. A good example is the National 32000 series, which implements eight addressing modes. Each of the eight modes can be indexed in four ways. This causes a move instruction, which has 32 possibilities for the source and 32 possibilities for the destination, to match 1024 different register transfers!

This combinatorial explosion is handled by a subclass of **InstructionBuilder**, called **CISCInstructionBuilder**, that contains an operand table in addition to the instruction table. The entries in the instruction table are templates with pattern variables for place holders. These templates are filled out with entries from the operand table. For example, the following is the entry for add instructions in the table for the National 32000 series:

| Register Transfer | Instructions |
|---|---|
| \$-1 $\leftarrow$ \$2 + \$1 | ADDQ, ADD |

The symbol $ is used to represent a pattern variable in RTL; expressions such as $2, $1, and $-1 will match any memory reference, register, or constant. Consequently, instruction-level matching is based on the structure of the register transfer.

The CISC instruction builder looks up a register transfer in the table in exactly the same way as the instruction builder. The matching process generates a set of bindings of pattern variables to RTL expressions. These bindings are passed to the instruction that was matched. The instruction then asks the CISC instruction builder to create operands for the RTL expressions that were matched to the pattern variables.

The instruction requests operands by sending the **rtAsOperand:for:operandNumber:expressions:** message (Figure 8.2) to the instruction builder. This method is similar to the **rtAsInstruction:expressions:** method, except the instruction must be sent the **operandOk:operandNumber:** message to make sure that the operand and the instruction are compatible.

## 8.4 Constraints

The RTL System uses static single-assignment (SSA) form [AWZ88]. A program in SSA form gives each register exactly one definition. This means that a 2-address register transfer such as $r101 \leftarrow r101 + r102$ will never appear in an RTL program. Instead, a 3-address transfer such as $r101.2 \leftarrow r101.1 + r102.1$ will be used.

Naturally, the entries in the instruction table must also be in SSA form in order to match register transfers in SSA form. However, instructions on 2-address machines cannot be represented by transfers in SSA form.

```
rtAsOperand: anRT for: anInstruction operandNumber: aNum expressions: expressions
    | operands bindings theOperand |
    operands ← operandDictionary at: anRT matchingHashString ifAbsent: [↑nil].
    operands do:
        [:poss |
        bindings ← OneKeyDictionary new.
        (poss rtl
            match: anRT
            with: bindings
            expressions: expressions)
            ifTrue:
                [theOperand ← poss copy.
                ((theOperand bind: bindings expressions: expressions)
                    and: [anInstruction operandOk: theOperand operandNumber: aNum])
                    ifTrue: [↑theOperand]]].
    ↑nil
```

Figure 8.2: Associating an RTL expression with an operand.

This conflict is resolved by performing a conversion of the register transfers before they are put in the table. The table entries are converted into three-address form, but constraints are stored for the register allocator to use later. For example, the following is a template for a register add on the Motorola 68020. In this case, $ will match any register name. Thus, r$2 could match any register.

$$r\$2 \leftarrow r\$2 + \$1$$

This will be converted into the following transfer, in three-address form:

$$r\$\text{-}1 \leftarrow r\$2 + \$1$$

A constraint will be stored to ensure that r$2 and r$-1 are eventually assigned to the same physical register. It is the responsibility of the register allocator to enforce this constraint.

Constraints are more complicated for instructions that contain pattern variables that are placeholders for operands. The following template appears in the instruction table for the National 32000. A constraint is stored with the template associating $2 and $-1.

$$\$\text{-}1 \leftarrow \$2 + \$1$$

This constraint is more complicated than a register constraint because $2 and $-1 could match any memory access, register, or constant.

Thus, when the above instruction template is bound to a register transfer, the constraint must ensure that $2 and $-1 are bound to similar RTL expressions. Further, register constraints may be necessary to insure that the physical registers for the expressions are the same.

**MemoryAccess**:

*Protocol for: constraining*

**constrainToBe: aDestination**
aDestination isMemoryAccess ifFalse: [↑nil].
↑address constrainToBe: aDestination address

**Register**:

*Protocol for: constraining*

**constrainToBe: aDestination**
(self eqClass: aDestination)
ifFalse: [↑nil].
self category = aDestination category ifFalse: [↑nil].
↑OrderedCollection with: (RTLConstraint source: self destination: aDestination)

Figure 8.3: Constraining an RTL expression.

This is done by the method **constrainToBe:** that is implemented by all RTL expressions. This method either fails and returns nil or succeeds and produces register constraints. The **constrainToBe:** methods of **MemoryAccess** and **Register** are shown in Figure 8.3.

## 8.5 Instruction Assignment Example

Suppose that the target machine is the National 32000 and the **rtAsInstruction:expressions:** message is sent to a **CISCInstructionBuilder**, and that anRT is the following register transfer:

$$*(r101 + 6 + (r102 * 4)) \leftarrow *(r103 + 6 + (r104 * 4)) + *(3000)$$

This transfer would match the template for an ADD on the National that is shown above. The variables would bind in the following way:

- $-1 binds to *(r101 + 6 + (r102 * 4))
- $2 binds to *(r103 + 6 + (r104 * 4))
- $1 binds to *(3000)

These bindings would fail for the ADDQ instruction because it requires that $1 match a quick operand. The ADD instruction would send the **rtAsOperand:for:operandNumber:expressions:**

message to the instruction builder for each operand it needed. This method would be called first with anRT equal to the binding for $1.

The RTL expression *(3000) would be looked up in the operand table and matched with an absolute mode operand. The instruction builder would then send a message to the instruction to make sure that an absolute mode operand is legal for the ADD instruction. This process would be repeated for the binding of $2, and it would match a register-relative indexed operand.

Once the operands bind successfully, the constraint associating $2 with $-1 would be bound. The constraint would bind by sending the **constrainToBe:** message to the RTL expression to which $2 was bound, namely *(r103 + 6 + (r104 * 4)). This binding would succeed and produce a register constraint associating r101 and r103 and a register constraint associating r102 and r104.

## 8.6 Canonicalization

Occasionally the format of a register transfer will need to be altered slightly in order to obtain a match in the instruction table. A good example is the LD instruction for the SPARC shown above. The address of a load on the SPARC is the sum of two registers. This causes a transfer such as $r101 \leftarrow *(r102)$ to fail to match an instruction.

The SPARC has a global register (g0) that always contains 0. Thus the above RTL should be converted into $r101 \leftarrow *(G0\{g0\} + r102)$. The notation {g0} indicates that the register has been preassigned to a physical register.

The difficult question is: Which object should be responsible for performing the conversion? The register transfers and the instruction builder are machine-independent. However, the machine description should not depend on the way register transfers are implemented. This problem is solved by performing double-dispatching between the machine description and the register transfer. The canonicalization methods of **MemoryAccess** and **SPARCMachineDescription** are shown in Figure 8.4.

It turns out that the double-dispatching in the canonicalization methods is useful for many tasks. For example, there is no one-operand test instruction on the National 32000 that would implement $z \leftarrow r101 = 0$. Canonicalization for the National will convert the transfer into $z \leftarrow r101 - 0 = 0$.

Canonicalization is also useful for converting inefficient transfers into more efficient transfers. A clock cycle can be saved on the Motorola 68000 by converting $r101.2 \leftarrow r101.1 << 1$ into $r101.2 \leftarrow r101.1 + r101.1$. The symbol << indicates a logical left shift in RTL.

## 8.7 Handling Multiple Register Categories

There is one more subtlety involved in code generation. On many machines, most notably the Motorola 68000 [Mot85], there is more than one type of register. Some instructions require the use of a certain type of register, others do not. On the Motorola, there are data registers and address registers. Adds, subtracts, and compares can be performed on both register types. Register shifts must be performed on data registers, but memory accesses require address registers.

Each register in the RTL System has an instance variable that represents its category, or type. The category can be assigned either by the process of assigning transfers to instructions or by register allocation. Thus, shift instructions require that the registers in the transfers that they

**MemoryAccess:**

*Protocol for: simplifying*

**canonicalizeFor: machineDescription expressions: expressions**

```
    "double dispatch back to the machine description"

    ↑machineDescription canonicalizeMemoryAccess: self expressions: expressions
```

**SPARCMachineDescription:**

*Protocol for: canonicalizing*

**canonicalizeMemoryAccess: anRT expressions: expressions**

```
    "convert m[a1] --> m[g0 + a1]"

    | address t |
    address ← anRT address.
    (address isRegister or: [address isConstant])
        ifTrue: [t ← expressions
                        BinaryExpressionFirst: (expressions RegisterName: 'G0')
                        operation: '+'
                        second: address].
    t isNil ifFalse: [↑anRT
            newExpressionOfSameType: expressions
            memory: anRT memory
            address: t].

    "Nothing worked so call super"
    ↑super canonicalizeMemoryAccess: anRT expressions: expressions
```

Figure 8.4: Canonicalization of a memory access.

```
bind: bindings expressions: expressions
    "byte sized values cannot be placed in address
     registers"

    register sizef = #BYTE ifTrue: [register mustBeInCategory: #data ifImpossible: [↑false]].
    ↑super bind: bindings expressions: expressions
```

Figure 8.5: Binding a composite instruction.

match be data registers. Adds, subtracts, and compares impose no restrictions on the categories of the registers they match.

This is nontrivial to implement because the opcode, the size of the operands, and the effect on the flags all depend on whether the instruction is performed on a data or address register. Thus, there is a class **M68KCompositeInstruction** that behaves as either a data register arithmetic instruction or an address register arithmetic instruction throughout the process of binding to a register transfer. When code is generated, the instruction checks the category of its register and generates the appropriate instruction.

**M68KCompositeInstruction** represents instructions that contain a register and an operand. The operand can be any of the seven addressing modes on the Motorola. **M68KCompositeInstruction** inherits quite a bit of its behavior from a superclass that implements all behavior for instructions that contain an operand. The **bind:expressions:** method (Figure 8.5) of **M68KCompositeInstruction** binds an **M68KCompositeInstruction** to a register transfer.

```
operationWord: aByteStream
    "if the destination is an address register, generate an
     ADDA/SUBA/CMPA instruction. if the source is an
     immediate value, generate an ADDI/SUBI/CMPI
     instruction. Otherwise, I am just an
     M68KEADataInstruction."

    register category = #pointer ifTrue: [↑self operationWordAsAddress: aByteStream].
    operand isImmediate ifTrue: [↑self asImmediateInstruction operationWord: aByteStream].
    ↑self asEADataInstruction operationWord: aByteStream
```

Figure 8.6: Generating machine language for a composite instruction.

The **M68KCompositeInstruction** method for generating code (Figure 8.6) must deal with a number of possibilities. The instruction could be an immediate instruction, a data instruction, or an address instruction.

# Chapter 9

# The National 32000

The implementation of the National 32000 instruction set is a good example of how a CISC instruction set is handled by the RTL System. There are two class hierarchies that correspond to the National instruction set: the **N32KInstruction** hierarchy and the **N32KOperand** hierarchy. Following is a quick overview of the National instruction set and an overview of the class hierarchies that are used to implement it.

## 9.1 National 32000 Instruction Set

The National 32000 series has a very wide selection of instructions and a large number of addressing modes [Hun87]. Most of the operands are plug-compatible with most of the instructions. Macro-instructions are provided for entering and exiting contexts, string processing, array indexing, and other computations. The following operand categories are implemented on the National:

- gen — general
- reg — register
- quick — small constants
- short — miscellaneous
- imm — constants
- disp — offset from PC

Each instruction specifies an access class for its operands. An operand's behavior depends on its access class. All of these operands except gen are called implied operands because their access class is implied by the operand type. General operands are by far the most common of the operands. General operands can occur as one of several addressing modes, listed below.

- register
- immediate
- absolute

- register relative
- memory space
- memory relative
- external
- top of stack
- scaled index

The indexed addressing mode is used with one of the other addressing modes. An indexed operand contains another operand and modifies its address according to the contents of a register.

Each instruction specifies an access class for each of its operands. The legal access classes for general operands are shown below:

- read
- write
- rmw
- addr
- regaddr

As mentioned earlier, the access class can alter the behavior of the addressing modes. For example, the operand for the register addressing mode is usually in a register, but for an access class of addr the operand is at the memory location to which the register is pointing. Also, stack pops and pushes are implemented using the read and write access classes (see below).

## 9.2 N32KInstruction Hierarchy

The **N32KInstruction** hierarchy is shown below. Almost all instructions are instances of **N32KInstruction**, but there are some special cases that require subclasses.

**N32KInstruction**
  **N32KBranchInstruction**
  **N32KPopPushInstruction**
    **N32KPopInstruction**
    **N32KPushInstruction**

The class **N32KInstruction** implements almost all of the behavior of instructions from the National 32000 instruction set. Instances of **N32KInstruction** have the following instance variables, many of which are inherited from superclasses.

Inherited from **MachineInstruction**:

rtl: A register transfer that represents the operation that this instruction performs.

flags: A register transfer that represents the changes in the flags that are produced by this instruction. This is frequently an empty register transfer.

code: The opcode of this instruction.

Inherited from **CISCInstruction**:

registerConstraints: The constraints on physical register allocation caused by converting the RTL for this instruction into three-address form. This variable is nil until the instruction is bound to a register transfer.

ciscConstraints: The constraints on variables and registers that are caused by converting the RTL for this instruction into three-address form. These constraints produce register constraints when the instruction is bound to a register transfer.

Defined in **N32KInstruction**:

operandList: A list of the operands that this instruction uses.

codeBits: The length of the opcode in bits.

sizeOffset: Most N32K instructions contain two bits that indicate the size of the operation. This instance variable contains the offset into the instruction of the least-significant size bit.

accessList: A list of access classes for the operands that will be components of this instruction. See above.

instructionBuilder: The instruction builder where the operands can be looked up during the binding process.

It is tempting to combine a pair of instance variables when their values are never used simultaneously. For example, ciscConstraints is needed only before the instruction has been bound to a register transfer, and registerConstraints is used only afterwards. The variables accessList and argumentList could be merged in a similar manner. However, it is better to use a different instance variable name for each different item of information stored by the object. This clarifies the code and makes debugging easier, since the history of the object is available in its instance variables.

The class **N32KInstruction** has a protocol for each of the "events" that occur to machine instructions in a table-driven implementation.

### 9.2.1 Table initialization

All National 32000 instructions wind up being initialized by the methods in Figure 9.1. The text representation of the RTL representation of the instruction must be parsed to produce RTL expressions. This code appears in **MachineInstruction** because parsing is necessary for all types of instructions. Also, the RTL representation must be converted into three-address form so that it will match register transfers properly. This code appears in **CISCInstruction** because the CISC machines that have been implemented are two-address.

**CISCInstruction**

**rtl: anRTL flags: flagRTL code: aNum**

*"Set the RTL, flags, and code for this Instruction."*

```
super rtl: anRTL flags: flagRTL.
code ← aNum.
ciscConstraints ← self class convertTo3Address: rtl
```

**MachineInstruction**

**rtl: anRTL flags: aFlags**

```
rtl ← InstructionParser parseTransfer: anRTL.
flags ← InstructionParser parseTransfer: aFlags
```

Figure 9.1: Initializing an **N32KInstruction**.

### 9.2.2 Binding to register transfers

Once a register transfer has successfully matched an instruction, the instruction must bind itself to the register transfer. For the National 32000 (and in fact most CISCs) binding involves trying to bind the operands and trying to bind the constraints.

The pattern variables in an instruction's RTL representation are place-holders for operands. This makes binding the operands a process of looking up the RTL expressions that matched the pattern variables in the operand table. The method that binds an operand of an instruction is **bindArgNum:bindings:expressions:** (Figure 9.2). In the ADD example from above, $2 has been matched to *(r101 + 6 + (r102 * 4)). Thus, when aNum is 2, this method looks up *(r101 + 6 + (r102 * 4)) in the operand table.

After the instruction builder has found an operand that matches the RTL expression, it sends the message **operandOk:operandNumber:** (Figure 9.3) to the instruction to verify that the operand is acceptable. For the National 32000, this simply involves checking that the operand is legal for the necessary access class.

The ciscConstraints are bound by the method **bindConstraints:expressions:** (Figure 9.4). This is inherited from **CISCInstruction**.

### 9.2.3 Generating machine code

A good object-oriented design has made the method to generate machine code (Figure 9.5) is almost identical to the portion of the National manual that describes code generation. Much of the code generation work is done by the operands. There are essentially two portions of the machine code for an instruction: the basic instruction and the extensions. The basic instruction consists of the opcode for the instruction, bits to indicate the size of the operands, and the codes indicating the

```
bindArgNum: aNum bindings: bindings expressions: expressions
    "return true or false to indicate whether the binding was successful"
    "ask the instruction builder for an operand for myself."

    | operand |
    operand ← instructionBuilder
                rtAsOperand: (bindings at: (SavedVariables at: aNum))
                for: self
                operandNumber: aNum
                expressions: expressions.
    operand isNil ifTrue: ["no operand will match."
        ↑false].
    operandList at: aNum put: operand.
    ↑true
```

Figure 9.2: Creating an operand for a **N32KInstruction**.

```
operandOk: anOperand operandNumber: aNum
    "return a boolean to indicate whether anOperand can be used by me.
     I do this by checking the access class with the operand."

    ↑anOperand accessClassOk: (accessList at: aNum)
```

Figure 9.3: Checking an operand's access class.

```
bindConstraints: bindings expressions: expressions
    "return true or false to indicate whether binding was successful"
    "binding a variable constraint could result in failure or the creation
     of one or more registerConstraints. If any of the variable constraints
     fail, the whole binding fails. Otherwise set registerConstraints to be
     the union of the results of binding the variable constraints."

    | newConstraints result |
    newConstraints ← OrderedCollection new.
    ciscConstraints do:
        [:varConstraint |
        result ← varConstraint copy bind: bindings expressions: expressions.
        result isNil
            ifTrue: [↑false]
            ifFalse: [newConstraints addAll: result]].
    newConstraints isEmpty
        ifTrue:
            [registerConstraints ← RTLEmptyConstraint new.
            ↑true].
    newConstraints size == 1
        ifTrue: [registerConstraints ← newConstraints first]
        ifFalse: [registerConstraints ← RTLUnionConstraint components: newConstraints asArray].
    ↑true
```

Figure 9.4: Binding the constraints for an **N32KInstruction**.

types of the operands. The extensions are produced by the operands and depend on the addressing mode.

### 9.2.4 N32KPopPushInstruction

On the National 32000, a stack pop or push is performed by using a general operand in the top-of-stack addressing mode. If the access class is read, the instruction pops the value from the stack. If the access class is write, the instruction pushes the value onto the stack. For other access classes, the value is read from the top of the stack, but the stack pointer is not changed.

A top-of-stack mode operand with an access class of read or write is the only case where an operand affects the template RTL of the instruction. The RTL that would appear in the instruction table for a normal ADD instruction is shown below. For all addressing modes except top-of-stack, the operand would simply be an RTL expression that would replace one of the variables.

$$\$2 \leftarrow \$\text{-}1 + \$1.$$

For a top-of-stack operand with access class read (a pop instruction), the RTL in the instruction table would be (!10 could match any constant):

$$\$2 \leftarrow \$\text{-}1 + *(SP); SP \leftarrow SP + !10.$$

Since the top-of-stack mode operand affects the template RTL of its instruction, it must be handled as a special case. The initialization method for **N32KInstruction** checks for access classes of read or write and creates instances of **N32KPopInstruction** and **N32KPushInstruction**, accordingly. This method is shown in Figure 9.6.

The difference in RTL causes instances of **N32KPopPushInstruction** to require a different binding method (Figure 9.7). The method must bind as usual, except one of the operands has already been bound.

### 9.2.5 N32KBranchInstruction

Generating code for a branch instruction is complicated by the fact that the destination of a branch may not be known when code for the branch instruction is generated. If the destination is not known, zeros are written to the byte stream to leave space for the destination, which is filled in later by the machine description. The method **generateMachineCodeOn:for:** is shown in Figure 9.8

## 9.3 N32KOperand Hierarchy

The **N32KOperand** class hierarchy is shown below. The hierarchy is designed to mimic the classification of the uses of the operands. Some of the implied operands are not implemented, because they appear only in instructions that are not used by TS.

**operationWord: aByteStream**

```
"This method is modeled after p. 5–23 of the 32000 Reference Manual.
 The basic instruction is written least–significant byte first, most-
 significant byte last. It is followed by the index bytes, addressing
 extensions, and implied extensions."

| basicInstruction len opList sizedCode |
basicInstruction ← 0.
len ← 0.

"place the code for each argument in the basic instruction"
operandList do:
    [:arg |
    basicInstruction ← basicInstruction bitShift: arg numBits.
    len ← len + arg numBits.
    basicInstruction ← basicInstruction + arg encoded].

"add the two bits for the size to my code"
sizedCode ← code + (self encodedSize bitShift: sizeOffset).

"add the sized code to the basic instruction"
basicInstruction ← basicInstruction bitShift: codeBits.
basicInstruction ← basicInstruction + sizedCode.
len ← len + codeBits.

"write the basic instruction to the byte stream, least–
 significant byte first."
[len > 0]
    whileTrue:
        [aByteStream nextPut: (basicInstruction bits: 0 thru: 7).
        basicInstruction ← basicInstruction bits: 8 thru: len.
        len ← len – 8].

"write the extensions to the byte stream"
operandList do: [:arg | arg isIndexed ifTrue: [aByteStream nextPut: arg indexByte]].
operandList do: [:arg | arg addressingExtension: aByteStream].
operandList do: [:arg | arg impliedExtension: aByteStream]
```

Figure 9.5: Generating machine language for an **N32KInstruction**.

**rtl: anRTL flags: aFlags code: aNum codeBits: aLen sizeOffset: anOffset accessList: anAccessList**

```
    "check the accessList for access classes of #read or #write.
     A top-of-stack mode operand will behave as a pop for #read
     and a push for #write. These must be handled as special
     cases since the operand changes the format of the instruction."

    1 to: anAccessList size do:
        [:i |
        (anAccessList at: i)
            = #read ifTrue: [N32KPopInstruction
                rtl: anRTL
                flags: aFlags
                code: aNum
                codeBits: aLen
                sizeOffset: anOffset
                accessList: anAccessList
                argNum: i].
        (anAccessList at: i)
            = #write ifTrue: [N32KPushInstruction
                rtl: anRTL
                flags: aFlags
                code: aNum
                codeBits: aLen
                sizeOffset: anOffset
                accessList: anAccessList
                argNum: i]].
    self
        rtl2: anRTL
        flags: aFlags
        code: aNum
        codeBits: aLen
        sizeOffset: anOffset
        accessList: anAccessList
```

Figure 9.6: Creating pop and push instructions.

```
bind: bindings expressions: expressions
    "return true if the addressing modes worked out"
    "iterate through my argument list as normal, except
     handle argument number argumentNum as a special
     case. It does not need to be bound, but we must
     make sure that the amount of stack increment or
     decrement is legal."

    1 to: accessList size do: [:i | i = argumentNum
            ifTrue: [(self isLegalStackExpression: bindings)
                    ifTrue: [↑false]]
            ifFalse: [(self
                    bindArgNum: i
                    bindings: bindings
                    expressions: expressions)
                    ifFalse: [↑false]]].
    ↑self bindConstraints: bindings expressions: expressions
```

Figure 9.7: Binding a pop or push instruction.

**N32KOperand**
- **N32KQuickOperand**
- **N32KShortOperand**
- **N32KGeneralOperand**
  - **N32KIndexedOperand**
  - **N32KNotIndexedOperand**
  - **N32KRegisterModeOperand**
  - **N32KRegRelModeOperand**
  - **N32KTopStackModeOperand**
  - **N32KAbsoluteModeOperand**
  - **N32KImmedModeOperand**
  - **N32KMemRelModeOperand**

**N32KOperand**s have two responsibilities: verify that the RTL expression to which they have been bound is acceptable, and respond to messages sent during code generation.

Almost all operands are quite trivial. A typical operand is **N32KQuickOperand**. It represents an implied quick operand on the National. The class **N32KQuickOperand** has a method for binding and a method for generating machine language.

Operands must respond to the message **accessClassOk:** and return a boolean that indicates whether the operand is legal for the access class. Implied operands are legal in only one access class.

```
generateMachineCodeOn: byteStream for: rtlProgram
    "Generate the machine code necessary to implement this
    instruction, writing the bytes to the byte stream."

    | pos diff |
    byteStream nextPut: code.
    pos ← label codePosition.
    pos isNil
        ifTrue:
            ["leave room for the branch displacement, then come back
            later and fill it in. This is filled in by
            N32KMachineDescription>>generateJumpAddress:at:on:"
            byteStream next: 4 put: 0.
            label addReferencePosition: byteStream position − 4]
        ifFalse:
            ["the displacement can be calculated, so write it to the byte
            stream"
            diff ← pos − byteStream position.
            N32KMachineDescription write4ByteDisp: diff to: byteStream]
```

Figure 9.8: Generating machine language for a branch instruction.

```
accessClassOk: anAccessClass
    ↑anAccessClass = #quick
```

Figure 9.9: Checking the access class for a quick operand.

```
encoded
    "The number is included directly in the basic instruction."

    ↑value
```

Figure 9.10: Encoding a quick operand.

Quick operands require the quick access class. The **accessClassOk:** method of **N32KQuickOperand** is shown in Figure 9.9.

All operands must also implement some methods that are required for generating machine language. Most of the methods have to do with indexing and/or extensions, and are ignored by most operands. All operands do implement **encoded**, which returns a code for the operand that will be placed in the instruction. For a quick instruction, the encoding is just the quick value. The **encoded** method of **N32KQuickOperand** is shown in Figure 9.10.

# Chapter 10

# Scheduling

Register transfers in an RTL program have a partial order caused by data dependencies. Scheduling assigns a total ordering to the transfers. A good way to think about scheduling is that it converts an RTL program from a DAG to a sequence.

The scheduling phase of the RTL System has three purposes. First, it combines register transfers together to form more complicated transfers that correspond to machine instructions. Second, it orders the transfers within a basic block so that register lifetimes are minimized. Third, it orders the transfers for RISC machines so that pipeline interlocks are avoided.

Scheduling is divided into several parts: combining, scheduling, and filling branch delay slots (on RISC machines). Each part is represented by an object. Although algorithms are usually operations on objects, making these algorithms objects has a number of advantages. It is easier to "mix and match" algorithms, e.g. to leave out filling branch delay slots on machines with no branch delay slots. Also, algorithms that are objects can be customized using inheritance. For example, the scheduling algorithm is extended for RISC machines to minimize pipeline interlocks due to memory accesses.

## 10.1 Combiner

The combining algorithm builds complex transfers from simple ones. **Combiner** is very good at generating complex memory accesses, creating post-increments and pre-decrements, combining compares and subtracts, and generating some macro instructions. It works closely with the machine description in order to determine whether a transfer that it has created corresponds to a machine instruction.

The interface between the combiner and the machine description can be described as follows. When the combiner produces a new register transfer, it asks the machine description to produce an instruction that corresponds to the transfer. If the machine description produces an instruction, the combiner assigns the instruction to the transfer and adds the transfer to the RTL program.

**Combiner** tries all possibilities for combining until its search limit is reached. Trying all possibilities guarantees that no combinations are missed, so the RTL System performs perfect peephole optimization. However, combiner is fast, taking less than a tenth of the time of optimization and less than half of the time of register allocation.

The combiner is especially useful for CISC machines, but some combining can frequently be done for RISC machines. Take the following three transfers for example:

$$r102 \leftarrow r101 + 10.$$
$$r104 \leftarrow r102 + r103.$$
$$r105 \leftarrow *(r104).$$

A combiner for the Motorola 68000 will combine these three transfers into the following transfer:

$$r105 \leftarrow *(r101 + r103 + 10).$$

A combiner for the SPARC will produce these two transfers:

$$r102 \leftarrow r101 + 10.$$
$$r105 \leftarrow *(r102 + r103).$$

For the Motorola, combining the last two transfers or the first two transfers does not result in a legal instruction. The combiner had to try to combine three transfers even when no subset of the three had successfully combined. This depth of searching is never necessary for most RISC machines.

This difference in searching behavior is taken care of by an instance variable of **Combiner**, searchDepth. It contains the maximum number of transfers that the scheduler will attempt to combine when no subset of the transfers has previously combined successfully. For the Motorola, searchDepth is 3. For most RISC machines, searchDepth is 2. When transfers are successfully combined, the depth count is reset; thus there is no limit on the number of combinations that will be tried when the intermediate transfers that are created correspond to machine instructions.

The combiner tries combinations for one transfer at a time. For each transfer, the combiner iterates over its "window," which is the set of transfers that are likely to combine successfully with the transfer. The method **tryCombinationsFor:** (Figure 10.1) tries to combine a transfer with each transfer in its window.

The **tryToCombine:with:** method (Figure 10.2) combines two register transfers and asks the machine description for an instruction that matches them.

The **candidatesForCombinationWith:do:** method (Figure 10.3) iterates over a transfer's window.

When the combined transfer fails to match an instruction, the **tryMoreCombinations** (Figure 10.4) method is executed. This method checks the searchDepth parameter to determine whether to give up the search for a successful combination.

## 10.2 Scheduler

The class **Scheduler** implements the scheduling algorithm. It is responsible for reducing register lifetimes and handling the dependencies between register transfers. The algorithm is based on list scheduling [Cof76], [Tie89].

The scheduling algorithm for a flow node can be described as follows: At any given time, some set of transfers is ready. A ready transfer is one whose supporters have been scheduled previously. The scheduler assigns a priority to each of the ready transfers and then schedules the one with the highest priority. The priorities are assigned so that register lifetimes are reduced. This process is repeated until all the transfers in the flow node have been scheduled.

The scheduler gives a very high priority to transfers whose supporters have been scheduled recently. This causes the scheduler to follow a single path in the dependency graph until it completes.

```
tryCombinationsFor: anRT
    "Try combining anRT with other transfers to form
    new transfers corresponding to machine instructions."

    usedRTs ← OrderedCollection with: anRT.
    self windowFor: anRT do:
        [:windowRT |
        "try combining anRT with an RT from its window.
        the result is stored in the instance variable combinedRT"
        (self tryToCombine: anRT with: windowRT)
            ifTrue:
                [usedRTs addLast: windowRT.
                self replaceUsedRTs.
                ↑self tryCombinationsFor: combinedRT].
        ↑self tryMoreCombinations: 3]
```

Figure 10.1: Trying combinations for a register transfer.

If no ready transfers have a recently scheduled supporter, the scheduler gives a higher priority to the one with the most dependents. This increases the choices that the scheduler has in the future.

If a recently scheduled transfer has more than one dependent that is ready, the scheduler will schedule the one with the fewest dependents. This is because the value that all of the dependents use will have to be stored while each one but the last is computed. Scheduling the most expensive one last minimizes the amount of time that the value must be stored.

The method **supporterPriorityFor:** (Figure 10.5) calculates a component of the priority for a register transfer. The supporter priority of a transfer is the sequential number of its most recently scheduled supporter. For example, if three instructions have been scheduled, any transfer that depends on the third one will have a supporter priority of 3.

The method **priorityFor:** (Figure 10.6) calculates the total priority of a transfer. The supporter priority is multiplied by 100, since the other priorities exist only to "break ties" that occur for supporter priority.

If two transfers have the same most recently scheduled supporter, their supporter priorities will be positive and equal. This tie is broken by subtracting the total number of dependents in this node for each transfer from its priority. This will cause the one with the fewest dependents to have a higher priority.

If no ready transfers have a scheduled supporter, all supporter priorities will be zero. This tie is broken by setting the priority to be the total number of dependents in this node for each transfer. This will favor transfers on which a large number of other transfers depend.

These priorities are similar, but not identical to traditional scheduling algorithms [SU70]. Scheduling is more complicated in the RTL System because the level of optimization produces graph-structured dependencies. Without optimization the dependency graph would always be a

```
tryToCombine: rt1 with: rt2
    "Combine rt1 with rt2 to create a
    new register transfer having the same effect as
    the two separate register transfers. rt1 could be
    the result of combining the members of usedRTs.
    Return a boolean to indicate whether the instruction
    corresponds to a machine instruction."

    | list expressions instruction |
    expressions ← rtlProgram expressions.
    list ← OrderedCollection new: rt1 transferCount + rt2 transferCount.

    "Add each transfer in rt1 to the transfer list,
    using any assignments found in rt2 as renames. Care is
    taken to form the combined transfer list from copies of
    the originals, so that when the originals are
    destroyed, the combined transfer list is unaffected."
    rt1 transfersDo: [:rt | "Note that replaceRegister: returns a copy."
        list add: (rt replaceRegister: rt2 expressions: expressions)].

    "Add everything if rt2 has a flow dependent not in
    usedRTs (i.e., sets something needed after me), or if rt1 sets
     a bit register. Otherwise, only add the transfers that set a
    memory location."
    (rt1 setsFlag not and: [rt2 flowDependents isSubsetOf: usedRTs])
        ifTrue: [rt2 transfersDo: [:rt | rt destination isRegister ifFalse: [list add: rt copy]]]
        ifFalse: [rt2 transfersDo: [:rt | list add: rt copy]].
    combinedRT ← RegisterTransfer of: list.
    combinedRT topNumber: rt1 topNumber.
    instruction ← rtlProgram machineDescription rtAsInstruction: combinedRT.
    instruction notNil
        ifTrue:
            [combinedRT instruction: instruction.
            ↑true]
        ifFalse: [↑false]
```

Figure 10.2: Combining two register tranfsers.

```
candidatesForCombinationWith: anRT do: aBlock
    "The heuristic used is that it is likely to be worthwhile to try
     to combine anRT with its supporters and the dependents of
     its supporters."

    | flowSupporters |
    flowSupporters ← anRT flowSupporters.
    flowSupporters do: [:supporter | (anRT canCombineWith: supporter)
            ifTrue: [aBlock value: supporter]].
    flowSupporters do: [:supporter | supporter flowDependents do: [:rt | rt == anRT ifFalse: [(anRT can-
CombineWith: rt)
                    ifTrue: [aBlock value: rt]]]]
```

Figure 10.3: Determining a register transfer's window.

```
tryMoreCombinations: depth
    "combinedRT wasn't an instruction, so try combining it
    with a more transfers. Conceptually, we are iterating over
    combinedRT's window. However, its dependency
    information hasn't been initialized yet (replaceUsedRTs
    does that), so we use (usedRTs last)'s window instead."

    depth > searchDepth ifTrue: [↑nil].
    self windowFor: usedRTs last do:
        [:windowRT |
        (self tryToCombine: combinedRT with: windowRT)
            ifTrue:
                [usedRTs addLast: windowRT.
                self replaceUsedRTs.
                ↑self tryCombinationsFor: combinedRT].
        ↑self tryMoreCombinations: depth + 1]
```

Figure 10.4: Trying combinations for a register transfer.

**supporterPriorityFor: anRT**

```
    "This priority is high for RTs that have supporters that are at
    the end of scheduled (i.e., their supporters were scheduled
    recently). The reasoning behind this priority is to schedule
    anRT as soon as possible after its supporters and minimize
    the lifetime of the registers defined by anRT's supporters.
    This priority should be in effect only when all of the supporter's
    dependents are in this block. In other words, if the register
    that anRT's supporter defines lives outside this flow node,
    there is no reason to try to schedule anRT early, since the
    reason to try to schedule anRT early in the first place was to
    try to minimize the lifetime of the register that anRT's
    supporter defines."

    | priority |
    priority ← 0.
    anRT flowSupporters do: [:rt | ((scheduled includes: rt)
            and: [rt definedRegisterDiesInThisFlowNode])
            ifTrue: [priority ← priority max: (scheduled indexOf: rt)]].
    ↑priority
```

Figure 10.5: Determining a transfer's supporter priority.

```
priorityFor: anRT
    "schedule based on the following priorities, in this order:
    1. recency of most recent flow supporter.
    2. total number of flow dependents -- if anRT has no supporters that have already
     been scheduled: the more flow dependents, the higher the priority. if anRT
     has supporters that have already been scheduled: the more flow dependents,
     the lower the priority."

    | priority |
    priority ← self supporterPriorityFor: anRT.
    priority > 0
        ifTrue:
            ["anRT has previously scheduled flow supporters"
            priority ← priority * 100 − (self dependentsClosureFor: anRT) size.
            ↑priority]
        ifFalse:
            [priority ← (self dependentsClosureFor: anRT) size.
            ↑priority]
```

Figure 10.6: Determining a transfer's priority.

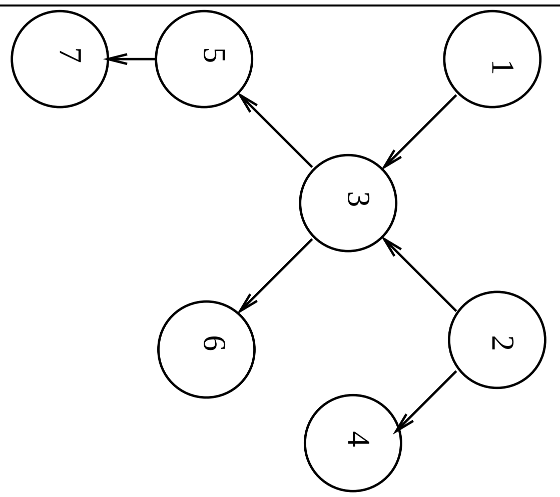



Figure 10.7: Example dependence graph for scheduling.

tree, which makes scheduling more straightforward.

## 10.3 Scheduling Example

The graph shown in Figure 10.7 illustrates how the priorities affect the scheduling process. In this example, let (4) be a memory read and (6) be a memory write. This means that (6) is an anti dependent of (4).

The ready transfers are initially (1) and (2). The priority for each of these is the size of the closure of their flow dependents. Transfer (2) has one more member in its flow dependents closure, so it is scheduled first. After (2) is scheduled, (4)'s priority becomes very high since its supporter was scheduled recently. It is scheduled next. Then (1) and (3) are scheduled because on each of the next two iterations, there is only one ready transfer. After (3) is scheduled, (5) and (6) would have equally high supporter priorities since they both depend on the most recent transfer. However, (5)'s priority is reduced since it has a dependent. Thus (6) is scheduled, followed by (5) and (7).

Thus, the final schedule is (2), (4), (1), (3), (6), (5), (7).

# Chapter 11

# RISC Scheduling

RISC machines can be characterized by their exposure of low-level parallelism to the software. RISC machines are pipelined, which means that each instruction goes through independent computation stages. The machine has the ability to process each stage of the pipeline concurrently. A pipeline interlock occurs when an instruction at one stage must wait for another instruction to complete a stage. On the SPARC, a pipeline interlock occurs when a memory read instruction is immediately followed by an instruction that uses the value read from memory. The instruction must wait for memory to be accessed.

Another characteristic of RISC machines is the use of delayed branches. Conventional branches transfer control immediately. A delayed branch transfers control after the instruction that follows the branch instruction is executed. The instruction that follows the branch instruction is said to be in the branch delay slot.

A RISC scheduler has two responsibilities in addition to those of a scheduler. First, the instructions within a flow node should be ordered so that the number of pipeline interlocks is minimized. Second, branch delay slots should be filled. It turns out that there is little in common between these two responsibilities. Avoiding pipeline interlocks involves flow dependency analysis. Filling branch delay slots involves control analysis.

There is a trade-off in instruction scheduling for RISC machines. One goal is to avoid as many pipeline interlocks as possible. Trying to avoid interlocks usually results in scheduling a great deal of computation in parallel. All the load instructions would be scheduled first, then all the computation, then all the stores. This parallelism creates a great deal of overlap between register lifetimes, which creates register spilling, which causes slow program execution.

The best way to reduce register lifetimes is to perform one series of computation at a time. The instruction sequence to produce minimal register lifetimes would have each load occur just before its register is used. Unfortunately, this sequence is likely to create a great number of pipeline interlocks on most RISC machines.

Thus, the hard part of RISC scheduling is avoiding pipeline interlocks without causing a great number of register spills. In fact, it has been proven that scheduling for a pipelined processor is an NP-complete problem [Gro83]. Since RISC scheduling is so difficult, the algorithm implemented in **RISCScheduler** does not try to produce a perfect schedule in all cases. Instead, it produces a good schedule in almost all cases.

The approach implemented by **RISCScheduler** places a great deal of emphasis on minimizing register lifetimes. The scheduler will alter the schedule only for the purpose of avoiding an interlock if the alteration guarantees that an interlock will be avoided. This can be justified by the fact that a register spill is much more costly than a pipeline interlock. A pipeline interlock wastes only one or two clock cycles, but a spill involves a memory read and/or write, which requires many more cycles.

## 11.1 RISCScheduler

The class **RISCScheduler** is a subclass of **Scheduler** and inherits most of its behavior. There are two ways in which the schedule can be modified in order to avoid an interlock, and these two ways are the major modifications that **RISCScheduler** makes to its superclass. First, the RISC scheduler will not schedule an instruction that causes an interlock with the previous instruction unless there is no alternative. Second, objects of class **PotentialInterlock** add and remove instructions from the pool of available instructions for the purpose of avoiding interlocks.

The most straightforward way to avoid pipeline interlocks would be to incorporate an instruction's likelihood of causing an interlock into its priority. This approach is implemented by [Tie89]. However, it turns out to be quite difficult to alter the priorities enough to guarantee that interlocks will be avoided in some situations without causing spilling in other situations. The approach implemented in **RISCScheduler** separates a transfer's likelihood of causing an interlock from its priority.

The most basic way that the **RISCScheduler** avoids interlocks is by checking each transfer that it is about to schedule to make sure that the transfer will not cause a pipeline interlock. It saves any instructions that would have been scheduled if they had not caused an interlock. The priorities of these instructions are raised for the next iteration through the scheduling loop. This is done to insure that the scheduling process is not completely sidetracked by an interlock. It should be only temporarily sidetracked.

Interlocks are also avoided by the intervention of potential interlock objects. These objects add and remove transfers from the schedule for the purpose of preventing pipeline interlocks. They also occasionally insert transfers into the schedule.

From the scheduler's point of view, potential interlocks affect scheduling in somewhat the same way that dependencies do. However, potential interlocks are not as important as dependencies. Scheduling a transfer in disregard of a dependency would cause the program to run incorrectly. Scheduling a transfer that had been removed by a potential interlock would only cause an interlock; program correctness would not be affected.

For this reason, **RISCScheduler** maintains three pools of transfers. There is a pool of transfers that are ready to be scheduled. There is a pool of transfers that can be scheduled without affecting program correctness, but scheduling one of them would probably cause a pipeline interlock. And there is a pool of transfers that cannot be scheduled under any circumstances.

The method **chooseUsingPrioritiesFrom:** (Figure 11.1) selects a transfer to schedule from a collection of ready transfers. It sorts the transfers and selects the one with the highest priority that does not cause a pipeline interlock. The transfers that would have been scheduled had they not caused an interlock are saved so their priorities can be raised on the next iteration through the scheduling loop.

## 11.2 PotentialInterlock

Objects of class **PotentialInterlock** are created by the RISC scheduler in order to avoid most pipeline interlocks. Each potential interlock represents two transfers that could, if they were scheduled in immediate succession, cause a pipeline interlock.

A potential interlock is responsible for keeping track of the transfers that are available for scheduling that could fill the slot between the two transfers. At any point during scheduling, if there is only one possibility left, the potential interlock will remove the fill transfer from scheduling. Once its first transfer has been scheduled, it will insert its fill transfer into the schedule and avoid an interlock.

Potential interlocks remove transfers from scheduling by sending the message **instruction:claimedBy:** (Figure 11.2). Since two potential interlocks could try to remove a transfer at the same time, a request queue is maintained by the scheduler. Thus, a request by a potential interlock to remove a transfer could be thought of as a request to claim the instruction. The scheduler decides whether this claim is fulfilled or denied.

When a potential interlock has successfully claimed a transfer, it waits until the first of the two transfers for which it is responsible is scheduled. Immediately afterwards, it inserts the transfer that it has claimed into the schedule. Thus, when a transfer is scheduled, the scheduler must check with each potential interlock to see if it needs to insert a transfer into the schedule. If an interlock does insert a transfer, the scheduler must check again with the interlocks to see if another transfer needs to be inserted. This is implemented by **scheduleWithFillers:** (Figure 11.3). The method of potential interlock that returns a transfer if one needs to be inserted is **fillAfter:** (Figure 11.4).

There are several scheduling events that require updating of potential interlocks. Since it is possible that a potential interlock could initially have just one fill possibility left, the scheduler notifies the potential interlocks when scheduling is about to begin. It also notifies the potential interlocks when a transfer is scheduled, removed from scheduling, or added to scheduling.

The **determineFillPossibilitiesIn:** message (Figure 11.5) is sent to each potential interlock by the scheduler when scheduling is about to begin. The potential interlocks determine which transfers could fill their delay slot.

```
chooseUsingPrioritiesFrom: readyInstructions
    "Choose the instruction with the highest priority, that does not
    cause an interlock."

    | prioritizedInstructions |
    readyInstructions isEmpty ifTrue: [self error: 'cannot choose from empty collection'].
    readyInstructions size = 1 ifTrue: [↑self scheduleWithFillers: readyInstructions first].
    prioritizedInstructions ← readyInstructions asSortedCollection: [:a :b | (self priorityFor: a)
                    > (self priorityFor: b)].

    "if scheduled is empty, this is the first instruction and we can't have an
     interlock"
    scheduled isEmpty ifTrue: [↑self scheduleWithFillers: prioritizedInstructions first].
    lastTimeInterlocks ← OrderedCollection new.

    "try to avoid scheduling an instruction that will cause an interlock. Iterate through
     the collection in priority order and schedule the first one that does not cause an
     interlock."
    prioritizedInstructions do: [:rt | (scheduled last instruction causesInterlockWhenFollowedBy: rt instruction)
            ifTrue: [lastTimeInterlocks add: rt]
            ifFalse: [↑self scheduleWithFillers: rt]].

    "all ready instructions cause an interlock"
    lastTimeInterlocks ← OrderedCollection new.
    self scheduleWithFillers: prioritizedInstructions first
```

Figure 11.1: Selecting the next instruction to schedule.

```
instruction: anRT claimedBy: aPotentialInterlock
    "a potential interlock wants to use anRT to fill its slot. place the
     request on the queue."

    claimRequests addLast: (Association key: aPotentialInterlock value: anRT)
```

Figure 11.2: Scheduler methods for claiming and unclaiming.

```
scheduleWithFillers: anRT
    "schedule anRT and any fillers that may be generated by the
     potential interlocks."
    "a potential interlock may want to schedule a filler after anRT.
     Another potential interlock may want to schedule another
     filler after the first filler, etc. Fillers is an OrderedCollection
     of these fillers."

    | fillers i |
    unclaimed remove: anRT.
    fillers ← self determineFillersFollowing: anRT.
    self scheduleRT: anRT.

    "some fillers may be taken out of scheduling by an
     incidental dependency. so make sure that
     nowScheduling includes a filler before we schedule
     it."
    i ← 1.
    [i <= fillers size and: [(nowScheduling includes: (fillers at: i))
            and: [self rtIsReady: (fillers at: i)]]]
        whileTrue:
            [self scheduleRT: (fillers at: i).
            i ← i + 1].
    potentialInterlocks do: [:pi | pi justClaimed: anRT].
    1 to: i − 1 do: [:index | potentialInterlocks do: [:pi | pi justClaimed: (fillers at: index)]].
    self processRequests
```

Figure 11.3: Scheduling a register transfer.

```
fillAfter: anRT
    anRT = firstRT
        ifTrue:
            [firstHasBeenScheduled ← true.
            requestOk ifTrue: [↑fillRT]].
    ↑nil
```

Figure 11.4: Scheduling a fill transfer to avoid an interlock.

**determineFillPossibilitiesIn: someInstructions**

*"determine the instructions in someInstructions that could fill my delay slot. If there is only one, claim it."*

```
fillPossibilitiesLeft ← someInstructions select: [:rt | self rtCouldFillSlot: rt].
fillPossibilitiesLeft size = 1 ifTrue: [self claimFiller]
```

Figure 11.5: Initializing the collection of fill possibilities.

**justClaimed: anRT**

*"anRT has just been claimed by a potential interlock (not me). update my fillPossibilitiesLeft."*

```
firstHasBeenScheduled ifTrue: [↑self].
anRT = fillRT ifTrue: ["my fillRT was taken by someone else"
    self requestDenied].
fillPossibilitiesLeft remove: anRT ifAbsent: [].
fillPossibilitiesLeft size = 1 ifTrue: [self claimFiller]
```

Figure 11.6: Updating the collection of fill possibilities.

Once scheduling starts, whenever a transfer is scheduled or claimed by a potential interlock, the scheduler notifies each of the potential interlocks via the **justClaimed:** message (Figure 11.6). They may respond by claiming another instruction.

## 11.3 RISC Scheduling Example

The following example illustrates the behavior of RISCScheduler and PotentialInterlock. It is taken from [Tie89], but two transfers that could be used to fill a delay slot are left out to make life more difficult for the scheduler. The RTL code is shown in Figure 11.7. The dependency graph, showing only flow dependencies, appears in Figure 11.8.

During initialization, an instance of potential interlock is created to represent the potential interlock between each of the following pairs of register transfers: (1) and (2), (3) and (4), and (6) and (7). No anti dependencies affect the schedule: transfer (9) could write on the memory location used by (1), (3), and (6), but it is guaranteed to be scheduled after them anyway due to flow dependencies.

Initially, the fill possibilities left for each of the potential interlocks are the following:

1) $r103.1 \leftarrow *(r100.1 + 6)$.
2) $r103.2 \leftarrow r103.1 << 24$.
3) $r104.1 \leftarrow *(r100.1 + 4)$.
4) $r104.2 \leftarrow r104.1 << 16$.
5) $r103.3 \leftarrow r104.2 | r103.2$.
6) $r105.1 \leftarrow *(r100.1 + 2)$.
7) $r105.2 \leftarrow r105.1 << 8$.
8) $r103.4 \leftarrow r103.3 | r105.2$.
9) $*(r101.1) \leftarrow r103.4$.

Figure 11.7: Example RTL code for RISC scheduling.

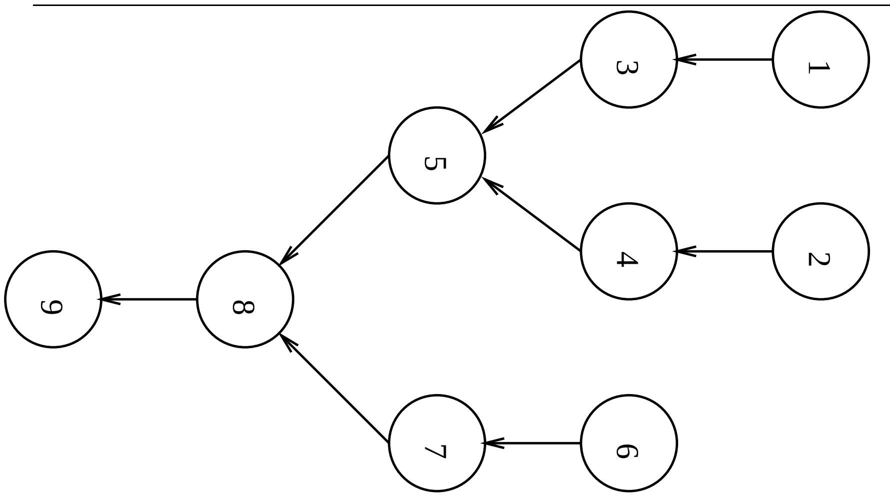



Figure 11.8: Example flow graph for RISC scheduling.

| | |
|---|---|
| PI(1)-(2): | (3),(4),(6),(7) |
| PI(3)-(4): | (1),(2),(6),(7) |
| PI(6)-(7): | (1) through (5) |

Initially, (1) and (3) have the highest priority since they have the most dependents. The scheduler arbitrarily chooses (1). On the next iteration, (2) has the highest priority since it depends on (1). However (2) causes an interlock with (1), so (3) is scheduled.

At this point (4) would have a slightly higher priority than (2) since its supporter was scheduled more recently. However (2)'s priority is increased because it would have been scheduled last iteration had it not caused an interlock. This causes (2) and (4) to have the same priority; (2) is arbitrarily chosen. After that (4) has the highest priority and it is scheduled.

Ignoring the potential interlocks, the priorities would force the rest of the schedule to be (5), (6), (7), (8), (9). Notice that this schedule includes a pipeline interlock between (6) and (7). The potential interlock objects are what prevents the schedule from causing a pipeline interlock.

After each transfer is scheduled, each potential interlock object removes the transfer from its set of fill possibilities for its interlock. Thus, after (1), (3), (2), and (4) have been scheduled, (5) is the only fill possibility remaining for the potential interlock that represents the interlock between (6) and (7). So this potential interlock removes (5) from scheduling.

Once (5) has been removed from scheduling, the only ready transfer is (6), and it is scheduled next. After (6) is scheduled, the potential interlock that represents the interlock between (6) and (7) inserts (5) into the schedule. The scheduler then schedules (7), (8), and (9).

Thus, the final schedule is (1), (3), (2), (4), (6), (5), (7), (8), (9). All potential pipeline interlocks were avoided.

## 11.4 RISCBranchScheduler

In the RTL System, every flow node ends with at least one register transfer that represents a transfer of control. It turns out that it is not necessary to generate branch instructions for most of the branch transfers. This is because the code for the flow nodes to which these transfers branch appears immediately after the code for the flow node in which these transfers appear. In other words, if the branch instruction were generated, it would be a branch to the next instruction.

Therefore, whether a branch transfer corresponds to a branch instruction or not depends on the structure of the flow graph. Thus, branch scheduling must be done after any phase of compilation that could affect the flow graph of the RTL program. In fact, branch scheduling is the last optimization performed on an RTL program before machine code is generated.

The branch scheduler schedules branches one flow node at a time. For each flow node, the scheduler determines which branches are "real" (i.e., which branch transfers will correspond to branch instructions). For each "real" branch transfer, the scheduler tries to find a transfer that can fill the branch's delay slot. This process is implemented by **scheduleBranches** (Figure 11.9).

There are three conditions that a transfer must meet in order to fill a branch delay slot. First, the transfer must correspond to a machine instruction. Transfers such as phi-assignments have logical meaning but no machine code is generated for them. Second, the transfer must have no dependents in its flow node. Otherwise, putting the transfer in the delay slot would move it after its

```
scheduleBranches
    "Branch scheduling is done by traversing the nodes and
     determining which jumps are real' (i.e., which jumps will
     correspond to branch instructions in the machine code),
     and trying to fill delay slots for these instructions."

    | realJumps totalSlots |
    currentNode ← nil.
    Transcript show: 'Scheduling branches.'; cr.
    totalSlots ← 0.
    rtlProgram
        rtNodesDo:
            [:node |
            currentNode isNil
                ifFalse:
                    [realJumps ← currentNode realJumpsForSuccessor: node.
                    totalSlots ← totalSlots + realJumps size.
                    self fillDelaySlotsFor: realJumps].
            currentNode ← node].
    realJumps ← currentNode realJumpsForSuccessor: nil.
    totalSlots ← totalSlots + realJumps size.
    self fillDelaySlotsFor: realJumps.
    self processClaims.
    Transcript show: numSlotsFilled printString , ' out of ' , totalSlots printString , ' branch delay slots
filled.'; cr
```

Figure 11.9: Scheduling RISC branches.

```
findBranchDelayFiller
    "look for an instruction in aNode that could fill a branch delay
     at the end of the node. If the instruction is going to be moved
     we need to check to make sure that moving it won't trash a flag
     or memory value."

    | movingPast cantMove |
    movingPast ← OrderedCollection with: currentNode lastTransfer.
    currentNode block asOrderedCollection
        reverseDo:
            [:rt |
            (rt correspondsToInstruction and:
                [rt dependentsInSameNode isEmpty and:
                [self canMove: rt below: movingPast]])
                ifTrue: [↑rt].
            movingPast add: rt].
    ↑nil
```

Figure 11.10: Locating a transfer to fill a delay slot.

flow dependents. Third, moving the transfer must not cause a flag or memory value to be trashed. All of these conditions are checked by **findBranchDelayFiller** (Figure 11.10).

The method **canMove:below:** (Figure 11.11) checks to make sure that moving a transfer will not cause a flag or memory value that it sets or uses to be trashed.

*Protocol for: filling slots*

**canMove: anRT below: rtCollection**

```
    "Return true if moving one transfer below a collection of other transfers
    (it is currently before those transfers) will not cause a flag or memory
    value to be overwritten. Note that this method is quite conservative,
    because the dependency information assumes just about every transfer
    sets flags, which may not be true."

    | dependents |
    dependents ← anRT allDependents.
    rtCollection do: [:rt | (dependents includes: rt)
            ifTrue: [↑false]].
    ↑true
```

Figure 11.11: Checking flag and memory dependencies.

# Chapter 12

# Conclusion

This section has described how a highly optimizing code generator can be made extremely portable. This is done through the use of an object-oriented framework which divides the system into a well-defined set of machine-independent classes and a well-defined set of machine-dependent classes. Also, optimizations that depend on the type of machine, such as code reorganization for RISC machines, are viewed as part of the portable scheduling phase of compilation.

# Part III

# Register Allocation

# Chapter 13

# Introduction

The past decade has seen the advent of new and sophisticated approaches toward the problem of register allocation. Early techniques relied upon reference counts to assign frequently used variables to registers. In practice this meant that efficient allocation could only be performed within basic blocks. Modern graph coloring algorithms, on the other hand, can allocate registers for entire procedures.

The essential problem of register allocation is to map a large number of symbolic program variables onto a machine's small physical register set. To do this requires finding and solving the interference relationships between the variables. In [CAC*81] it was suggested that this problem could be modeled as a graph coloring problem. Nodes in the graph represent variables, temporal conflicts among variables become edges between nodes, and the physical registers are the colors.

The goal of allocating scalar variables in registers is to reduce memory delays caused by memory load and store operations.[1] Fast memory caches are another technique used to solve the same problem. Caches differ from registers in that they work dynamically in hardware. Register allocation is performed statically before a program executes. Static allocation can only approximate perfect register allocation since loops and branch frequencies cannot be determined. Perfect allocation can be performed within basic blocks. Unfortunately a global solution composed of optimal local solutions is likely to be far from optimal due to the program control structures. One approach taken by Wall [Wal86] combines execution profiling with link-time allocation and seems to achieve good results.

A graph coloring approach to the allocation problem has several advantages. Graph coloring makes global allocation possible and provides a uniform and systematic method of dealing with idiosyncrasies of machine instructions. Perhaps most importantly it offers a simple conceptual framework. Simple frameworks tend to lead toward elegant, efficient implementations and this seems to be the experience reported in the literature.

The weaknesses of graph coloring crops up when the graph cannot be colored. When there is a shortage of physical registers, certain symbolic registers must be allocated in memory. Allocating symbolic registers to memory is called "spilling" and it eventually results in a colorable graph. Spill selection is hard because there are two important considerations. Registers should be spilled first if they are infrequently used. However, it is also important to spill a registers that is part of the conflict's bottleneck, so removing it will contribute substantively toward a colorable graph.

[1] we shall not consider complicating factors such as pipeline interlocks and RISC delay slots. See Hsu 87 for details

Spilling is also difficult because spilling a node is not exactly the same as removing it from the conflict graph. Spilling a lifetime places it in memory, but it will still require register resources as it is loaded and stored. Spilling should thus be viewed as breaking a lifetime into small pieces, and not as eliminating it from the conflict graph.

The problem of spilling has lead to two different graph-based register allocation algorithms. The original paper by Chaitin [CAC*81] was refined in [Cha82] and more recently by others [BGG*89], [BCKT89], and [Nic90]. Meanwhile [CH84] adapted the graph coloring model to use a very different method of spilling. [LH86] presents his implementation of a Chow type allocator.

This section will examine aspects of the various allocation schemes and describe their implementation in the RTL System.

# Chapter 14

# Register Allocation with Graph Coloring

The unit of register allocation is the *lifetime*. A lifetime is composed of symbolic registers that should all be assigned to the same storage location, be that a register or a memory spill location. A lifetime is said to *conflict* with another lifetime if they are ever live simultaneously. Conflicting lifetimes can never be assigned to the same physical register.

Register allocation becomes a graph coloring problem by creating a *conflict graph* using lifetimes as nodes and conflicts between lifetimes as edges. Assigning a color to node Y requires that the color chosen is not already assigned to a neighbor of Y in the graph. This is exactly the same problem as that of assigning a register to node Y such that no overlapping lifetime has been assigned the same register.

Register allocation becomes a graph coloring problem by creating a *conflict graph* using lifetimes as nodes and conflicts between lifetimes as edges. To assign a register to node Y requires that the register assigned is not already assigned to a neighbor of Y in the conflict graph. This is exactly the same problem as that of assigning a color to node Y.

This section refers to lifetimes sometimes as *nodes* or *live ranges*. Similarly, *coloring* a node means the same thing as *assigning* that lifetime to a register.

## 14.1 Coloring the Conflict Graph

The task of finding a coloring for the conflicts graph is performed using a simple heuristic technique. A general solution to finding a minimal graph coloring is an NP-complete problem. This would pose unacceptable time bounds for any production compiler. By instead using a heuristic to color the graph we can find a near optimal coloring quickly. Chaitin introduced a heuristic technique in [CAC*81] and subsequent experience indicates that it is highly effective. This gives coloring a linear bound to the number of nodes.

A coloring of the conflict graph $G$ with $R$ number of colors (registers) is attempted as follows. For each node $Y$ in the conflict graph we define *degree(Y)* to be the number of neighboring nodes that conflict with it. Conflicting nodes are connected by an edge in the graph, so the degree is simply a count of the edges incident upon node Y. We observe that if the degree is less than the number of colors, this node can be trivially colored once its neighbors have been given colors. Such nodes are considered to be *unconstrained* and are removed from the graph and placed on a list of unconstrained nodes, which will be colored later.

Removing an unconstrained node from the graph also removes all of its conflicts. This may cause nodes which were earlier constrained to now be unconstrained. The process of finding and removing all unconstrained nodes is thus repeated until the graph does not contain any unconstrained nodes or the graph has had all of its nodes removed and is now empty. If the graph has been reduced to the empty graph, we have found a coloring for it.

The actual coloring is performed by reconstructing the graph in the reverse order in which nodes were removed. The last node is given any arbitrary color. Nodes added back into the graph in reverse order can then be given any color not already in use by a neighboring node.

When the coloring process is stuck because there are no more unconstrained nodes to remove, spill code is introduced to assign a selected lifetime to memory. Spilling a node removes it from the conflict graph and eliminates its conflicts. Then, another attempt is made to color the graph and if this too fails another lifetime is spilled. The spill/color/spill cycle is repeated until finally a coloring for the graph is found.

## 14.2 Implementation and Data Structures

The coloring algorithm is implemented primarily in two classes: **RegisterAssigner** and **LifeTime**. The abstract **RegisterAssigner class** implements the register allocation algorithm. Its data structures are those needed to perform coloring on the conflict graph. It has four instance variables:

- lifetimes A unordered collection of constrained lifetime objects in the conflict graph.
- assignableRegisters An ordered list of unconstrained lifetime objects removed from the conflict graph.
- preAssigned An unordered collection of lifetime objects in the conflict graph having been pre-assigned to a physical register.
- rtlProgram An intermediate language program whose registers are being assigned. The rtl-Program is queried by the allocator for machine specific information.

The abstract **RegisterAssigner** class supports several different register assignment algorithms. The hierarchy is :

**RegisterAssigner**
    **ChaitinAssigner**
        **HaifaAssigner**
    **OnePassAssigner**

The conflict graph is simply a collection of LifeTime objects. No single collection holds the entire conflict graph, it is instead partitioned into subsets: lifetimes, assignableRegisters and preAssigned. Certain registers are pre-assigned to specific physical registers. They are included in the conflict graph because the same physical registers can sometimes be assigned to other non-conflicting lifetimes and because the allocator may want to spill these registers if they are not frequently used. Typically the pre-assigned registers are NIL, SP, CONTEXT pointers, return values, and procedure

```
colorConflicts: registers
    "Make the conflict graph colorable with NumberOfRegister colors for the
     given class of registers (data or address). The graph consists of a
     collection of lifetimes connected to other lifetimes they overlap with."

    LifeTime numberOfAvailableRegisters: registers.

"    step 1.     Mark preAssigned lifetimes as already colored and
                 resolve spills between them."

    preAssigned ← lifetimes select: [:alife | alife isAssigned].
    lifetimes ← lifetimes select: [:alife | alife notAssigned].

    "Resolve conflicts between preassigned registers"

    self preparePreAssigned.

    lifetimes isEmpty ifTrue: [↑false].

"    step 2.     Mark as don't care all unconstrained nodes (degree < n) since
                 they are trivial to color. If G is empty, return"

    assignableRegisters ← OrderedCollection new.
    self reduceConflicts.

    lifetimes isEmpty ifTrue: [↑true].

"    step 3. Spill a register and repeat step 2."

    "calculate and sort into priority order (ok if empty)"

    self prioritizeLifetimes.

    [lifetimes isEmpty]
        whileFalse: [    self spillRegister.
                         self reduceConflicts ].
    ↑ true
```

Figure 14.1: Coloring the conflict graph.

```
reduceConflicts
    "A lifetime node is unconstrained (assignable) if the number of nodes it
     conflicts with is less than k, where k is the number of available registers.
     This means it can be trivially colored after the constrained nodes are colored.
     Note that if we were to color an unconstrained node before doing all of the
     constrained nodes, it would unnecessarily complicate coloring the
     constrained nodes and very likely cause extra spilling.

     Since removing a node can cascade and allow us to reduce other conflicts,
     we repeat this process until no nodes from lifetimes are deleted. All unconstrained
     nodes are marked as assignable by setting the colored status to dontCare. "

    | oldSize |

    oldSize ← (lifetimes size) + 1.
    [ oldSize > (lifetimes size) ] whileTrue: [
        oldSize ← lifetimes size.
        lifetimes copy
            do: [:life |
                (life isUnConstrained) ifTrue: [
                    life markAsAssignable.
                    lifetimes remove: life.
                    assignableRegisters addLast: life].
                ]]
```

Figure 14.2: Graph reduction.

parameters. Initially the pre-assigned lifetimes are placed in the **preAssigned** set and the others in the **lifetimes** set. The list of **assignableRegisters** is initially empty.

The **LifeTime** class is abstract with **RegisterLifeTime** and **CompositeLifeTime** as concrete subclasses. The distinction between the subclasses arises in lifetime creation and will be discussed later in that chapter. A lifetime object is used both as a node in the conflict graph and a data structure that describes its live range. Instance variable for the **LifeTime** class are:

- *conflicts* Neighboring nodes in the conflict graph whose lifetime overlaps with this lifetime. These includes constraining and unconstraining conflicts.
- *colored* A status (true, false or dontCare) representing my status during graph coloring.
- *priority* A floating point priority value used to make spill choices.
- *end* A set of last uses for this lifetime.

The colored status variable allows us to color the graph in place without actually removing any conflicts. If a lifetime's colored status is *dontCare* then the lifetime has an unconstraining relationship with its conflict neighbors. A true or false status indicates constraining conflicts with its neighbors. The value *true* is given to lifetimes which are pre-assigned (already assigned a "color"). All other lifetimes are initialized to be *false*.

The heuristic coloring of the graph finds unconstrained nodes in **lifetimes** and moves them to **assignableRegisters**. When a lifetime is moved its colored status changes to dontCare. This has the effect of logically removing it from the conflict graph. The graph has been reduced to the empty graph when **lifetimes** is empty. At this point all the lifetimes requiring assignment are in **assignableRegisters** and assigned to physical registers is performed in reverse order.

Figure 3 shows an example conflict graph with 8 nodes to be colored using three physical register. The first first reduction of the graph, will remove nodes C, D, E, F, G and H. The graph will then appear as in Figure 4 with unconstrained nodes shown shaded.

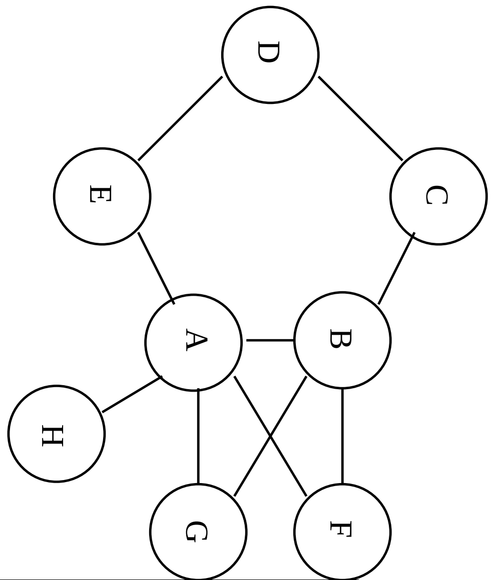
A second reduction pass will eliminate the remaining nodes A and B, moving them to **assignableRegisters**. The graph is now colorable, and registers are assigned in reverse order from the **assignableRegisters** list, starting with B. The final graph with registers assigned is show in Figure 5. The order of assignment is B, A, H, G, F, E, D, C.

## 14.3 Chaitin vs. Chow

Register allocation via graph coloring is practical and widely practiced. The original paper by Chaitin [CAC*81] and its follow up [Cha82] presents a complete description of an algorithm implementing this type of allocator. More recently, Chow [CH84] presents a somewhat different algorithm that attempts to correct some of the perceived inefficiencies of Chaitin's algorithm.

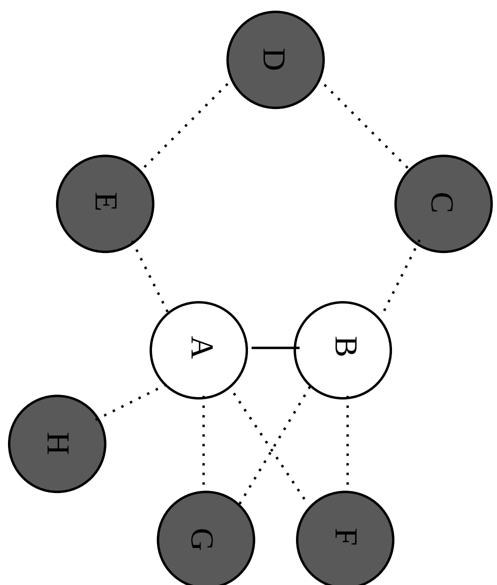
There are many differences between the two algorithms in other areas of the compiler that make it difficult to strictly compare one with the other. There are two serious differences here :

1. Chaitin initially assumes all symbolic registers are in machine registers and must be spilled if not allocated. Chow works with all variables initially allocated in memory, and moves them to registers when assigned by the allocator. For Chow, the advantage is that programs can be run without having to run the allocator.



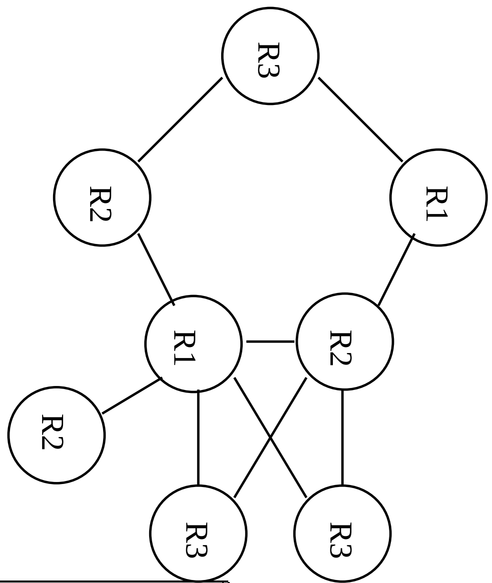

| lifetimes | preAssigned | assignableRegisters |
|---|---|---|
| A B C D E F G H | | |

Figure 14.3: Lifetimes in the conflict graph before assignment.




| lifetimes | preAssigned | assignableRegisters |
|---|---|---|
| A B | | C D E F G H |

Figure 14.4: Lifetimes in the conflict graph during assignment.




| lifetimes | preAssigned | assignableRegisters |
|---|---|---|
| | | C D E F G H A B |

Figure 14.5: Lifetimes in the conflict graph after assignment.

2. Chow's is both a global and local allocator. Local allocation is performed within basic blocks, with global allocation afterwards. This reduces the cost of global allocation by reducing the size of the conflicts graph. Chaitin's is a pure global allocation which works at the intermediate language statement level.

The RTL System has an unlimited number of symbolic registers, so it naturally adapts to the Chaitin convention of initially unassigned registers. In [LH86] it is shown that overall allocation is better when less local allocation is performed by reducing the allowable size of the basic blocks. This indicates that local allocation is unwise as long as the cost of allocation is not a paramount concern.

The major differences between the register allocation algorithms are:

1. splitting lifetimes vs spilling lifetimes
2. different priority metrics
3. one pass vs multipass coloring

Since no published reports have compared objectively the two algorithms in detail, the relative merits of the two algorithms are matters of conjecture. To an unknown degree they also depend upon the characteristics of the target machines register set. For example, it would seem reasonable that lifetime splitting would work better than spilling on a machine with a small register set and a large number of spills. Making good spill decisions is hard and will be examined in more detail later.

### 14.3.1 One Pass vs. MultiPass Coloring

We stated earlier that spilling a lifetime would remove its node and conflicts from the conflict graph. This is not quite true. Spilling a lifetime still requires register resources to load and store its value. Spilling is better thought of as breaking a long lifetime into several very short lifetimes. For example, spilling a lifetime having a single definition and two uses would require three lifetimes. Each of these new smaller lifetimes inherit only a subset of the parent life's conflicts, so the goal of reducing conflicts in the conflict graph is still achieved.

One possible approach is to simply ignore the fact that spilling breaks lifetimes into pieces. If we take the naive assumption that spilling removes a lifetime from the conflict graph, we can make up for this later by iterating the coloring process again. In this scenario, nodes are spilled repeatedly until a colorable graph is found. Then spill code for the lifetimes spilled is inserted. The entire conflict graph is re-built and we attempt to find a coloring again. Only when the graph can be colored without any spills have we found a true coloring. Chaitin reports that convergence with this technique is rapid, and usually only two or three passes are required.

A one pass method of coloring is certainly possible as well. This involves updating the conflict graph and adding a new lifetime for each spill generated. The drawback here is the amount of information that has to be updated for every spill. New conflicts must be computed for the lifetimes spawned, priorities adjusted, etc. Also neighboring lifetimes that were previously unconstrained may be affected by the new lifetimes and now be constrained. These lifetimes must be examined and changed from an unconstrained status to constrained status.

How can unconstrained lifetimes be affected by spilling? If a lifetime overlaps the lifetime being spilled then it might overlap two or more of the new lifetimes. In this case the overlapping lifetimes degree is increased and might become constrained.

# Chapter 15

# Lifetime Creation

Before we can build the conflict graph we must first compute the live ranges of each variable that is a candidate for register allocation. A live range is essentially the extents of def-use chains of a variable. A variable becomes "live" at the definition point and goes "dead" after its last use. Each lifetime is composed of one or more live ranges. Coloring later assigns each lifetime to one physical register or one memory location.

Computing lifetimes involves scanning in topological order the assignment statements in the intermediate language program. Every assignment statement which defines an allocatable symbolic register also defines a new live range. The term "allocatable register" is used to distinguish those symbolic register which map into our pool of physical registers to be assigned. Other assignment statements to memory or bit registers represent storage not allocated by the register allocator.

The basic operation of this phase involves creating individual lifetimes and merging them into composite lifetime. Lifetime creation is an opportune time to perform other necessary functions as well. The RTL program is converted from 3 address to 2 address form and phi assignments are converted into null assignments. These operations perform their work by inserting copies and merging lifetimes. Prior to register allocation, a SSA program does not contain any simple register-to-register copies. The exceptions to this are copies between register types (e.g. moving from a data to pointer register) or loads into pre-assigned registers (e.g. loading an argument into a0 for a procedure call).

Live range information is represented by a lifetime object. The abstract **LifeTime** object was discussed earlier. It has two concrete subclasses **RegisterLifeTime** and **CompositeLifeTime**.

A **RegisterLifeTime** object has one instance variable

- register the symbolic register this lifetime represents

**CompositeLifeTime** objects have two instance variables

- components a list of **RegisterLifeTime** objects that make up this composite lifetime.
- start a collection of one or more definition points for this lifetime that represent the start of the lifetimes live range.

A **RegisterLifeTime** object represents a single live range. It has a single definition point and one or more uses. A **CompositeLifeTime** object represents two or more live ranges that are contiguous. It has multiple definition points and multiple uses. All components of a composite lifetime will be assigned to the same physical register. The distinction between composite and single lifetimes is necessitated by SSA form. Without using SSA, single register lifetimes could be easily combined by renaming all the occurrences of one to the name of the other. But preserving the single definition clause of SSA prohibits this so we merge lifetimes into composite objects instead. The components of a **CompositeLifeTime** are each **RegisterLifeTime** objects.

When two lifetimes are merged, the effect is to require their assignment to the same registers. Should they be spilled to memory, they will share the same memory location. Only live ranges that "touch" at a common assignment statement can be merged. If we merged lifetimes that didn't touch, this would create holes where a physical register would be free but could not be allocated. Merging of lifetime names is performed using fast union-find algorithm.

## 15.1 Names

A crucial part of this phase involves deciding what constitutes a live range. We want to find the maximal live range units but without introducing any "holes". [Cha82] calls this "getting the names right".

SSA greatly simplifies this process because the program variables have already been sorted into def-use chains. Since there can be one and only one definition for each SSA variable, each variable is exactly one live range. Without SSA form we would have to rename or somehow map the program variables into live range variables. Consider this example :

| | $non-SSA$ | $SSA$ |
|---|---|---|
| 1. | $a \leftarrow 0.$ | $a_1 \leftarrow 0.$ |
| 2. | $b \leftarrow a + 1.$ | $b_1 \leftarrow a_1 + 1.$ |
| 3. | ... | ... |
| 4. | ... | ... |
| 5. | $a \leftarrow 1.$ | $a_2 \leftarrow 1.$ |
| 6. | $b \leftarrow a + b.$ | $b_2 \leftarrow a_2 + 1.$ |

The course granularity of variable names $a$ and $b$ on the left make it costly to allocate them to registers. A register allocated to $a$ at statement 1 will remain in use through statement 10. On the right, we see that breaking up the variable $a$ into its true live ranges allows us to allocate a register to $a_1$ and $a_2$ separately such that this register can be reused in statements 3 and 4. We might even allocate different registers to $a_1$ and $a_2$.

It is interesting to note that Chow's chose to allocate on the basis of program variable names and not def-use names. The reasons given for this are two fold: the efficiency of coloring is proportional to the number of nodes in the conflict graph. Secondly, Chow's technique of live range splitting means that when there is competition for registers a live range can be split, better approximating its def-use components. [1] Using program variable names also reduced the memory used by spill code.

[1] [Hsu87] shows how this can sometimes lead to poor allocation by leaving "holes" in high priority lifetimes since low priority lifetimes are the ones split.

## 15.2 Merging

Once variables have been broken down into their live range components the contiguous live ranges are re-joinged by lifetime merging. Two live ranges are contiguous or "touch" at an assignment if the end of one live range is the start of the other. Several other operations are performed as a side effect of merging: phi assignments become null operations and the program is converted from three-address to two-address form. Another benefit is that conflict graph will be smaller. Thus, merging recoups some of the advantages of Chow's naming convention without none of the side effects.

Merging live ranges takes place by ordering assignment statements to allocatable registers by their node's position in the control flow graph and then merging them in that order. The correct ordering of these SSA assignments is essential for correct merging. When an assignemnt statement merges its target with with one or more of its sources, the target registers lifetime must be an unmerged lifetime. Phi assignments with back edges present a special exception to this rule.

The assignments to merge come in two classes, ordinary three-address assignments and phi assignments. Different methods are used to perform their respective merging operations.

The basic methods used to merge assignments are show in Figure 15.1

### 15.2.1 Assignment Merging

Ordinary assignments often require merging to satisfy a target machine language two address instruction format. This conversion must be performed before beginning register allocation.

The conversion is very straight forward. Any three address statement such as

$$s1 \leftarrow s2 + s3$$

can be reduced to a two address form by simply requiring that the symbolic registers *s1* and *s2* be assigned to the same physical register. Enforcement is done by constraining *s1* and *s2* to belong to the same lifetime. Instruction scheduling, which preceded this step of the compiler, has attached to each assignment a list of constraints to be enforced. The register assigner performs this task by merging and copying.

When the constraints of an assignment specify that the source (s2) and destination (s1) symbolic registers must be assigned the same physical register a new **CompositeLifeTime** is created which represents the combined lifetimes of s2 and s1.

It is not always possible to trivially merge s1 and s2 because s2 may have a later use after s1. In this case merging would destroy the still live value of s2. This requires that a copy be inserted and then the copy of s2 can be legally merged with s1. A copy is required for either of two reasons:

- the source register is live after this assignment
- the source and destination registers have been pre-assigned to different physical registers.

If a copy is required, the lifetime of the new copy is merged with the destination lifetime leaving the original source lifetime unchanged. Below is an example of merging in a short code fragment.

**RegisterTransfer**:

**mergeLifetime: lifetimeDict**

```
    "Enforce my constraints, if any, by merging my destination
     with one of the source registers."

    self instruction constraints enforceOn: self in: lifetimeDict
```

**PhiAssignment**:

**mergeLifetime: lifetimeDict**

```
    "Merge the lifetimes of all my registers and insert copies
     when necessary in the predecessor nodes. Merging
     converts me into a null operation as all of my sources
     must reside in the same physical register as my destination

     Phi source registers need not be unique. For example:
     phi( d126.3 d116.3 d116.3 ) is perfectly legal and must be
     taken into account when inserting copies. The intersects:
     method must be used here instead of liveAfter: because
     phi sources may be composite life times.
    "

    | regLife mergeReg |

    self isPhiAtIntervalRootNode
        ifTrue: [↑self mergeEntryPadLifeTime: lifetimeDict].

    mergeReg ← nil.
    nodeToRegisterMap values asSet do: [:register |
        regLife ← lifetimeDict at: register.
        ((regLife liveAfter: self) or:
                [mergeReg notNil and: [regLife intersects: (lifetimeDict at: mergeReg)]])
            ifTrue: [
                "insert copy in a predecessor node"
                register ← register
                                insertCopyAtEndOfNode: (self commonNodeFor: register)
                                phi: self in: lifetimeDict].
        mergeReg isNil
            ifTrue: [mergeReg ← register]
            ifFalse: [mergeReg mergeWith: register in: lifetimeDict]].

    destination mergeWith: mergeReg in: lifetimeDict
```

Figure 15.1: Merging lifetimes.

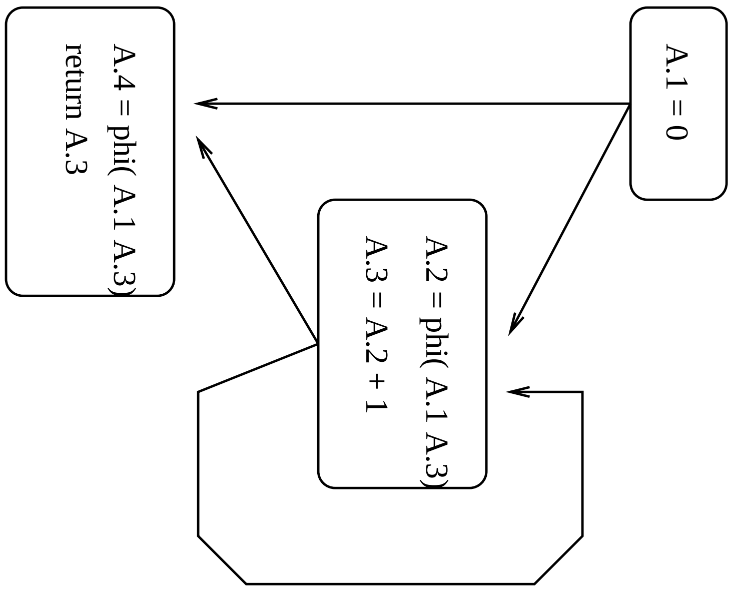



Figure 15.2:

| *before* | *after* |
|---|---|
| $a \leftarrow 1.$ | $a \leftarrow 1.$ |
| $b \leftarrow a + 5.$ | $d \leftarrow a.$ |
| $c \leftarrow a + 10.$ | $b \leftarrow d + 5.$ |
| | $c \leftarrow a + 10.$ |
| $lifetimes : (a)(b)(c)$ | $lifetimes : (ac)(bd)$ |

### 15.2.2 Phi Assignments

A program in SSA form contains phi assignments which must be eliminated before register allocation. Phi assignments are a logical construct having no representation in machine languages. Two strategies are possible for dealing with this phi's: convert out of SSA or reduce the phi assignments to null operations.

Phi assignments represent a copy operation from a variable source to a destination register. The source chosen depends upon the execution path leading to the phi assignment statement. A phi assignment contains a node to register map in which each predecessors to the phi assignment's node is keyed to a symbolic register value. The variables in a phi assignment may represent a single contiguous lifetime and allocatable to a single register as in Figure 2 for variables A.1, A.2, A.3 and A.4. Or they can be overlapping lifetimes which must be implemented with an actual copy as with variables A.1 and B.1 in Figure 3.

In [CFR*91], Cytron proposes converting out of SSA form and removing phi assignments before register allocation. Copies are placed in the immediate predecessor nodes of every source mapped in the phi assignment. Dead code removal and the copy subsumption phase of register allocation

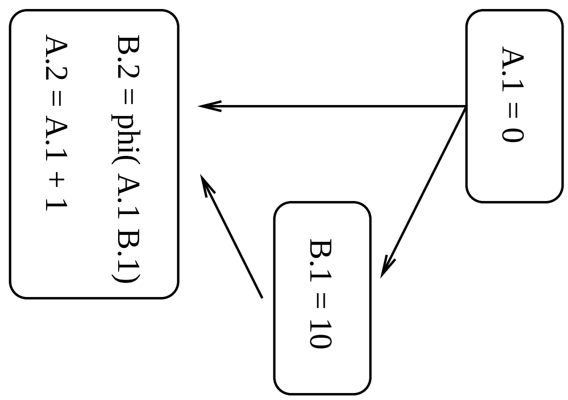



Figure 15.3:

will eventually eliminate unnecessary copies. This acts as a "naive" method of phi assignment removal.

Our technique preserves SSA and handles phi assignments within the allocation process by rendering them null assignments. A register-to-register copy is a null operation if the allocator has assigned its source and destination to the same physical register. No code will be generated on such an assignment. Similarly, a phi assignment is a null operation if its multiple sources and single destination have been assigned to the same physical register. To arrive at a point where all phi assignments are null operations, we insert copies where required (and only where required) and merge the lifetimes of the remaining registers. Merging the lifetimes ensures that whatever physical register will eventually be assigned, it will be assigned to all of the elements in the phi assignment. Phi assignments remain in the program text, but all their symbolic registers belong to the same lifetime.

There are two species of phi assignments: those not in the root node of an interval and those which are. Phi assignments in the root node of an interval perform the special task of merging an initial value used at the start of the interval with a value calculated in each iteration of the loop (see Figure 2). They have two special properties :

1. There are two and only two sources. One source originates in the interval predecessor, the other from the back-edge jump node.

2. The source mapped to the back edge jump has a use (in the static program text) before their definition.

Calculation of when copies need to be created is considerably more difficult for phi assignments than it is for ordinary assignments. This is a more rigorous approach than the naive method, but it preserves the program in SSA form. One major advantage of maintaining SSA is that it would allow optimization to be re-run on the program after the register assigner has inserted spill code.

## 15.3 After Merging

At the end of this phase we will have achieved several goals. Namely,

- phi assignment nullification.
- subsumption of contiguous live ranges into composite lifetimes.
- computing start and end points of the life range
- converting 3-address RTL to 2-address.

Merging lifetimes has many side effects upon the conflicts graph. It will usually raise the degree of the two nodes being merged, but lower the degree of their neighbors. Higher degree nodes are harder to color so merging comes at some cost. The smaller size of the conflicts graph means this cost is usually worthwhile.

# Chapter 16

# Conflicts

Merging lifetimes and computing the conflicts graph requires lifetimes to respond to messages asking where they end and if they overlap with a particular lifetimes. The three primary methods which implement this protocol are: liveAfter:, intersects: and contains:.

A lifetime stores information about where its live range starts and ends in instance variables. It could be computed on demand, but it is more efficient if pre-computed during lifetime creation. For a **CompositeLifetime** these are start and end. A **RegisterLifeTime** also has an end instance variable while its single definition point is identically its single start. A lifetime has at least one start and at least one end.

## 16.1 Live After

Merging two live ranges A and B in a typical assignment statement such as

$$B \leftarrow A + 1$$

requires knowing if A has any uses extending after this assignment. It would be obviously unwise to merge the two if A was used after this since its value would be destroyed.

The liveAfter: method is responsible for making these determinations. Two factors complicate the task: loops and phi assignments. In the normal situation, a variable is dead if no other uses follow it. The **RegisterTransfer** class implements protocol for this with the message isAncestorOfRT: and isDescendantOf:. There are two exceptions to this for phi assignments (which are not nulled until after merging). If a variable is a loop invariant having a definition outside of the loop and its last uses within, then the last uses are not dead after the use point. That is, they are only dead upon exit of the loop. The next iteration of the loop will use them again. Secondly, a use in a phi assignment that is a descendent of the current register transfer may be an inaccessible use. If the register in the phi assignment's nodeToRegisterMap is mapped to a node unreachable from the current assignment, then the use is inaccessible and ignored.

## 16.2 Contains and Intersects

The conflicts graph is built immediately after lifetimes have been created. This step consist of comparing each lifetime in turn with all of the other lifetime. The method *intersects:* implements

```
liveAfter: rt
    "Return true if I am live after rt. There is a use of my register after rt if
     either (1) the register is a loop invariant or (2) a use other than rt follows me. "

    | reg |

    reg ← self registerUsedAt: rt.                "this component"

"    A reg with a definition outside rt's interval is a loop invariant and is always live
    after this rt. An exception is phi assignments in the root node of an interval.
    For these phi rt's, the definition is not a loop invariant if it is in the interval
    entry pad."

    (reg definition interval level < rt interval level)
        ifTrue: [ ((rt isPhiAtIntervalRootNode not) or: [
                    (reg definition interval) ~~ (rt interval interval)]) ifTrue: [↑true]].

    reg uses do: [:user |
            (rt ~~ user) ifTrue: [
                "Use follows rt"
                (rt isAncestorOfRT: user) ifTrue: [
                    (user isPhiAssignment not) ifTrue: [↑true].

                    "phi assignment values must be reachable from current node"
                    (user register: reg isReachableFrom: rt) ifTrue: [↑true].
                    "user and rt are phi assignments in the same node"
                    (rt flowNode) == (user flowNode) ifTrue: [↑true]].

                (rt interval withinInterval: user interval) ifTrue: [
                    "Use is on back−edge at top of loop"
                    (user isBackEdgeUseOf: reg) ifTrue: [↑true]
                ]]].

    ↑false
```

Figure 16.1: Testing if a lifetime is live after a use point.

the comparison operation, so that sending the message "a intersects: b" will return true if a's live range overlaps that of b's.

Conflicts are dual links between conflicting lifetime nodes in the graph. If lifetime A conflicts with lifetime B, each will have the other in its conflict set. Conflicts are created symmetrically with the conflictsWith: message and removed symmetrically with the noLongerConflictsWith: message. A situation where A conflicts with B, yet B does not conflict with A is illegal. The conflict information is never changed once it is computed except when spilling.

### 16.2.1 Definition Point Intersection

Earlier we defined the notion of conflicts only losely. Our intuitive definition was that two lifetimes interfere if they are "live" simultaneously in the data-flow sense. This definition is overly conservative because it is possible for two lifetimes to be live simultaneously but not interfere when their live ranges exist over disjoint sets of basic blocks. A simple example of this is the conditional statement :

$if(a < 0)\{$
$b = a + 1;$
$x = b \ll 1;$
$\}else\{$
$c = a - 1;$
$x = c \ll 1;$
$\}$

The two variables $b$ and $c$ in this example are live simultaneously but do not interfere.

The definition point concept of interference is derived from [CAC*81]:

> *A interferes with B if B has a definition between A's start and end, or if A has a definition between B's start and end.*

Note that this definition means lifetimes that "touch" do not interfere. For example in

$$r1 \leftarrow r2 + 1$$

both r1 and r2 are live at the same assignment, but they do not interfere.

### 16.2.2 Computing Conflicts

One nice feature of using definition points is that it is possible to compute the conflicts in a single pass of the intermediate language program. This achieves a linear bound in the number of nodes.

```
. definition point example

1. a - 0
2. b - 2
3. c - 1 + b
```

**intersects: aLifetime**

*"return true if my lifetime overlaps with aLifetime. "*

```
self start do: [:rt | (aLifetime contains: rt) ifTrue: [↑true]].
aLifetime start do: [:rt | (self contains: rt) ifTrue: [↑true]].
↑false
```

Figure 16.2: Testing two lifetimes for conflicts.

**contains: rtOrInterval**

*"Return true if rtOrInterval is after a definition and before one of the uses."*

```
interval == rtOrInterval ifTrue: [↑true].
self start detect: [:rt | rtOrInterval isDescendantOf: rt] ifNone: [↑false].
self end do: [:rt | (rt isDescendantOf: rtOrInterval) ifTrue: [↑true]].
↑false
```

Figure 16.3: Testing if a definition point falls within a live range.

```
4. d - a + c

it is very easy to compute
    a :  <1><4> so it interferes with he defs b,c
    b :  <2><3> no defs, no new conflicts added
c :  <3><4> no new defs

. explain our (somewhat) similar method
```

# Chapter 17

# Priority Based Spilling

When the conflict graph is non-empty and can not be reduced further using a coloring heuristic, a node is removed from the graph by spilling it to memory. Spilling a node allocates storage for it in memory and introduces spill code that will load and store the value at definition and usage points in the program.

The most important part of spilling is making the choice of which node or nodes to spill. There are two criteria we have in mind while spilling :

- quickly find a coloring with as few nodes spilled as possible.
- choose those nodes whose spill code has the lowest run-time cost.

Choosing nodes by priorities is a greedy algorithm which selects the lowest priority lifetime having a conflict with the highest priority lifetime. Using a greedy algorithm means that the final list of nodes spilled is often not optimal because individual decisions are made at the expensive of the overall group of nodes spilled.

The priority function must do what it can to balance the above two criteria against each other so that our greedy approach works. A successful priority function must also perform well across a wide variety of situations, from spilling only one node to spilling nearly all of the nodes.

Priority functions are usually composed of two parts. The numerator is a function of spill code cost. The lower the estimated run-time cost of the spill code, the lower is the priority. The denominator is a function reflecting the expected utility of spilling this particular node from the graph. The denominator is used to direct spilling toward those nodes causing conflict bottlenecks.

The live ranges show in Figure 1 illustrate two examples where one of the stated criteria is appropriate for the situation. In these figures vertical length represents the relative number of instructions the range is live over and horizontal bars represent definition or use points where spill code would be inserted.

Consider the problem when we have two registers to allocate for these five live ranges. On the left, spilling node A will allow us to allocated B, C, D, and E to registers with only one spill. Spilling either B or C first would require a second spill as well. On the right, one spill of A would also make the graph colorable. But it would be less expensive to spill the lifetimes of B and D since the lifetime for A is more frequently used. Note that in both cases if we chose to spill B and *then* A this would have spilled more nodes than absolutely needed. A solution to this problem by altering the coloring heuristics is described in [BCKT89].

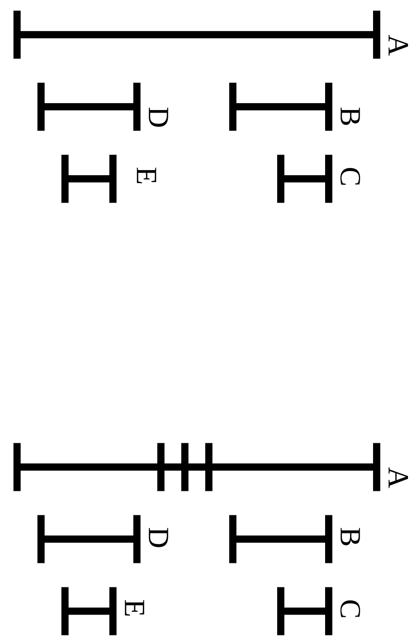



Figure 17.1: Live Ranges

## 17.1 Spilling by Priority

Priorities are not assigned until the first attempt to reduce the conflict graph has failed. At this time all nodes not colored *dontCare* are given priority values. A reasonable assumption to make is that the node having the highest priority should be allocated to a register. Thus the process of selecting a node to spill begins by finding the node we least want to spill from the lifetimes list. Given that the highest priority lifetime belongs in a register, some of its conflicts must belong in memory.

The current conflict neighbors of the node are examined, and the neighbor with the lowest priority is selected and spilled. Removing this node and its conflicts from the graph has probably reduced the degree of several nodes, so we attempt to reduce the graph again by applying the coloring heuristic. This process of spill/reduce repeats until the conflict graph is empty.

## 17.2 The Cost Function

Any priority function must take into account the spill cost of a given variable. This cost is an estimate based upon the number of definitions and uses which would require a load or store operation and is weighted by a factor reflecting loop nesting depth. The cost function then is:

$$Cost(n) = \sum_{i \in I} 5^{depth(i)}$$

where I is the set of instructions where lifetime n has a definition or use.

The degree to which instructions inside of loops should be weighted must be estimated in the absence of profiling information. Chaitin suggests a factor of 10, while the Haifa implementation

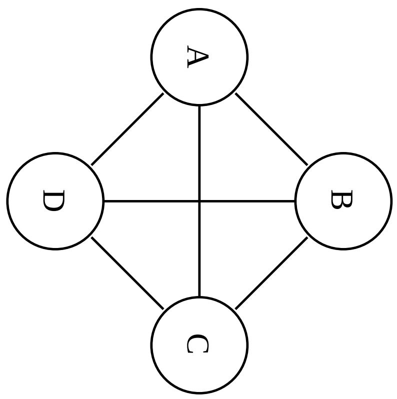



Figure 17.2: Conflict Graph

uses a more modest factor of 5. Since our target language Smalltalk tends to have frequent iterations over small sets, the more modest 5x factor seems appropriate. A lower weight also helps prevent granting unrealistic priorities to variables inside of deeply nested loops. It is dubious to assume that an instruction inside of a triply nested loop is executed 1000 times ($10^3$).

A priority function takes a lifetime's calculated spill cost and attenuates it by a measure of how easily or hard that node is to keep in a register. Generally speaking, a lifetime with a live range extending over many flow nodes should have a lower priority than a temporary lifetime having the same spill cost. One simple and effective priority function is the one used by Chaitin in [Cha82]

$$Priority(n) = Cost(n)/Degree(n)$$

This priority function is sufficient for intelligently allocating for both of the earlier examples.

A more complex example[1] where Chaitin's priority function is inadequate :

$$\begin{aligned}
&a \leftarrow 1\\
&b \leftarrow 2\\
&c \leftarrow 3\\
&d \leftarrow 0\\
&d \leftarrow d + c\\
&d \leftarrow d + b\\
&d \leftarrow d + a
\end{aligned}$$

Here the nodes all have degree of three and the conflict graph, shown in Figure 2, is a 4-clique. The spill cost for $d$ is seven, and for the others two, resulting in Chaitin priorities of 7/3 and 3/2 respectively. Given three registers, the priorities would cause an arbitrary choice to be made among nodes $a$, $b$ and $c$. But in fact the best spill choice would be variable $a$. Spilling either register $b$ or $c$ would require two spills since the spill code itself requires register resources. Although Chaitin's priority function is inadequate in this situation, it would at least avoid the worst possible choice of variable $d$. What is missing from the priority function in this situation is a measure of the lengths of the live ranges.

[1]Code scheduling can be used to reorder assignments to minimize lifetimes. In this case it could eliminate the need for any spilling. General methods for this are discussed in [Dav86]. The approach used in the RTL System is described earlier in this document.

### 17.2.1 Chow's Priority Function

Where Chaitin's priority function used the degree of a node as the divisor, Chow [CH84] replaces it with an estimate of the flow node density of a lifetime. His priority function is defined as :

$$Priority(n) = Cost(n)/bbCount(n)$$

where bbCount(n) is the number of basic blocks over which the variable $n$ is live. Thus, sparse long lifetimes are spilled, while short busy lifetimes are favored by the allocator. By eliminating from spill priorities any consideration of the node's degree it would seem to lead to more, but less expensive spills. Favoring more spills over fewer may be partially offset by the fact that Chow splits nodes rather than spilling them. That is, less spill code need be introduced for each spill.

### 17.2.2 Haifa priority functions

Both the Chaitin and Chow priority functions are ad hoc approaches without any strong empirical support. Yet an ideal priority function that will select the best lifetime or lifetimes to spill in every circumstance is probably not possible given the complexity of spilling. Recall that the way our algorithm works means one greedy choice early on can produce poor later ones. This suggest that a choice among several different spill lists corresponding to different priority functions might better approximate "perfect spilling".

The Haifa "best-of-three" algorithm is an extension of the basic Chaitin assigner which uses three different priority functions. Reductions of the graph with spilling are computed for each of the three functions, the algorithm selects from the different spill lists the least expensive. The multi-pass design of the Chaitin algorithm integrates very nicely with this technique because only after a complete spill list is computed is the spill code introduced. This allows us to compute three different spill lists and make an informed choice before inserting the spills. Because computing not coloring the conflicts graph dominates computationally the entire allocation process the Haifa method adds little overall overhead. The conflicts graph is computed for each pass with three reductions performed using the suite of priority functions. Between reductions the graph can be cheaply rebuilt since using the colored status for nodes means conflicts are not removed. Once the three spill lists have been computed, the least expensive is selected, and the spills inserted.

The three priority functions used are:

$$Priority_1(n) = Cost(n)/Degree(n)^2 Priority_2(n) = Cost(n)/(Area(n)*Degree(n)) Priority_3(n) = Cost(n)/(Area(n)*Degree(n)^2)$$

where $Area(n)$ is a new metric used to estimate the overall contribution of n to the toal register pressure. The formula to compute area is:

$$Area(n) = \sum_{i \in I} 5^{depth(i)} * width(i)$$

The width function is a count of the number of lifetimes live at instruction i.

There are three major advantages to the Haifa assigner. First, the problems arising from a using a greedy algorithm is helped by the ability to choose from different spill list. Those priority functions which lead us to take a particularly poor greedy choice can be eliminated. Second, a priority function which performs well in one situation may work poorly in another. Several priority functions can better span the needs of different situations.

## 17.3 Spilling Around Procedure Calls

When a procedure call occurs all registers in use by the caller must be saved to memory and then restored after the call has returned. This may be implemented on either a caller save or callee save basis.

Saving all registers in use can be an expensive operation, especially for machines with large register sets. Aggressive in-lining of procedure calls is one effective way to reduce, but not eliminate this problem. However, in-lining makes the register allocators job harder since the allocator has to now find physical registers for the lifetimes in both the caller and callee procedures.

Our allocator implements the caller save convention. The register save and restore code is inserted after a coloring of the conflict graph has been found but before physical registers have been assigned. At this point in time it is possible to discern which physical registers will be in use and hence which will need to be spilled around the call. Thus we spill only those physical registers actually in use and not all of the machines physical registers at each procedure call.

Spilling around calls is integrated into the normal register allocation process. Fake lifetimes are created for each of the physical registers reserved for use by a procedure call and pre-assigned to one of the physical registers. These lifetimes and their conflicts are entered into the conflict graph during lifetime creation. But the fake lifetimes lifetimes are removed from the preAssigned list and placed on the assignableRegisters list before coloring begins. They are assigned infinite priority so that the spilling will occur among only the real lifetimes which intersect the calls. All this ensures that their conflicts will be updated but passive until after the graph is colorable.

Immediately before actual physical register assignment, we spill all those lifetimes which have conflicts with the fake call lifetimes.

## 17.4 Resolving Pre-Assigned Conflicts

As noted earlier, some of the symbolic registers are pre-assigned to physical registers by other sections of the compiler. It sometimes happens that the pre-assignments conflict with each other. Because other parts of the compiler aren't aware of lifetimes and such things, pre-assignments are added without regard for possible conflicts. The work of resolving these conflicts is performed at register assignment time by the allocator. An alternate approach to dealing with pre-assigned registers would be to copy the pre-assigned registers out to un-assigned symbolic registers immediately after their definition or use which requires pre-assignment. Later copy subsumption can remove any unnecessary copies. Since our allocator avoids the need for copy subsumption, this is not currently a viable alternative.

Most pre-assignment conflicts arise as a result of procedure calls. When a call is made argument registers and return values have to be loaded into the appropriate physical registers. These physical registers may already be in use, for example by the original arguments to the caller procedure.

Conflics are resolved in pairs, and because of the nature of these pre-assigned conflicts there is usually an outerlife/innerlife relationship between them. One lifetime (i.e. argument zero to the current procedure) surrounds the other (i.e. argument zero being loaded for a procedure call). Because the outerlife typically has a long live range and may cause many pre-assignment conflicts we choose to spill it irrespective of spill cost. The outerlife is discovered by comparing the degree of the two lifetimes. The one having the greater degree is assumed to be the outerlife. This assumption, which is not true in general, works well for this special case. Priorities are not used in

```
resolvePreAssignedConflicts: aLifetime
    "Check if I overlap with another lifetime assigned to the
     same physical register. If so, spill the one of higher
     degree and return true to recheck the conflicts again."

    "never resolve conflicts between SP or CONTEXT"
    aLifetime representativeRegister isSPOrCONTEXT ifTrue: [ ↑false ].

    aLifetime assignedConflicts do: [ :life |
        (aLifetime physicalRegister == life physicalRegister)
        ifTrue:    [
            life degree < aLifetime degree
                ifTrue: [ self spillRegister: aLifetime]
                ifFalse: [ self spillRegister: life].
            ↑true]].
    ↑false
```

Figure 17.3: Spilling pre-assigned registers.

making pre-assignment spills. Admittedly, our resolution of these conflicts is handled in a simple fashion that could perhaps be improved upon.

## 17.5 Clean Spilling

Spilling a symbolic register with a single definition point and a single use is a trivial matter. A store is inserted immediately after the definition point, and a load instruction inserted before the use. However, more complex live ranges are the norm and these situations offer opportunities to reduce the amount of spillage required for a given live range.

Recall that register spills themselves require registers, although for only the briefest possible time. When we spill a node from the interference graph, we don't eliminate the node but rather break it up into very small lifetimes. If these small lifetimes are sufficiently close together, situations can occur where spilling uses may not free up a physical register. Instead, spilling a symbolic register will just mean that a physical register has to be allocated to handle the load and stores in the new temporary lifetimes. Consider this example :

| | |
|---|---|
| $a \leftarrow 7$ | $a \leftarrow 7$ |
| ... | ... |
| $b \leftarrow a \ll 1$ | $b \leftarrow a \ll 1$ |
| $x \leftarrow 3$ | $x \leftarrow 3$ |
| $c \leftarrow c + a$ | $c \leftarrow c + x$ |
| $c \leftarrow c + x$ | $c \leftarrow c + a$ |

Assuming that $a$ has been selected for spilling, how many reloads should be inserted? Inserting two loads for each of the uses of $a$ will not free up any physical registers for use by another variable. Notice that spilling $a$ in such a manner will not allow us to allocate a register to $x$ in the first figure. In the second figure the situation is different. Because a lifetime ends between the two uses of $a$ inserting two loads for $a$ will free a register for allocation to $x$.

These spilling situations are easily detected whenever a second unspill of a variable is attempted within a single basic block. When this happens the second unload is merged with the first, and no spill code is inserted for the second unspill.

## 17.6 Live Range Splitting

Splitting lifetimes instead of spilling is one of the fundamental concepts of the Chow register assigner. The splitting of a lifetime tries to divide the lifetime into two or more segments. The points at which to divide the lifetime are selected such that one segment is of degree N-1 where N is the number of registers. This segment is then unconstrained and removed from the graph.

Splitting lifetimes requires recomputing SSA for those variables split. This is thus far unimplemented.

# Chapter 18

# Testing Register Allocation

When register allocation is performed incorrectly, the result is a program which computes the wrong result or worse a program that crashes. It is often difficult in such situations to determine which section of the compiler is responsible for the error. A program with a large amount of spill code is particularly hard to debug by inspection. One partial remedy for this is to perform an audit of the register allocation.

There are several different types of register allocation failures which can result in incorrect programs. Auditing concentrates on the simplest and most frequent error: two overlapping live ranges are assigned to the same physical register. When this happens, one variable has its value overwritten by the other. There are three main failure points for this sort of error:

- merging has illegally combined two intersecting live ranges
- copies were not inserted when needed or were not inserted in the right place.
- interferences between the between lifetimes were incorrectly computed.

An audit of register allocation examines the possible paths of execution and for each path verifies all register-based variable accesses. When a variable is accessed, it must reside in the physical register that was assigned to to it by the allocator. If it is absent or has been overwritten by some other variable, we have detected a bad allocation. Naturally, any program having conditional branches and loops has more than one possible execution path. By traversing all the static paths in the program with different audits complete path coverage is achieved. A dynamic path is the path taken by an executing program, a static path is one observed in the flow graph. Not all paths of a program are necessarily executable. Certain loops may never be taken, for example. This fact is unimportant to the register allocator which can only assume each node may be executed and the register auditor which ensures that all paths of execution could at least theoretically be executed correctly.

Auditing is performed with one or more **AuditLedger** objects. An **AuditLeger** is maintained for each static path. It has instance variable :

- pathDictionary a dictionary mapping physical register names to the symbolic register lifetime currently residing in it.
- contextPointer register where context pointer resides

```
auditRTDefs: aLedger
    (self registersSet) do: [:lreg |
        lreg notAssigned
            ifFalse: [
                ((self isAssignment) and: [self isEquivAssignment])
                    ifTrue:     [ aLedger defs: lreg]
                    ifFalse:    [ aLedger def: lreg]]]
```

Figure 18.1: Auditing registers used in a transfer.

- stackPointer register where stack pointer resides
- trace an ordered list of all nodes traversed thus far in auditing the current path.
- predecessors the current node's predecessor node. This is necessary in order to calculate phi assignment uses.
- jumpsTaken conditional node jumps already taken which lead back to the top of a loop and should not be taken again.
- lastEdgeChosen the last conditional node having an available choice

The pathDictionary simulates the machines register set and is the principle data structure used to audit definitions and uses. Because the special registers assigned to hold the context and stack pointers do not have the usual def-use relationships that normal program variables do, extra information about them is kept in context and sp to perform some basic sanity checks.

For each assignment statement defining a symbolic register, the dictionary entry corresponding to its physical register is updated. For each use of a symbolic register, the dictionary is checked to make sure the symbolic register resides in the correct physical register and has not been overwritten by a prior assignment. The idea here is that when a live value is clobbered it will be missing from the dictionary the next time some other statement needs to use it.

The basic methods auditing assignments are show in Figures 18.2 and 18.1

## 18.1 Traversing All Paths

Each path in the program is given a unique audit dictionary. The paths are traversed by starting in the root node with a single audit dictionary and forking at points where paths divergence. In each node we audit all of the transfers within that node and then proceed to the following node. If there is only one predecessor of a node then the unconditional jump is followed and auditing begins on the predecessor. If the node has two predecessors (e.g. an **RTLConditionalJumpNode**) then the audit dictionary is duplicated, and a new audit is forked to follow the conditional jump while the parent follows the path of the unconditional jump. Thus, two independent audits continue down the two divergent paths.

```
auditRTUses: aLedger
    "note: we must skip context & SP registers because they are
     added without regard to SSA"

    (self registersUsed) do: [:lreg |
        lreg isSPOrCONTEXT
            ifFalse: [ lreg notAssigned
                            ifFalse: [ aLedger use: lreg ]
                            ifTrue: [ aLedger assertIsBit: lreg ]]]
```

Figure 18.2: Auditing registers defined in a transfer.

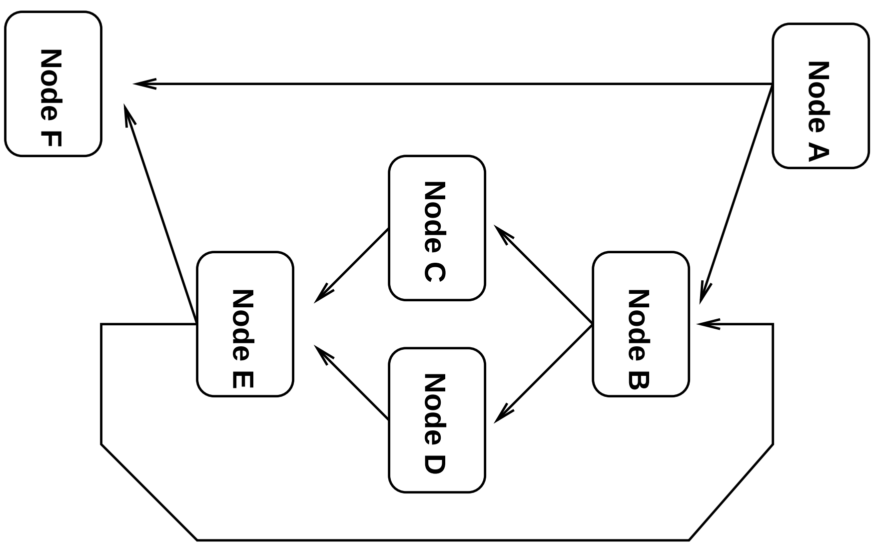



Figure 18.3: Nodes of an RTL Program

Conditional jumps which lead back to the top of a loop are given special consideration since we only wish to traverse, on a particular path, the loop once. When an audit path reaches the return statement, the audit of that path has been successful.

Figure 3 shows the flow nodes for an rtl program. There are seven possible paths through this RTL program. They are :

- A F
- A B C E F
- A B D E F
- A B C E B C E F
- A B C E B D E F

- A B D E B D E F
- A B D E B C E F

The number of paths is at worst $n^2$ where $n$ is the number of nodes having conditional jumps. Minimally, auditing need only be enabled when a problem arises so that the time taken by register allocation is of not of great importance. Because auditing simulates execution of the program, an infinite loop in the program can cause the auditor to loop forever.

In designing the auditor special care was given to making the algorithm as independent of other aspects of the compiler as possible. This was done to help ensure that errors elsewhere would not be replicated in the auditing process.

Experience using register auditing as a standard part of the register assigner has been very positive. Many errors were detected early on with attempts to modify either the assigner code or code in portions of the compiler upstream of the register assigner which broke the assigner. Indeed, it is hard in retrospect to envision working on the assigner without this safety net.

# Appendix A

# An Unoptimized RTL Program

The result of sending the printOn: message to an instance of **RTLProgram** representing the **Array** method

```
sumAll

    | sum |
    sum ← 0.
    self do: [:i | sum ← sum + i].
    ↑sum
```

appears below. Items in angle brackets are labels.

```
<RTLJumpNode 102>
<100> SP ← − − −
<101> CONTEXT ← − − −
<102> NIL ← − − −
<103> m ← − − −
<104> rom ← − − −
<105> a101 ← − − −
<106> ↑<RTLJumpNode 103>

<RTLJumpNode 103>
<107> a102 ← NIL
<108> a103 ← NIL
<109> a104 ← NIL
<110> a105 ← NIL
<111> a106 ← NIL
<112> a107 ← NIL
<113> a108 ← NIL
<114> a109 ← NIL
<115> a110 ← NIL
<116> a111 ← NIL
<117> a112 ← NIL
<118> a113 ← NIL
<119> a114 ← NIL
<120> a115 ← NIL
<121> a116 ← NIL
<122> a117 ← NIL
<123> a118 ← NIL
<124> a119 ← NIL
<125> a120 ← NIL
<126> a121 ← NIL
<127> a122 ← NIL
<128> a123 ← NIL
<129> a124 ← NIL
<130> a125 ← NIL
<131> a126 ← NIL
<132> a127 ← NIL
<133> a128 ← NIL
<134> a129 ← NIL
<135> a130 ← NIL
<136> a131 ← NIL
<137> a132 ← NIL
<138> a133 ← NIL
<139> a134 ← NIL
<140> a135 ← NIL
<141> a136 ← NIL
<142> a137 ← NIL
<143> a138 ← NIL
<144> a139 ← NIL
```

```
<145> a140 ← NIL
<146> a141 ← NIL
<147> a142 ← NIL
<148> a143 ← NIL
<149> a144 ← NIL
<150> a145 ← NIL
<151> a146 ← NIL
<152> ↑<RTLJumpNode 104>

<RTLJumpNode 104>
<153> d101 ← 0
<154> a102 ← d101
<155> ↑<RTLJumpNode 105>

<RTLJumpNode 105>
<156> d102 ← 0
<157> a112 ← d102
<158> ↑<RTLJumpNode 106>

<RTLJumpNode 106>
<159> a147 ← a101
<160> ↑<RTLJumpNode 107>

<RTLJumpNode 107>
<161> ↑<RTLConditionalNode 108>

<RTLConditionalNode 108>
<162> a149 ← a147
<163> a150 ← a149
<164> a150 ← a150+4
<165> d103 ← access(rom, a150)
<166> d104 ← d103
<167> d104 ← d104&8388608
<168> z ← d104=0
<169> z↑<RTLJumpNode 111>
<170> ↑<RTLJumpNode 109>

<RTLJumpNode 109>
<171> ↑<RTLJumpNode 110>

<RTLJumpNode 110>
<172> d105 ← 0
<173> ↑<RTLJumpNode 112>

<RTLJumpNode 111>
<174> d106 ← access(rom, a149)
<175> d107 ← d106
<176> d107 ← d107&65535
<177> d108 ← d107
<178> d108 ← d108−12
<179> d105 ← d108
<180> d105 ← d105>>2
<181> ↑<RTLJumpNode 112>

<RTLJumpNode 112>
<182> a148 ← d105
<183> ↑<RTLJumpNode 113>

<RTLJumpNode 113>
<184> ↑<RTLJumpNode 114>

<RTLJumpNode 114>
<185> a113 ← a148
<186> ↑<RTLJumpNode 115>

<RTLJumpNode 115>
<187> a151 ← a112
<188> d109 ← 1
<189> ↑<RTLJumpNode 116>

<RTLJumpNode 116>
<190> ↑<RTLJumpNode 117>

<RTLJumpNode 117>
<191> d110 ← a151
<192> d111 ← d109
<193> d110 ← d110<<1
<194> d111 ← d111<<1
<195> d112 ← d110
<196> d112 ← d112+d111
<197> d113 ← d112
<198> d113 ← d113>>1
<199> a152 ← d113
<200> ↑<RTLJumpNode 118>

<RTLJumpNode 118>
<201> ↑<RTLJumpNode 119>

<RTLJumpNode 119>
<202> a119 ← a152
```

```
<203> ↑<RTLJumpNode 120>

<RTLJumpNode 120>
<204> a117 ← a119
<205> ↑<RTLJumpNode 121>

<RTLJumpNode 121>
<206> a112 ← a117
<207> ↑<RTLJumpNode 122>

<RTLJumpNode 122>
<208> a153 ← a113
<209> a154 ← a112
<210> ↑<RTLJumpNode 123>

<RTLJumpNode 123>
<211> ↑<RTLConditionalNode 124>

<RTLConditionalNode 124>
<212> d114 ← a153
<213> d114 ← d114<<1
<214> d115 ← a154
<215> d115 ← d115<<1
<216> n ← d114−d115<0
<217> n↑<RTLJumpNode 127>
<218> ↑<RTLJumpNode 125>

<RTLJumpNode 125>
<219> ↑<RTLJumpNode 126>

<RTLJumpNode 126>
<220> d116 ← NIL+12
<221> ↑<RTLJumpNode 128>

<RTLJumpNode 127>
<222> d116 ← NIL+24
<223> ↑<RTLJumpNode 128>

<RTLJumpNode 128>
<224> a155 ← d116
<225> ↑<RTLJumpNode 129>

<RTLJumpNode 129>
<226> ↑<RTLJumpNode 130>

<RTLJumpNode 130>
<227> a129 ← a155
<228> ↑<RTLJumpNode 131>

<RTLJumpNode 131>
<229> a127 ← a129
<230> ↑<RTLJumpNode 132>

<RTLJumpNode 132>
<231> a125 ← a127
<232> ↑<RTLConditionalNode 133>

<RTLConditionalNode 133>
<233> a157 ← a125
<234> d117 ← NIL
<235> d117 ← d117+12
<236> z ← a157−d117=0
<237> ~z↑<RTLJumpNode 136>
<238> ↑<RTLJumpNode 134>

<RTLJumpNode 134>
<239> d118 ← NIL
<240> d118 ← d118+24
<241> a133 ← d118
<242> ↑<RTLJumpNode 135>

<RTLJumpNode 135>
<243> a156 ← a133
<244> ↑<RTLJumpNode 138>

<RTLJumpNode 136>
<245> d119 ← NIL
<246> d119 ← d119+12
<247> a135 ← d119
<248> ↑<RTLJumpNode 137>

<RTLJumpNode 137>
<249> a156 ← a135
<250> ↑<RTLJumpNode 138>

<RTLJumpNode 138>
<251> a123 ← a156
<252> ↑<RTLConditionalNode 139>

<RTLConditionalNode 139>
```

```
<253> a142 ← a123
<254> a159 ← a142
<255> d120 ← NIL
<256> d120 ← d120+12
<257> z ← a159−d120=0
<258> ˜z↑<RTLJumpNode 142>
<259> ↑<RTLJumpNode 140>

<RTLJumpNode 140>
<260> d121 ← NIL
<261> a144 ← d121
<262> ↑<RTLJumpNode 141>

<RTLJumpNode 141>
<263> a158 ← a144
<264> ↑<RTLJumpNode 166>

<RTLJumpNode 142>
<265> a160 ← a101
<266> a161 ← a112
<267> ↑<RTLJumpNode 143>

<RTLJumpNode 143>
<268> ↑<RTLConditionalNode 144>

<RTLConditionalNode 144>
<269> a163 ← a160
<270> d122 ← a161
<271> d123 ← d122
<272> d123 ← d123<<1
<273> d124 ← 1
<274> d124 ← d124<<1
<275> n ← d123−d124<0
<276> n↑<RTLJumpNode 153>
<277> ↑<RTLJumpNode 145>

<RTLJumpNode 145>
<278> ↑<RTLConditionalNode 146>

<RTLConditionalNode 146>
<279> a164 ← a163
<280> a164 ← a164+4
<281> d125 ← access(rom, a164)
<282> d126 ← d125
<283> d126 ← d126&8388608
<284> z ← d126=0
<285> z↑<RTLJumpNode 149>
<286> ↑<RTLJumpNode 147>

<RTLJumpNode 147>
<287> ↑<RTLJumpNode 148>

<RTLJumpNode 148>
<288> d127 ← 0
<289> ↑<RTLConditionalNode 150>

<RTLJumpNode 149>
<290> d128 ← access(rom, a163)
<291> d129 ← d128
<292> d129 ← d129&65535
<293> d130 ← d129
<294> d130 ← d130−12
<295> d127 ← d130
<296> d127 ← d127>>2
<297> ↑<RTLConditionalNode 150>

<RTLConditionalNode 150>
<298> d131 ← d122
<299> d131 ← d131<<1
<300> d132 ← d127
<301> d132 ← d132<<1
<302> n ← d132−d131<0
<303> n↑<RTLJumpNode 153>
<304> ↑<RTLJumpNode 151>

<RTLJumpNode 151>
<305> ↑<RTLJumpNode 152>

<RTLJumpNode 152>
<306> d122 ← d122<<1
<307> d133 ← d122
<308> d133 ← d133<<1
<309> a163 ← a163+8
<310> a165 ← a163
<311> a165 ← a165+d133
<312> d130 ← access(m, a165)
<313> ↑<RTLJumpNode 154>

<RTLJumpNode 153>
<314> d130 ← 0
```

```
<315> ↑<RTLJumpNode 154>

<RTLJumpNode 154>
<316> a162 ← d130
<317> ↑<RTLJumpNode 155>

<RTLJumpNode 155>
<318> ↑<RTLJumpNode 156>

<RTLJumpNode 156>
<319> a138 ← a162
<320> ↑<RTLJumpNode 157>

<RTLJumpNode 157>
<321> a166 ← a102
<322> a167 ← a138
<323> ↑<RTLJumpNode 158>

<RTLJumpNode 158>
<324> ↑<RTLJumpNode 159>

<RTLJumpNode 159>
<325> d134 ← a166
<326> d135 ← a167
<327> d134 ← d134<<1
<328> d135 ← d135<<1
<329> d136 ← d134
<330> d136 ← d136+d135
<331> d137 ← d136
<332> d137 ← d137>>1
<333> a168 ← d137
<334> ↑<RTLJumpNode 160>

<RTLJumpNode 160>
<335> ↑<RTLJumpNode 161>

<RTLJumpNode 161>
<336> a107 ← a168
<337> ↑<RTLJumpNode 162>

<RTLJumpNode 162>
<338> a105 ← a107
<339> ↑<RTLJumpNode 163>

<RTLJumpNode 163>
<340> a102 ← a105
<341> ↑<RTLJumpNode 115>

<RTLJumpNode 164>
<342> ↑<RTLJumpNode 165>

<RTLJumpNode 165>
<343> a158 ← a146
<344> ↑<RTLJumpNode 166>

<RTLJumpNode 166>
<345> d138 ← NIL
<346> a140 ← d138
<347> ↑<RTLJumpNode 167>

<RTLJumpNode 167>
<348> a169 ← a101
<349> a111 ← a169
<350> ↑<RTLJumpNode 168>

<RTLJumpNode 168>
<351> a170 ← a102
<352> d139 ← a170
<353> ↑<RTLReturnNode 169>

<RTLReturnNode 169>
<354> return d139
```

# Appendix B

# An Optimized RTL Program

The result of sending the optimize message to the instance of **RTLProgram** shown in Appendix A appears below.

```
<RTLJumpNode 102>
<100> SP.1 ← ---
<101> CONTEXT.1 ← ---
<102> NIL.1 ← ---
<103> m.1 ← ---
<104> rom.1 ← ---
<105> a171.1 ← ---
<106> ↑<RTLConditionalNode 108>

<RTLConditionalNode 108>
<154> d224 ← 0
<164> a173.2 ← a171.1+4
<165> d182.2 ← access(rom.1, a173.2)
<167> d184.2 ← d182.2&8388608
<168> z.2 ← d184.2=0
<169> z.2↑<RTLJumpNode 111>
<170> ↑<RTLJumpNode 110>

<RTLJumpNode 111>
<174> d185.3 ← access(rom.1, a171.1)
<176> d187.3 ← d185.3&65535
<178> d189.3 ← d187.3-12
<180> d191.3 ← d189.3>>2
<181> ↑<RTLConditionalNode 173>

<RTLJumpNode 110>
<1073> d193.2 ← 0
<173> ↑<RTLConditionalNode 173>

<RTLConditionalNode 173>
<1016> d192.3 ← phi( d191.3 d193.2 )
<411> d195.2 ← d192.3<<1
<414> d196.2 ← 2
<415> n.2 ← d195.2-2<0
<388> n.2↑<RTLJumpNode 177>
<389> ↑<RTLJumpNode 180>

<RTLJumpNode 177>
<433> ↑<RTLReturnNode 169>

<RTLJumpNode 180>
<1074> d206.2 ← 0
<309> a179.3 ← a171.1+8
<454> ↑<RTLConditionalNode 124>

<RTLJumpNode 181>
<332> d215.3 ← d214.3>>1
<1075> d225 ← d215.3
<455> ↑<RTLReturnNode 169>

<RTLReturnNode 169>
<1042> d223 ← phi( d224 d225 )
<1076> d222.2 ← d223
<354> return d222.2

<RTLConditionalNode 124>
<1061> d208.3 ← phi( d206.2 d214.3 )
<1057> d207.3 ← phi( d196.2 d216.3 )
<308> d210.3 ← d207.3<<1
<311> a180.3 ← a179.3+d210.3
<312> d212.3 ← access(m.1, a180.3)
<328> d213.3 ← d212.3<<1
```

```
<330> d214.3 ← d208.3+d213.3
<196> d216.3 ← d207.3+2
<216> n.6 ← d195.2−d216.3<0
<217> n.6↑<RTLJumpNode 181>
<218> ↑<RTLJumpNode 172>

<RTLJumpNode 172>
<387> ↑<RTLConditionalNode 124>
```

# Appendix C

# A Scheduled RTL Program

The result of sending the **schedule** message to the instance of **RTLProgram** shown in in Appendix B appears below.

```
<RTLJumpNode 102>
<100> SP.1 ← − − −
<101> CONTEXT.1 ← − − −
<102> NIL.1 ← − − −
<103> m.1 ← − − −
<104> rom.1 ← − − −
<105> a171.1 ← − − −
<106> ↑<RTLConditionalNode 108>

<RTLConditionalNode 108>
<1077> d182.2 ← access(rom.1, a171.1+4)
<154> d224 ← 0
<1081> z.2 ← d182.2&8388608=0 ;
    d184.2 ← d182.2&8388608
<169> z.2↑<RTLJumpNode 111>
<170> ↑<RTLJumpNode 110>

<RTLJumpNode 111>
<174> d185.3 ← access(rom.1, a171.1)
<176> d187.3 ← d185.3&65535
<178> d189.3 ← d187.3−12
<180> d191.3 ← d189.3>>2
<181> ↑<RTLConditionalNode 173>

<RTLJumpNode 110>
<1073> d193.2 ← 0
<173> ↑<RTLConditionalNode 173>

<RTLConditionalNode 173>
<1016> d192.3 ← phi( d191.3 d193.2 )
<411> d195.2 ← d192.3<<1
<414> d196.2 ← 2
<415> n.2 ← d195.2−2<0
<388> n.2↑<RTLJumpNode 177>
<389> ↑<RTLJumpNode 180>

<RTLJumpNode 177>
<433> ↑<RTLReturnNode 169>

<RTLJumpNode 180>
<309> a179.3 ← a171.1+8
<1074> d206.2 ← 0
<454> ↑<RTLConditionalNode 124>

<RTLJumpNode 181>
<1106> d225 ← d214.3>>1
<455> ↑<RTLReturnNode 169>

<RTLReturnNode 169>
<1042> d223 ← phi( d224 d225 )
<1076> d222.2 ← d223
<354> return d222.2

<RTLConditionalNode 124>
<1061> d208.3 ← phi( d206.2 d214.3 )
<1057> d207.3 ← phi( d196.2 d216.3 )
<196> d216.3 ← d207.3+2
<1109> d212.3 ← access(m.1,
    a179.3+d207.3<<1)
<328> d213.3 ← d212.3<<1
<330> d214.3 ← d208.3+d213.3
<216> n.6 ← d195.2−d216.3<0
```

```
<217> n.6↑<RTLJumpNode 181>
<218> ↑<RTLJumpNode 172>

<RTLJumpNode 172>
<387> ↑<RTLConditionalNode 124>
```

# Appendix D

# 68020 Code

The result of sending the **assign** message to the instance of **RTLProgram** shown in Appendix C appears below.

```
00  move.l  a7,a5
02  addq.l  4,a5
04  move.l  (4,a0),d0
08  moveq.l 0,d2
0A  and.l   #8388608,d0
10  bne     28
14  move.l  (a0),d0
16  and.l   #65535,d0
1C  sub.l   #12,d0
22  lsr.l   2,d0
24  bra     2A
28  moveq.l 0,d0
2A  add.l   d0,d0
2C  moveq.l 2,d1
2E  cmp.l   #2,d0
34  bmi     5C
38  add.l   #8,a0
3E  moveq.l 0,d2
40  move.l  d1,d3
42  add.l   #2,d3
48  move.l  (0,a0,d1),d4
4C  add.l   d4,d4
4E  add.l   d4,d2
50  cmp.l   d3,d0
52  bmi     5A
56  move.l  d3,d1
58  bra     40
5A  lsr.l   1,d2
5C  move.l  d2,d0
5E  add.l   (4,a5),a5
62  rts
```

# Appendix E

# NATIONAL 32000 INSTRUCTION TABLE

The **N32KInstruction class** methods **initializeN32KMachineLanguage** and **initializeN32KJumps** initialize the instruction table for the National 32000 series.

**initializeN32KMachineLanguage**

```
    "the order of accessList should correspond to the numerical order
     of the variables used to define the rtl. In other words, accessList[1]
     should be the access class for variable $1. Also variable $1 should
     correspond to the operand that appears left–most in the code for
     the instruction in the manual."

    "ADDQi"
    self
        rtl: '$1 ← $1 + !2'
        flags: 'c ← $1 ?+ !2; f ← $1 !+ !2'
        code: 12
        codeBits: 7
        sizeOffset: 0
        accessList: #(rmw quick ).
    "ADDi"
    self
        rtl: '$2 ← $2 + $1'
        flags: 'c ← $2 ?+ $1; f ← $2 !+ $1'
        code: 0
        codeBits: 6
        sizeOffset: 0
        accessList: #(read rmw ).
    "ADDCi"
    self
        rtl: '$2 ← $2 + $1 + c'
        flags: 'c ← $2 ?+ $1 ?+ c; f ← $2 !+ $1 !+ c'
```

```
    code: 16
    codeBits: 6
    sizeOffset: 0
    accessList: #(read rmw ).
"SUBi"
self
    rtl: '$2 ← $2 − $1'
    flags: 'c ← $2 ?− $1; f ← $2 !− $1'
    code: 32
    codeBits: 6
    sizeOffset: 0
    accessList: #(read rmw ).
"SUBCi"
self
    rtl: '$2 ← $2 − $1 − c'
    flags: 'c ← $2 ?− $1; f ← $2 !− $1'
    code: 48
    codeBits: 6
    sizeOffset: 0
    accessList: #(read rmw ).
"ANDi"
self
    rtl: '$2 ← $2 & $1'
    flags: ''
    code: 40
    codeBits: 6
    sizeOffset: 0
    accessList: #(read rmw ).
"ORi"
self
    rtl: '$2 ← $2 | $1'
    flags: ''
    code: 24
    codeBits: 6
    sizeOffset: 0
    accessList: #(read rmw ).
"MULi"
self
    rtl: '$2 ← $2 * $1'
    flags: ''
    code: 8398
    codeBits: 14
    sizeOffset: 8
    accessList: #(read rmw ).
"DIVi"
```

```
self
    rtl: '$2 ← $2 / $1'
    flags: ''
    code: 15566
    codeBits: 14
    sizeOffset: 8
    accessList: #(read rmw ).
"NEGi"
self
    rtl: '$2 ← − $1'
    flags: 'c ← 0 ?− $1; f ← 0 !− $1'
    code: 8270
    codeBits: 14
    sizeOffset: 8
    accessList: #(read write ).
"LSHi"
self
    rtl: '$2 ← $2 << $1'
    flags: ''
    code: 5198
    codeBits: 14
    sizeOffset: 8
    accessList: #(read rmw ).
"ADJSPi"
self
    rtl: 'SP ← SP − $1'
    flags: ''
    code: 1404
    codeBits: 11
    sizeOffset: 0
    accessList: #(read).
"LPRi"
self
    rtl: '$2 ← $1'
    flags: ''
    code: 108
    codeBits: 7
    sizeOffset: 0
    accessList: #(read short).
"SPRi"
self
    rtl: '$1 ← $2'
    flags: ''
    code: 44
    codeBits: 7
```

```
        sizeOffset: 0
        accessList: #(read short).
    "MOVQi"
    self
        rtl: '$1 ← !2'
        flags: ''
        code: 92
        codeBits: 7
        sizeOffset: 0
        accessList: #(write quick ).
    "MOVi"
    self
        rtl: '$2 ← $1'
        flags: ''
        code: 20
        codeBits: 6
        sizeOffset: 0
        accessList: #(read write ).
    "CMPQi"
    self
        rtl: ''
        flags: 'z ← ($1 − !2) = 0; n ← ($1 − !2) < 0'
        code: 28
        codeBits: 7
        sizeOffset: 0
        accessList: #(read quick ).
    "CMPi"
    self
        rtl: ''
        flags: 'z ← ($2 − $1) = 0; n ← ($2 − $1) < 0'
        code: 4
        codeBits: 6
        sizeOffset: 0
        accessList: #(read read )
```

**initializeN32KJumps**

```
    "the order of accessList should correspond to the numerical order
     of the variables used to define the rtl. In other words, accessList[1]
     should be the access class for variable $1."
    "The l flag has been written as o because l is reserved
     by the parser for labels."

    "ENTER"
    self
        rtl: 'SP ← SP − 4; *(SP − 4) ← CONTEXT; CONTEXT ← SP − 4'
```

```
    flags: ''
    code: 130
    codeBits: 8
    accessList: #().
"EXIT"
self
    rtl: 'SP ← CONTEXT + 4; CONTEXT ← *(CONTEXT)'
    flags: ''
    code: 146
    codeBits: 8
    accessList: #().
"RET"
self
    rtl: '@'
    flags: ''
    code: 18
    codeBits: 16
    accessList: #().
"Bcond"
N32KBranchInstruction
    rtl: 'z ↑ l$1'
    code: 10
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: '~z ↑ l$1'
    code: 26
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: 'c ↑ l$1'
    code: 42
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: '~c ↑ l$1'
    code: 58
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: 'o ↑ l$1'
    code: 74
    codeBits: 8.
"Bcond"
N32KBranchInstruction
```

```
    rtl: '~o ↑ l$1'
    code: 90
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: 'n ↑ l$1'
    code: 106
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: '~n ↑ l$1'
    code: 122
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: 'f ↑ l$1'
    code: 138
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: '~f ↑ l$1'
    code: 154
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: '((~z) & (~o)) ↑ l$1'
    code: 170
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: '(z | o) ↑ l$1'
    code: 186
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: '((~z) & (~n)) ↑ l$1'
    code: 202
    codeBits: 8.
"Bcond"
N32KBranchInstruction
    rtl: '(z | n) ↑ l$1'
    code: 218
    codeBits: 8.
"BR"
N32KBranchInstruction
```

rtl: '↑ l$1'
code: 234
codeBits: 8.
*“BSR”*
N32KBranchInstruction
rtl: '↑↑ $1'
code: 2
codeBits: 8

# Bibliography


[ASU86] Alfred C. Aho, Ravi Seth, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proceedings of the SIGPLAN '89 Symposium on Programming Language Design and Implementation, SIGPLAN Notices*, pages 275–284, 1989.

[BGG*89] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. In *Proceedings of the SIGPLAN '89 Symposium on Programming Language Design and Implementation, SIGPLAN Notices*, pages 258–263, 1989.

[CAC*81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[CFR*89] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[CFR*91] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CH84] Fred C. Chow and John L. Hennessy. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, pages 222–232, 1984.

[Cha82] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, pages 98–105, 1982.

[Cof76] E. G. Coffman. *Computer and Job-shop Scheduling Theory*. Wiley, 1976.

[Dav81] Jack W. Davidson. *Simplifying Code Generation Through Peephole Optimizations.* PhD thesis, University of Arizona, 1981.

[Dav85] Jack W. Davidson. *Simple Machine Description Grammars.* Technical Report, Department of Computer Science, University of Virgina, 1985.

[Dav86] Jack W. Davidson. A retargetable instruction reorganizer. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, pages 234–241, 1986.

[DF80] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[DF84] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, 1984.

[FW88] Christopher W. Fraser and Alan L. Wendt. Integrating code generation and optimizations. In *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation, SIGPLAN Notices*, pages 79–84, 1988.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Massachusetts, 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[Gro83] Thomas K. R. Gross. *Code Optimization of Pipeline Constraints.* PhD thesis, Stanford University, 1983.

[HD89] Robert R. Henry and Peter C. Damron. *Algorithms for Table-Driven Code Generators Using Tree-Pattern Matching.* Technical Report, Department of Computer Science, U. of Washington, February 1989.

[Hen87] Robert R. Henry. *Code Generation by Table Lookup.* Technical Report, Department of Computer Science, U. of Washington, July 1987.

[Hsu87] Wei-chung Hsu. *Register Allocation and Code Scheduling for Load/Store Architectures.* PhD thesis, University of Wisconsin at Madison, 1987.

[Hun87] Colin Hunter. *Series 32000 Programmer's Reference Manual.* National Semiconductor Corporation, 1987.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An optimizing compiler for Smalltalk. In *Proceedings of OOPSLA '88, Object-Oriented Programming Systems, Languages and Applications*, pages 18–26, November 1988. printed as SIGPLAN Notices, 23(11).

[Kuc78] D. J. Kuck. *The Structure of Computers and Computations.* John Wiley and Sons, 1978.

[LH86] James R. Larus and Paul N. Hilfinger. Register allocation in the SPUR Lisp compiler. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, pages 255–263, 1986.

[Mot85] Inc. Motorola. *MC68020 32-Bit Microprocessor User's Manual.* second edition, 1985.

[Nic90] Brian R. Nickerson. Graph coloring register allocation for processors with multi-register operands. In *Proceedings of the SIGPLAN '90 Symposium on Programming Language Design and Implementation, SIGPLAN Notices*, pages 40–52, 1990.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[Sta89] Richard M. Stallman. *Using and Porting GNU CC.* Technical Report, Free Software Foundation, 1989.

[SU70] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the Association for Computing Machinery*, :715–728, 1970.

[Tar74] R. E. Tarjan. Testing flow graph reducibility. *J. Comp. Sys. Sci.*, 9:355–365, 1974.

[Tie89] Michael D. Tiemann. *The GNU Instruction Scheduler.* Technical Report, Stanford University, June 1989.

[Wal86] David W. Wall. Global register allocation at link time. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, pages 264–275, 1986.

[Wie87] John David Wiegand. *An Object-oriented Code Optimizer and Generator.* Master's thesis, University of Illinois, Urbana-Champaign, 1987.