Rascl Intermediate Form (RIF) Developer's Guide

Introduction

RIF is an Intermediate Form comprised of quadruples (3-address code) that resembles MIPS instructions but can be easily translated to most any register-based architecture. RIF is small and simple making it easy to learn and apply to a compiler project.

RIF allows for an infinite number of registers expressed as named temporary variables. This frees the parser and semantic analyzer from register allocation decisions that are more specific to the target architecture.

Three address code takes the form:

operator, operand1, operand2, result

The field meanings are:

- **operator** – An operation to perform. This may also contain a directive name (see Directives section below)
- **operand1, operand2** – These are inputs to the operation. These can be identifier names, immediate integer values, or named temporaries.
- **result** – This is the target where the operation will store its result. This can also be the target label of a conditional or unconditional branch (a quad number). For directives, this may be a string representing a label or a variable.

How the operations and arguments are stored internal to the compiler is implementation dependent. For purposes of dumping intermediate code to a file:

- Each quadruple should be written to a single line of the file
- Each line should begin with a 'quadruple index' that reflects its sequence in the generated code
- Fields (operator, operand1, etc) should be separated by a comma (,)
- Operators should be stored as a mnemonic string as shown in this document (See Instructions section below)
- Operands and result fields should be strings (for identifiers, named temporaries, labels, etc.) or numeric strings (for integer and float constants)

Temporaries

Instead of generating an architecture specific register ($s0, $s1, etc), RIF uses simple 'temporaries' that generally represent a register. Later, the code generator can select registers appropriate to the operations so that burden is removed from intermediate code generation.

There are two types of temporaries: integers temporaries and floating point temporaries.

Integer temporaries are formatted as strings beginning with 'T' and have an appended 3-5 digit sequence number. The sequence number for a temporary should never be repeated during RIF intermediate code generation. Float temporaries can have a separate sequence generator. They start with 'F' and have an appended 3-5 digit sequence number.

So each time a temporary is needed, generate the next one for the type of register needed (T001, T002, T003, etc. and F001, F002, F003, etc.)

Instructions

Following is a list of instruction operators allowed in RIF:

- **Load and Store instructions**: li (load immediate), lw (load word), sw (store word)
- **Arithmetic instructions**: add, addi, sub, subi, div, mul, fadd, fsub, fdiv, fmul, and, andi, or, ori, neg
- **Conversion routines**: tf (to float), ti (to int)
- **Conditional branch instructions**: beq, bne, bgt, bge, blt, ble
- **Unconditional jump**: j
- **Shift operators**: sl (shift left), sr (shift right)
- **System Traps**: syscall

Following are details on each instruction.

Load and Store Instructions

li – Load Immediate

This operation loads a constant integer into a register.

li, <tempname>, 0, <immediate int>

la – Load address

This loads the address of an identifier into a temporary register.

la, <tempname>, 0, <identifier>

The above operation loads the address of the named identifier into the register tempname.

lw – Load Word

This operation loads a value from memory. There are two forms:

lw, <tempname >, 0, <identifier>

This will cause a load of the identifier (wherever it is in memory) to a temporary (which later becomes a machine register during code generation).

lw, <resttempname>, 0, <tempname>

This assumes that an address in memory was calculated into <tempname> (which later becomes a machine register) and loads what is at that address into <restempname> (which also becomes a machine register during code generation).

lw, <resttempname>, 0, (B<+/->#)

Here, # is n offset in bytes from the base pointer. B is the basepointer and the symbol + or – tell whether the value is added or subtracted from the base.

sw – Store Word

This operation stores a register into memory. There are two forms of the instruction.

sw, <tempname>, 0, <identifier>

This will cause a store of <tempname> (which later becomes a machine register during code generation) into the identifier (wherever it is in memory).

sw, <tempname>, 0, <resttempname>

This assumes that an address in memory was calculated into <restempname> (which later becomes a machine register) and stores the value in <tempname> into the address indicated in <restempname>.

Arithmetic Instructions

The arithmetic instructions include all of the basic arithmetic operators some of which allow forms with immediate operands.

add – Add integers

This operation adds two integers stored in temporary registers. The computed result is operand1_temp + operand2-temp.

add, <operand1_temp>, <operand2_temp>, <result_temp>

addi – Add integer with immediate operand

This operation adds an immediate integer value to a temporary register

addi, <oper1_temp>, <integer_constant>, <res_temp>

sub – Subtract integers

This operation subtracts two integer values. It takes two temporaries and the result is a temporary. The computed result is operand1_temp – operand2-temp.

sub, <opernd1_temp>, <operand2_temp>, <res_temp>

subi – Subtract integer with immediate operand

This operation subtracts an immediate integer value from a temporary and places the result in a temporary. The computed result is operand1_temp – integer_constant.

subi, <operand1_temp>, <integer_constant>, <res_temp>

div – Integer divide

This operation divides two integers (the integers come from temporaries). It places the result (of an integer divide operation) into another temporary. The computed result is opernad1_temp / operand2_temp.

div, <operand1_temp>, <operand2_temp>, <res_temp>

mul – Integer multiply

This operation multiplies two integers (from temporaries) and places the result in a temporary. The computed result is opernad1_temp * operand2_temp.

mul, <operand1_temp>, < operand2_temp >, <res_temp>

fadd – Floating point add

This operation adds two floating point numbers (from temporaries) and places the sum into a floating point temporary. The computed result is fopernad1_temp + foperand2_temp. (floating point addition)

fadd, <foperand1_temp>, <foperand2_temp >, <res_ftemp>

fsub – Floating point subtract

This operation subtracts two floating point numbers (from temporaries) and places the difference into a floating point temporary. The computed result is fopernad1_temp - foperand2_temp. (floating point subtraction)

fsub, <foperand1_temp>, <foperand2_temp>, <res_ftemp>

fdiv – Floating point divide

This operation divides two floating point numbers (from temporaries) and places the quotient into a floating point temporary. The computed result is fopernad1_temp / foperand2_temp. (floating point division)

fdiv, <foperand1_temp>, <foperand2_temp>, <res_ftemp>

fmul – Floating point multiply

This operation multiplies two floating point numbers (from temporaries) and places the product into a floating point temporary. The computed result is fopernad1_temp * foperand2_temp. (floating point multiplication)

fmul, <foperand1_temp>, <foperand2_temp>, <res_ftemp>

Conversion Instructions

toInt – Convert float to integer [with truncation]

This operation converts a floating point to an integer. The source float is a float temporary in arg1. The converted value is placed into an integer temporary in the result field. The arg2 field is not used. The converted value is likely to lose some information as the resulting integer will be truncated (not rounded).

toInt, <foperand1_temp>, 0, <res_temp>

toFloat – Convert integer to float

This operation converts and integer to a floating point number. The source integer is a temporary in arg1. The converted value is placed into a float temporary in the result field. The arg2 field is not used.

toInt, <operand1_temp>, 0, <res_ftemp>

Conditional Branches

beq – Branch if equal

This operation compares temporaries from arg1 and arg2. If they are equal, it branches to the quad indexed by the result field.

beq, <operand1_temp>, <operand2_temp>, <quad_index>

bne – Branch if not equal

This operation compares temporaries from arg1 and arg2. If they are not equal, it branches to the quad indexed by the result field.

bne, <operand1_temp>, <operand2_temp>, <quad_index>

bgt – Branch if greater than

This operation compares temporaries from arg1 and arg2. If arg1 is greater than arg2, it branches to the quad indexed by the result field.

bgt, <operand1_temp>, <operand2_temp>, <quad_index>

bge –Branch if greater than or equal

This operation compares temporaries from arg1 and arg2. If arg1 is greater than or equal to arg2, it branches to the quad indexed by the result field.

bge, <operand1_temp>, <operand2_temp>, <quad_index>

blt – Branch if less than

This operation compares temporaries from arg1 and arg2. If arg1 is less than arg2, it branches to the quad indexed by the result field.

blt, <operand1_temp>, <operand2_temp>, <quad_index>

ble – Branch if less than or equal

This operation compares temporaries from arg1 and arg2. If arg1 is less than or equal to arg2, it branches to the quad indexed by the result field.

ble, <operand1_temp>, <operand2_temp>, <quad_index>

Unconditional Jumps

There is only one unconditional jump instruction: j.

J – Jump

This operation causes control to transfer to the indicated label (the quad numberi nthe result field).

j, 0, 0, <quad_index>

Shift Operators

sl – Shift Left

This operation shifts the value in a temporary 0-31 bits to the left filling 0 bits on the right. The result is placed in the temporary from the result field.

sl, <operand1_temp>, <immediate_shift_amount>, <res_temp>

sr – Shift Right

This operation shifts the value in a temporary 0-31 bits to the right, filling in 0 bits on the left side. The result is placed in the temporary from the result field.

sr, <operand1_temp>, <immediate_shift_amount>, <res_temp>

Subroutine calls

Several instructions are provided for calling functions with parameters and handling return values from those functions.

Regardless of parameter passing conventions (some architectures pass parameters on a stack, some pass them in registers), RIF just uses a list of parameters that can be an identifier, a temporary name holding a computed address, or a value computed into a temporary.

param

The param instruction is used to add a parameter to the parameter list to call a function. The form of the instruction is:

param, 0, 0, <identifier>
param, 0, 0, <tempname>
param, 0, 0, (<tempname>)

The first form adds the value from an identifier to the parameters list (this can be loaded from memory and pushed onto the stack, or put into a special register in the final generated code. RIF doesn't use that level of detail).

The second form adds a value that was computed into a temporary (i.e. a machine register) to the parameters list.

The third form adds a value at the address stored (computed into) <tempname>. This is useful for adding array elements to a parameter list.

call

This instruction is used to generate the appropriate control transfer to the target function.

call, <#>, <tempName>, <.segment, 0, 0, .data
.int, 0, 5, a
.int, 0, 1, blabel>

The number in argument 1 is the number of parameters that the function takes. The <label> is that name of the label generated at the top of the function's code. There should be preceding *param* instructions for each parameter. So if argument 1 is 5, there should be 5 *param* instructions immediately preceding the *call*.
<tempName> is the temporary where the result of the function will be stored.

Referencing values inside of the function can be done by using an offset from a base-pointer (this base-pointer does not need to be set up. It will be assumed as set up in prolog code for the function when final code generation happens.
Assume the first argument is 8 bytes from the stack pointer. Since both integers and floats take 4 bytes, each argument is 4 bytes further (12, 16, etc). The reason we start at 8 is that immediately above the base pointer would be the saved base pointer and just above that is the return address.

The form you would use in a load word is:

lw, <param1_temp>, 0, (B-8)

which would move the first parameter into a newly allocated temporary register.

return

This instruction is used to specify a return value. It can also just indicate a return of control to the caller.

return, 0, 0, <tempname>
Return to the caller. The value to return is in the temporary <tempname>.

return, 0, 0, 0

Simply return control to the caller with no return value.

System Traps

A system trap is an instruction designed to interrupt operations and call operating system support routines. The intermediate instruction available is syscall (similar to a trap provided in the MIPS simulator SPIM).

syscall

The syscall operation sets up and dispatches to the operating system or a library to handle a request. Rascl needs 4 different operations: read integer, write integer, read float, and write float). The trap numbers for these operations are 1,2,3, and 4, respectively. The argument provided should indicate a temporary where the integer or float to print is currently stored. For the read operations, it represents the destination temporary from the read. The result field is not used.

syscall, <trapNumber>, <arg>, 0

Note that the register (or temporary) listed in arg should match the type being read or written (so T# for integer values, FT# for float values).

Directives

RIF also provides directives for memory allocation, label specification, and segment specification. These include:

- **.label** – Emit a label at this point in the code. This will be used as branch and jump targets.
- **.segment** [text, data, bss] – Start a segment of the named type
- **.int** – Reserve space for a 32 bit integer
- **.float** – Reserve space for a float

Examples

NOTE: The examples are based on an earlier version of Rascl **without** functions or a main. They show the translations of most of the basic features and the code that should be generated for them. This includes array references, if statements and while loops.

Program 1

```
int a[5], b;
float c, d[4];
{
  a[2] = 5;
  a[3] = 10;
  b = a[2] + a[3] * 2;
  while (b > 0)
  {
    print b;
```

```
   if (b < 10)
   {
     b = b - 1;
   }
   else
   {
     b = b - 2;
   };
  };
  print a[2];
}
```

Equivalent Intermediate code in RIF:

```
.segment, 0, 0, .data
.int, 0, 5, a
.int, 0, 1, b
.float, 0, 1, c
.float, 0, 4, d


.segment, 0, 0, .text
# Start ASSIGN statement ---
li, T0, 0, 8
la, T1, 0, a
add, T0, T1, T2
li, T4, 0, 5
sw, T4, 0, T2
# Start ASSIGN statement ---
li, T5, 0, 12
la, T6, 0, a
add, T5, T6, T7
li, T9, 0, 10
sw, T9, 0, T7
# Start ASSIGN statement ---
la, T11, 0, b
li, T12, 0, 8
la, T13, 0, a
add, T12, T13, T14
lw, T15, 0, T14
li, T16, 0, 2
la, T17, 0, a
add, T16, T17, T18
lw, T19, 0, T18
li, T20, 0, 2
mul, T19, T20, T21
add, T15, T21, T22
sw, T22, 0, T11
```

```
# Start WHILE statement ---
.label, 0, 0, L2
la, T24, 0, b
lw, T23, 0, T24
li, T25, 0, 0
bgt, T23, T25, L1
j, 0, 0, L0
.label, 0, 0, L1
# Start PRINT statement ---
la, T27, 0, b
lw, T26, 0, T27
syscall, 2, T26, 0
# Start if statement ---
la, T29, 0, b
lw, T28, 0, T29
li, T30, 0, 10
blt, T28, T30, L5
j, 0, 0, L4
# Start if statement THEN part ---
.label, 0, 0, L5
# Start ASSIGN statement ---
la, T32, 0, b
la, T34, 0, b
lw, T33, 0, T34
li, T35, 0, 1
sub, T33, T35, T36
sw, T36, 0, T32
# Start if statement ELSE part ---
.label, 0, 0, L4
# Start ASSIGN statement ---
la, T38, 0, b
la, T40, 0, b
lw, T39, 0, T40
li, T41, 0, 2
sub, T39, T41, T42
sw, T42, 0, T38
j, 0, 0, L3
.label, 0, 0, L3
.label, 0, 0, L0
# Start PRINT statement ---
li, T43, 0, 8
la, T44, 0, a
add, T43, T44, T45
lw, T46, 0, T45
syscall, 2, T46, 0
```

[Note: Comments starting with # are not needed in your output. I implemented this and it is a 'nicety' if you wish to add it but not essential]

Program 2

```
float a;
float g[10];
int b, c;
int d[5];

{
  a = 5.;
  b = 1;
  if (a == b)
  {
    d[a] = 5;
    g[0] = a + -b;
  }
  else
  {
    while (b < 5) {
       g[1] = -a * b;
           b = b + 1;
    };
  };
  print c;
}
```

Equivalent Intermediate code in RIF:

```
.segment, 0, 0, .data
.float, 0, 10, g
.float, 0, 1, a
.int, 0, 1, b
.int, 0, 1, c
.int, 0, 5, d


.segment, 0, 0, .text
# Start ASSIGN statement ---
la, T0, 0, a
li, FT1, 0, 5.0
sw, FT1, 0, T0
# Start ASSIGN statement ---
la, T2, 0, b
li, T3, 0, 1
sw, T3, 0, T2
# Start if statement ---
la, T4, 0, a
lw, FT2, 0, T4
la, T6, 0, b
lw, T5, 0, T6
```

```
beq, FT2, T5, L2
j, 0, 0, L1
# Start if statement THEN part ---
.label, 0, 0, L2
# Start ASSIGN statement ---
lw, T7, 0, a
sl, T7, 2, T7
la, T8, 0, d
add, T7, T8, T9
li, T11, 0, 5
sw, T11, 0, T9
# Start ASSIGN statement ---
li, T12, 0, 0
la, T13, 0, g
add, T12, T13, T14
la, T15, 0, a
lw, FT4, 0, T15
la, T17, 0, b
lw, T16, 0, T17
li, T18, 0, 0
sub, T18, T16, T19
fadd, FT4, FT6, FT5
sw, FT5, 0, T14
# Start if statement ELSE part ---
.label, 0, 0, L1
# Start WHILE statement ---
.label, 0, 0, L5
la, T21, 0, b
lw, T20, 0, T21
li, T22, 0, 5
blt, T20, T22, L4
j, 0, 0, L3
.label, 0, 0, L4
# Start ASSIGN statement ---
li, T23, 0, 4
la, T24, 0, g
add, T23, T24, T25
la, T26, 0, a
lw, FT8, 0, T26
li, FT9, 0, 0.0
fsub, FT9, FT8, FT10
la, T28, 0, b
lw, T27, 0, T28
fmul, FT10, FT12, FT11
sw, FT11, 0, T25
# Start ASSIGN statement ---
la, T30, 0, b
la, T32, 0, b
lw, T31, 0, T32
li, T33, 0, 1
```

```
add, T31, T33, T34
sw, T34, 0, T30
.label, 0, 0, L3
j, 0, 0, L0
.label, 0, 0, L0
# Start PRINT statement ---
la, T36, 0, c
lw, T35, 0, T36
syscall, 2, T35, 0
```

Additional notes on code generation

Compilers can select instructions in different ways that are still semantically correct. The above examples are only that, examples. The examples show generating two instructions to load a value: *la* to load the address of a variable, then *lw* to load the value from that memory location. Semantically, RIF allows you to generate a single instruction *lw,* which will load the value from memory into a 'temporary'. There can be other variances in code output depending on the compiler's choice of instructions sequences.