Rascl Intermediate Form (RIF)

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 compile 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 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. 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 (,)
- Operands and result fields should be strings (for identifiers, named temporaries, labels, etc.) or numeric strings (for integer and float constants)

Comments

Comments may be placed anywhere in the code. Comments take an entire line and start with a hash symbol (#).

Operands and Labels

Operands for arithmetic instructions can be:

- An Integer or floating point constant (coded as a string)
- A 'register'
    - A new register is generated for every intermediate result.
    - Integer registers are generated as T# where # is an integer. There are an unlimited number of registers.
    - Floating point registers are generated as FT# where # is an integer. There are an unlimited number of registers.
- Labels are generated when needed and have the form L# where # is an integer. There are an unlimited number of labels.

Instructions

Following is a list of instruction operators allowed in RIF:

- **Load and Store instructions**: li (load immediate), lw (load word), sw (store word), lwi (load word indexed), swi (store word indexed)
- **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, <immediate int>, 0, <tempname>

lw – Load Word

This operation loads a value from memory.

lw, <identifier>, 0, <tempname>

lwi – Load Word Indexed

This operation loads a value from an array element in memory. The base address of the array is an identifier or temporary. Arg2 is the index of the desired element (0 base and NOT size adjusted).

lwi, <identifier>, <index>, <tempname>

sw – Store Word

This operation stores a register into memory.

sw, <temp1>, 0, <identifier>

swi – Store Word Indexed

This operation stores a value from a temporary into an array element in memory. The base address of the array is an identifier or temporary. Arg2 is the index of the desired element (0 base and NOT size adjusted).

swi, <identifier>, <index>, <tempname>

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.

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.

sub, <temp1>, <temp2>, <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

subi, <temp1>, <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.
div, <temp1>, <temp2>, <res_temp>

mul – Integer multiply

This operation multiplies two integers (from temporaries) and places the result in a temporary.

mul, <temp1>, <temp2>, <res_temp>

fadd – Floating point add

This operation adds two floating point numbers (from temporaries) and places the sum into a floating point temporary.

fadd, <ftemp1>, <ftemp2>, <res_ftemp>

fsub – Floating point subtract

This operation subtracts two floating point numbers (from temporaries) and places the difference into a floating point temporary.

fsub, <ftemp1>, <ftemp2>, <res_ftemp>

fdiv – Floating point divide

This operation divides two floating point numbers (from temporaries) and places the quotient into a floating point temporary.

fdiv, <ftemp1>, <ftemp2>, <res_ftemp>

fmul – Floating point multiply

This operation multiplies two floating point numbers (from temporaries) and places the product into a floating point temporary.

fmul, <ftemp1>, <ftemp2>, <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, <ftemp1>, 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, <temp1>, 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, <temp1>, <temp2>, <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, <temp1>, <temp2>, <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, <temp1>, <temp2>, <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, <temp1>, <temp2>, <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, <temp1>, <temp2>, <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, <temp1>, <temp2>, <quad_index>

Unconditional Jumps

There is only one unconditional jump instruction: j.

J - Jump

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, <temp1>, <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, <temp1>, <immediate_shift_amount>, <res_temp>

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

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
- **.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

.label

This has the form:

.label, 0, 0, L#

It indicates a label for the target of an unconditional or conditional branch.

.segment

This directive indicates the start of a segment. The two supported segments are '.text' for code and '.data' for data declarations.

The form is:

.segment, 0, 0, <segment_name>

.float, .int

These reserve space for variables of type float or int. The form is:

.float, 0, <count>, <name>
.int, 0, <count>, <name>

<count> is the number of values of that type. Usually, it is 1 for a regular variable or some larger count for an array of that type. *<name>* is the variable name associated with the space.

Examples

Following are some examples of RIF code generated for different Rascl source files. Ignore the text between the lines '====='. That is there to show the name of the source or generated file.

=====
T00_rascl_test_exprs1.rsc
=====

```
int a, b;
float c, d;
{
  a = 5;
  b = a + 2 * (-b + -a);
}
```

Equivalent Intermediate code in RIF:

=====
T00_rascl_test_exprs1.rso
=====

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

.segment, 0, 0, .text
# Start ASSIGN statement ---
la, a, 0, T1
li, 5, 0, T2
sw, T2, 0, T1
# Start ASSIGN statement ---
la, b, 0, T4
la, a, 0, T6
lw, T6, 0, T5
li, 2, 0, T7
la, b, 0, T9
lw, T9, 0, T8
li, 0, 0, T10
sub, T10, T8, T11
la, a, 0, T13
lw, T13, 0, T12
li, 0, 0, T14
sub, T14, T12, T15
add, T11, T15, T16
mul, T7, T16, T17
add, T5, T17, T18
sw, T18, 0, T4
```

```
=====
T41_rascl_test_io.rsc
=====
float a;
float b;

{
   a = 3.7;
   b = a * 2.0;
   print b;
}
```

Equivalent Intermediate code in RIF:

```
=====
T41_rascl_test_io.rso
=====
.segment, 0, 0, .data
.float, 0, 1, a
.float, 0, 1, b


.segment, 0, 0, .text
# Start ASSIGN statement ---
la, a, 0, T0
li, 3.7, 0, FT1
sw, FT1, 0, T0
# Start ASSIGN statement ---
la, b, 0, T1
la, a, 0, T2
lw, T2, 0, FT3
li, 2.0, 0, FT4
fmul, FT3, FT4, FT5
sw, FT5, 0, T1
# Start PRINT statement ---
la, b, 0, T3
lw, T3, 0, FT6
syscall, 4, FT6, 0
```

=====
T62_rascl_test_all_features3.rsc
=====

```
int b, c;
int d;
float e;
float f;

{
  b = 1;
  c = 10;
  e = 5.0;
  f = e * c;
  if (c > b)
  {
    d = 5;
    c = c + -b;
  }
  else
  {
    while (b < 5) {
        c = -d * b;
            b = b + 1;
    };
  };
  print c;
}
```

=====
T62_rascl_test_all_features3.rso
=====

```
.segment, 0, 0, .data
.float, 0, 1, e
.float, 0, 1, f
.int, 0, 1, b
.int, 0, 1, c
.int, 0, 1, d

.segment, 0, 0, .text
# Start ASSIGN statement ---
la, b, 0, T1
li, 1, 0, T2
sw, T2, 0, T1
# Start ASSIGN statement ---
la, c, 0, T4
li, 10, 0, T5
sw, T5, 0, T4
# Start ASSIGN statement ---
la, e, 0, T6
li, 5.0, 0, FT1
sw, FT1, 0, T6
# Start ASSIGN statement ---
la, f, 0, T7
la, e, 0, T8
lw, T8, 0, FT3
la, c, 0, T10
lw, T10, 0, T9
fmul, FT3, FT5, FT4
sw, FT4, 0, T7
# Start if statement ---
la, c, 0, T12
lw, T12, 0, T11
la, b, 0, T14
lw, T14, 0, T13
bgt, T11, T13, L2
j, 0, 0, L1
# Start if statement THEN part ---
.label, 0, 0, L2
# Start ASSIGN statement ---
la, d, 0, T16
li, 5, 0, T17
sw, T17, 0, T16
# Start ASSIGN statement ---
la, c, 0, T19
```

```
la, c, 0, T21
lw, T21, 0, T20
la, b, 0, T23
lw, T23, 0, T22
li, 0, 0, T24
sub, T24, T22, T25
add, T20, T25, T26
sw, T26, 0, T19
# Start if statement ELSE part ---
.label, 0, 0, L1
# Start WHILE statement ---
.label, 0, 0, L5
la, b, 0, T28
lw, T28, 0, T27
li, 5, 0, T29
blt, T27, T29, L4
j, 0, 0, L3
.label, 0, 0, L4
# Start ASSIGN statement ---
la, c, 0, T31
la, d, 0, T33
lw, T33, 0, T32
li, 0, 0, T34
sub, T34, T32, T35
la, b, 0, T37
lw, T37, 0, T36
mul, T35, T36, T38
sw, T38, 0, T31
# Start ASSIGN statement ---
la, b, 0, T40
la, b, 0, T42
lw, T42, 0, T41
li, 1, 0, T43
add, T41, T43, T44
sw, T44, 0, T40
.label, 0, 0, L3
j, 0, 0, L0
.label, 0, 0, L0
# Start PRINT statement ---
la, c, 0, T46
lw, T46, 0, T45
syscall, 2, T45, 0
```