# REGISTER TRANSFER STANDARD

*Manuel E. Benitez*

Department of Computer Science
University of Virginia
Charlottesville, Virginia

Version 1.0

## 1. INTRODUCTION

This document describes the Register Transfer List (*RTL*) standard. The intent of the standard is to minimize the effort required to retarget a systems that utilize *RTL*s as an intermediate representation to new architectures. In addition, the standard also permits tools developed to manipulate *RTL*s to work on any set of *RTL* regardless of the target architecture of the instructions that the *RTL*s represent. A secondary consideration of the standard is to increase the efficiency of systems that utilize *RTL*s. Some aspects of the standard were motivated by the desire to keep *RTL* files as compact as possible and to decrease the amount of time needed to process *RTL*s.

### 1.1 Conventions

Characters are represented as they would in a string in a *C* program.

Registers sequences, even though they are compressed, are depicted as:

*reg(type, number)*

where *type* is a hexadecimal number from 0 through F and *number* is a decimal number between 0 and 512, inclusively.

Global identifier sequences, even though they are compressed, are depicted as:

*global(number)*

where number is a decimal number between 0 and 4195, inclusively.

Local identifier sequences, even though they are compressed, are depicted as:

*local(number)*

where number is a decimal number between 0 and 4195, inclusively.

## 2. CHARACTERS

The *RTL* standard uses the 8-bit ASCII character code set. The motivation for this is that the majority of modern computer systems utilize this character set as their standard for the storage and transmission of text. The set of characters having values between 0 and 31, inclusively, will be referred to as the *control* characters. The set of characters having values between 32 and 127, inclusively, will be referred to as the *printable* characters. The remaining set of characters, having values between 128 and 255, inclusively, are referred to as the *extended* characters.

### 2.1 Control Characters

The control characters, having values between 0 and 31, inclusively, must not appear in an *RTL*.

### 2.2 Extended Characters

The extended characters, having values between 128 and 255, inclusively, are reserved for denoting registers, global identifiers and local identifiers.

### 2.3 Printable Characters

Table 1 describes the designations given to each printable character.

## 3. REGISTERS

Registers in an *RTL* are encoded as two characters. The two most-significant-bits of the first character must be set. The next four bits denote the register type (a total of sixteen different register types are permitted). The last two bits are the most-significant bits of the register number field. The second character must have its most-significant-bit set. The remaining seven bits make up the least-significant seven bits of the register

| Char | Val | Designation | Char | Val | Designation |
|---|---|---|---|---|---|
| ' ' | 32 | Left shift, unsigned | 'P' | 80 | Two character tokens |
| '!' | 33 | Not equal, signed and unsigned | 'Q' | 81 | Two character tokens |
| '"' | 34 | Right shift, unsigned | 'R' | 82 | Two character tokens |
| '#' | 35 | Modulus, unsigned | 'S' | 83 | Two character tokens |
| '$' | 36 | Sign extend | 'T' | 84 | Two character tokens |
| '%' | 37 | Modulus, signed | 'U' | 85 | Two character tokens |
| '&' | 38 | Bitwise AND | 'V' | 86 | Two character tokens |
| '\'' | 39 | Less than or equal to, signed | 'W' | 87 | Two character tokens |
| '(' | 40 | Numeric expressions | 'X' | 88 | Two character tokens |
| ')' | 41 | Numeric expressions | 'Y' | 89 | Two character tokens |
| '*' | 42 | Multiplication, signed | 'Z' | 90 | Two character tokens |
| '+' | 43 | Addition | '[' | 91 | Memory references, macros |
| ',' | 44 | List separator | '\' | 92 | Division, unsigned |
| '-' | 45 | Subtraction, unary minus | ']' | 93 | Memory references, macros |
| '.' | 46 | Floating point constant | '^' | 94 | Bitwise XOR |
| '/' | 47 | Division, signed | '_' | 95 | Unassigned |
| '0' | 48 | Numeric constant | '`' | 96 | Greater than or equal to, signed |
| '1' | 49 | Numeric constant | 'a' | 97 | Unassigned |
| '2' | 50 | Numeric constant | 'b' | 98 | Unassigned |
| '3' | 51 | Numeric constant | 'c' | 99 | Unassigned |
| '4' | 52 | Numeric constant | 'd' | 100 | Auto-decrement |
| '5' | 53 | Numeric constant | 'e' | 101 | Floating-point constant |
| '6' | 54 | Numeric constant | 'f' | 102 | Unassigned |
| '7' | 55 | Numeric constant | 'g' | 103 | Greater than or equal to, unsigned |
| '8' | 56 | Numeric constant | 'h' | 104 | Greater than, unsigned |
| '9' | 57 | Numeric constant | 'i' | 105 | Auto-increment |
| ':' | 58 | Equal, signed and unsigned | 'j' | 106 | Unassigned |
| ';' | 59 | *RTL* separator | 'k' | 107 | Unassigned |
| '<' | 60 | Greater than, signed | 'l' | 108 | Less than, unsigned |
| '=' | 61 | Assignment | 'm' | 109 | Unassigned |
| '>' | 62 | Less than, signed | 'n' | 110 | Unassigned |
| '?' | 63 | Compare, signed | 'o' | 111 | Unassigned |
| '@' | 64 | Multiplication, unsigned | 'p' | 112 | Unassigned |
| 'A' | 65 | Two character tokens | 'q' | 113 | Unassigned |
| 'B' | 66 | Two character tokens | 'r' | 114 | Unassigned |
| 'C' | 67 | Two character tokens | 's' | 115 | Less than or equal to, unsigned |
| 'D' | 68 | Two character tokens | 't' | 116 | Unassigned |
| 'E' | 69 | Two character tokens | 'u' | 117 | Compare, unsigned |
| 'F' | 70 | Two character tokens | 'v' | 118 | Unassigned |
| 'G' | 71 | Two character tokens | 'w' | 119 | Unassigned |
| 'H' | 72 | Two character tokens | 'x' | 120 | Unassigned |
| 'I' | 73 | Two character tokens | 'y' | 121 | Unassigned |
| 'J' | 74 | Two character tokens | 'z' | 122 | Unassigned |
| 'K' | 75 | Two character tokens | '{' | 123 | Left shift, signed |
| 'L' | 76 | Two character tokens, labels | '\|' | 124 | Bitwise OR |
| 'M' | 77 | Two character tokens | '}' | 125 | Right shift, signed |
| 'N' | 78 | Two character tokens | '~' | 126 | Unary negate |
| 'O' | 79 | Two character tokens | '\177' | 127 | Reserved for future use |

**TABLE 1.** Printable character designations

number. The last two bits from the first character concatenated with the last seven bits from the second character yield a nine bit number field which allows a total of 512 different registers for each of the sixteen register types.

For example, the following sequence of characters:

*'\312' '\206'*

yield the following sequences of bits:

*11001010 10000110*

The first two bits of the first character are set to indicate that it and the subsequent character represent a register. The first bit of the second character is set to prevent the character from being mistaken for any other special character. If we strip out these three bits, the remaining sequences are:

*001010 0000110*

Now the sequences are re-arranged so that the first four bits are grouped together to form the type field and the last nine bits are concatenated to comprise the number field:

*0010 100000110*

Decoding the fields then reveals that the character sequence represents a register of type *2* and number *262*. Using our standard notation, this sequence would have been represented as:

*reg(2, 262)*

## 4. GLOBAL IDENTIFIERS

Global identifiers in an *RTL* are encoded as two characters. The most-significant-bit of the first character must be set. The next two bits must be cleared. The remaining five bits are the most-significant-bits of the global identifier number. The second character must have its most-significant-bit set. The remaining seven bits make up the least-significant seven bits of the global identifier number. The last five bits from the first character concatenated with the last seven bits from the second character yield a twelve bit global identifier number field which allows a total of 4096 different global identifiers.

For example, the following sequence of characters:

*'\211' '\366'*

yield the following sequences of bits:

*10001001 11110110*

The first three bits of the first character indicate that it and the subsequent character represent a global identifier. The first bit of the second character is set to prevent the character from being mistaken for any other special character. If we strip out these four bits, the remaining sequences are:

*01001 1110110*

The sequences of bits are concatenated to comprise the number field:

*010011110111*

Decoding the field then reveals that the character sequence represents a global identifier whose number is 1271. Using our standard notation, this sequence would have been represented as:

*global(1271)*

Obviously, the global identifier sequence yields no real information about the nature of the identifier. The mechanism used to describe the global identifier in detail is described in the document entitled *Register Transfer List File Format*.

## 5. LOCAL IDENTIFIERS

Local identifiers in an *RTL* are encoded as two characters. The most-significant-bit of the first character must be set. The next bit must be cleared. The following bit must be set. The remaining five bits are the most-significant-bits of the local identifier number. The second character must have its most-significant-bit set. The remaining seven bits make up the least-significant seven bits of the local identifier number. The last five bits from the first character concatenated with the last seven bits from the second character yield a twelve bit local identifier number field which allows a total of 4096 different local identifiers.

For example, the following sequence of characters:

*'\211' '\366'*

yield the following sequences of bits:

*10101001 11110110*

The first three bits of the first character indicate that it and the subsequent character represent a local identifier. The first bit of the second character is set to prevent the character from being mistaken for any other special character. If we strip out these four bits, the remaining sequences are:

*01001 1110110*

The sequences of bits are concatenated to comprise the number field:

*010011110111*

Decoding the field then reveals that the character sequence represents a local identifier whose number is 1271. Using our standard notation, this sequence would have been represented as:

*local(1271)*

Obviously, the local identifier sequence yields no real information about the nature of the identifier. The mechanism used to describe the local identifier in detail is described in the document entitled *Register Transfer List File Format*.

## 6. COMPONENTS

### 6.1 Numeric Constants

There are two types of numeric constants: integer and floating-point. Integer constants consist of a string of one or more digit characters ('0' through '9'). Integer numeric constants are always base ten (decimal).

Floating point constants have the following form:

*constant { ['.' [constant]] | ['e' ['+'/'-'] constant] }*

### 6.2 Numeric Expressions

Numeric expressions are comprised of constants, registers, identifiers and labels joined together by operators. Parentheses can be used to override operator precedence. No operator precedence is defined in the *RTL standard.*

### 6.3 Assignment

The '=' character is the assignment operator. Each individual register transfer must have one and only one assignment operator separating the source item from the destination item.

### 6.4 Termination

The ';' character is the register transfer terminator. Each individual register transfer must have a terminator. Multiple register transfers can be concatenated to signify that the *RTL* represents an operation that has more than one effect. When an *RTL* has multiple effects, each effect executes simultaneously. Thus, all sources are computed before any assignments are made.

## 6.5 Memory References

Unlike register references, memory references are not compressed, but certain standards do hold for them. A memory reference has the following form:

*type '[' address_expression ']'*

where *type* is any of the upper-case letters between 'B' and 'Z', inclusively. The form of an *address_expression* is described in the following subsection.

## 6.6 Address Expressions

An address expression is similar to a numeric expression. An address expression, however, may need to be differentiated from numeric expression when the target machine provides addressing modes complex enough to make it difficult for a *yacc* grammar to differentiate binary operations form address expressions. An address expression must have the following form:

*'A' '[' expression ']'*

On machines where there is no "load effective address" type instruction to complicate the machine description, this special form for encapsulating address expressions is not needed.