Easy Circuit

-An easy language to design your circuit and figure it out

Language Reference Manual

Lei Zhang |LZ2343
Liming Zhang |LZ2342
Wei Zhang |WZ2254
Xingyue Zhou |XZ2299
Yingnan Li|YL2884

Contents

Introduction

Circuit design plays an important role in many areas. However, some circuit operations, such as computing equivalent resistance, or transforming a circuit, though intuitional in some ways, are still hard to formulary expressed in mathematic or program. Famous electronic design automation (EDA) and simulation program, such as Multisim[1], can help engineers to solve these problems. However, for common students, these developing environments seem to be heavy and expensive. Besides, even for people who are experts in using such simulation software, these applications sometimes still lacks of controllability and accuracy when dragging an element or drawing a wire on the screen. To overcome such weakness, Easy Circuit is a slight, succinct, accurate and easy-to-control programming language, which facilitates people to design, control and analysis circuits. By Easy Circuit, people who as long as possess basic knowledge of programming and circuit, can design a circuit from scratch, and then model it into a component with a function defining its properties. Such modularized components can be reused in a more complicated electronic system.

Easy Circuit has the following features:

Easy to learn: The style is similar to java, and contains basic primitive types and operators, such as int, float, long, double, boolean, +, -, *, /, etc.

Have customized operators and data structures:

Our language contains operators and types especially for circuit design and calculation, such as series connection, parallel
connection, and reciprocal; resistance, capacity, inductance, switch, power, wire, etc.

Functional: User can use our language to model a circuit, i.e., to define a function to describe the properties of certain circuit. In functions, the parameters are the adjustable component in the corresponding circuit.

Lexical conventions

Tokens

There are 5 classes of tokens: identifiers, keywords, whitespace, operators, and other separators.

Comments

The characters /* introduce a comment, which is terminated by the characters */.

The characters // also introduce a comment, which is terminated by the newline character. Comments do not nest. Lines marked as comments are discarded by the compiler.

Whitespace

Blanks, tabs, and newlines, collectively referred to as “white space” are ignored except to separate tokens.

Identifiers

An identifier is a sequence of letters and digits, the first character must be alphabetic. The underscore “_ ” counts as alphabetic. Upper and lower case letters are considered different.

Keywords

The following identifiers are reserved as keywords and may not be used otherwise. In this manual, keywords are bolded.

- if
- else
- for
- while
- return
- int
- float
- new

- switch
- case
- res
- bit
- function

Operators

An operator is a token that specifies an operation on at least one operand. The operand may be an expression or a constant.
Arithmetic operators: +,-, *, /, %, ~
Comparison operators: <, >, <=, >=, !=, ==
Bitwise operators: !, ^, !^, &, !&, |
Assignment operator:=
Circuit operator: # $

Expression and Operators

Easy Circuit language is described using expressions which are made up of one or more operators and operands.
An operand can be either a single res or a res equal to a set of res. All expressions will return a res or an equal res that is then be assigned to a resistance or a set of resistance. This section will detail operators ordered from the most basic blocks to complex operations.

Assignment

The most basic assignment is a single resistance or a single bit value. The value of resistance can be a decimal value or a binary value.

(a)

res resistance_name = new **res**(resistance_value);
res r0 = new **res**(); //assigning an empty value
res r1 = new **res**(6); //assigning a decimal value

(b)

bit bit_name = **bit**_valueb;

bit b1 = 001101b; //assigning a binary value

bit b2= 110010b;

Series Connection, Parallel Connection

To place two or more resistance in series or in parallel connection, we use the concatenation syntax and parenthesis to represent the highest precedence.

(a) Series:

res r_series = r1#r2;

(b) Parallel:

res r_parallel = r1$r2;

(c) Combination of series and parallel connection in left-right order.

res rc = r1#r2$r3#r4;

// result in: res rc = r1#r2; rc = rc$r3; rc = rc#r4;

(d) Parenthesis

Parenthesis has the highest precedence.

(r1#r2)$r3; //result in r1#r2$r3

r1#(r2$r3); //result in r2$r3#r1

Arithmetic

Arithmetic operators are shorthand for common equal operations. Most of they operate on the value of two resistances or their equal resistances. An exception is that Reciprocal operation is on the value of one resistance or its equivalent resistance. All operations are done in two's complement and they will return the value of the new resistance.

res r1 = new **res**(1);
res r2 = new **res**(2);
res r3 = new **res**();
res r4 = new **res**(1.0);
res r5 = new **res**(2.0);

Operator	Example	Result	Note
Plus	r3.value=r1.value+r2.value	r3.value = 3	
Minus	r3.value=r2.value–r1.value	r3.value = 1	r.value should be more than or equal to zero.
Multiplication	r3.value=r1.value*r2.value	r3.value = 2	
Division	r3.value=r1.value/r2.value r3.value=r4.value/r5.value	r3.value = 0 r3.value=0.5	The data type of r.value depends on the data type of its divisor and dividend.
Reciprocal	r3.value = ~r5.value r3.value = ~r2.value	r3.value=0.5 r3.value = 0	It returns the reciprocal value. The data type depends on the original data type.

Binary

Bit operators represents the basic AND, OR and NOT operation.

res a = new **res**(01b);
res b = new **res**(11b);

Operator	Example	Result
NOT	!a	10b
AND	a&b	01b
OR	a|b	11b
NAND	a!&b	00b
NOR	a!|b	10b
XOR	a^b	
XNOR	a!^b	

Comparison

Comparison operators will compare the values of two resistance which do not to be equally sized and will return a Boolean type value.

res r1 = new **res**(1);

res r2 = new **res**(2);

Operator	Example	Result
Less than	r1.value < r2.value	True
Less than or equal to	r1.value <= r2.value	True
Greater than	r1.value > r2.value	False
Greater than or equal to	r1.value >= r2.value	False
Equal to	r1.value == r2.value	False
Not equal to	r1.value != r2.value	True

Precedence and associativity

Operators, in order of decreasing precedence	Associativity
++ -- (postfix, for-loops only)	Left to right
! + - & !& ^ !^ | !|(unary) ~(reciprocal) ++ -- (prefix, for-loops only)	Right to left
* /	Left to right
+ -(binary)	Left to right
< <= > >=	Left to right
== !=	Left to right
& !& (binary)	Left to right
^ !^ (binary)	Left to right
| !| (binary)	Left to right
# $ (binary)	Left to right
=	Right to left

Declaration

There is one special type of declarations in Easy Circuit.

Identifiers

Identifiers are user-friendly names, much like variable names in most programming languages. They must start with upper-or lower-case letter followed by any sequence of letters, numbers, and underscores.

No other characters are allowed in identifier names.

Resistance Declaration

Resistance are declared using the keyword **res:**

res resistance_name = new **res**(resistance_value); //single Resistance

In addition, the declared **res** can also take values immediately during the declaration. Following the identifier a space, an equals sign, another space:
res R3 = R2;

Function Declaration

Easy Circuit language support user-defined and system-defined functions. Normally user of Easy Circuit can use functions to model a circuit component with specific properties (e.g., to compute equivalent resistance of such circuit component) and to reuse it in a more complicated electronic system.
Like C, a function in Easy Circuit contains zero or more statements to be executed when it is called, can be passed zero or more arguments, and can return a value.

Function types

A function's type means the type of the value that is returned after executing the function. The *type* can be any data type in Easy Circuit Language. If the function returns no value, the type of function is void.

Function definitions

In Easy Circuit Language, a function has the following syntax:
function <*function-type*> <function-name> (parameter1-type parameter1-name, parameter2-type parameter2-name,……)

{
statement 1;
statement 2;
…
return statement;
}

In the function definition, the function uses the input parameters to execute the statements and return the value of the statement, with the type defined in *<function-type>*

For example, the following function ***equi_res4*** can compute equivalent resistance of 4 parallel connected resistors.

function *res* **equi_res4**(res r1, res r2, res r3, res r4)
{
res res_equi= r1$r2$r3$r4;
return res_equi;
}

Function Calls

A *function call* is a primary expression. Just like in C or Java, in Easy Circuit Language, an expression used to invoke a function has the following format:

<function-name> (parameter1-name, parameter2-name,……);

(parameter1-name, parameter2-name,……) contains a comma-separated list of expressions that are the arguments to the function. The following is an example of a call to the function ***equi_res4*** defined in the previous part.

{

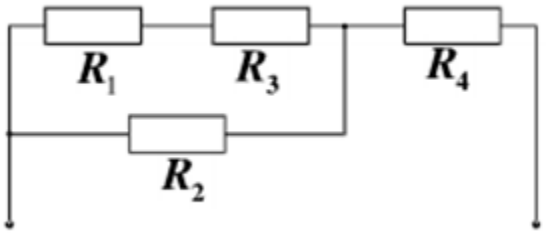
res r1=new **res**(2);
res r2=new **res**(2);
res r3=new **res**(2);
res r4=new **res**(2);
res r_total=new **res**();
r_total =**equi_res4**(r1,r2,r3,r4);
}
After execute the code above the r_total will be assigned a res type variant with value of 0.5.

Function Scope

In Easy Circuit Language, variant declared within one function is valid only in that function. The functions that have been defined will be accessible in the whole file.

Example

In the following code, a function called ***get_equal_res*** was defined to module the circuit component in the figure.



function res **get_equal_res** (res r1, res r2, res r3, res r4)
{
res res_total=new **res**();
res t_total.value=(((r1#r3)$r2)#r4).value

return res_total;
}
// Define the function to calculate the equivalent resistance in circuit module showed in the picture.

res r1=new **res**(3); // Assign the values of R1, R2, R3 and R4
res r2=new **res**(6);
res r3=new **res**(3);
res r4=new **res**(3);

res rTotal= **get_equal_res**(r1, r2, r3, r4);
// Get the equivalent resistance.

Statements

A semicolon is necessary after a statement in Easy Circuit. Because whitespace has no effect, the semicolon is used to signal the end of a statement.
res rTotal=13;

Conditional Statement

Conditional statements work just as they do in C with if and else.

```
If (r1.value < r2.value){
    res r3= r1;
    }
Else {
    res r4= r2;
    }
```

Case/Switch Structures

Case structures use the case keyword and work similar to the switch statement in C.

```
switch ( a ) {
        case b:
            // Code
            break;
        case c:
            // Code
            break;
        default:
            // Code
            break;
}
```

For-loops

For loops in Easy Circuit is similar to the for loops in C

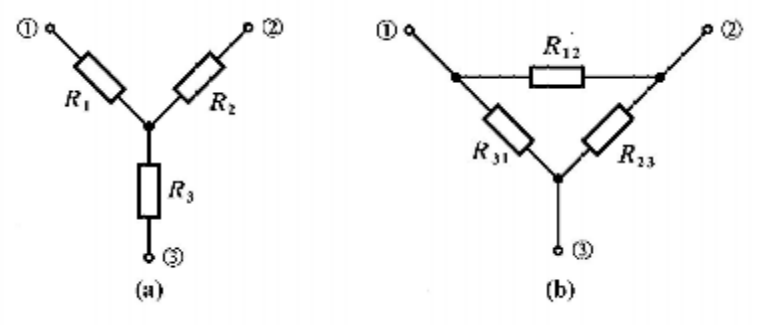
```
For(i=0; i<18; i++){
    …….;
    }
```

Standard Library

The standard library of Easy Circuit Language defines several basic and frequently used circuit components which the user can invoke easily without tedious coding of thus components.

Here are some examples:

As the figure shows as follow, y configuration and delta configuration are commonly used in circuit calculation. They can be transformed into each are by the formulas show as following.



y configuration(left) and delta configuration(right)

$$\begin{cases} R_{12} = \dfrac{R_1R_2 + R_2R_3 + R_3R_1}{R_3} \\ R_{23} = \dfrac{R_1R_2 + R_2R_3 + R_3R_1}{R_1} \\ R_{31} = \dfrac{R_1R_2 + R_2R_3 + R_3R_1}{R_2} \end{cases} \qquad \begin{cases} R_1 = \dfrac{R_{12}R_{31}}{R_{12} + R_{23} + R_{31}} \\ R_2 = \dfrac{R_{12}R_{23}}{R_{12} + R_{23} + R_{31}} \\ R_3 = \dfrac{R_{23}R_{31}}{R_{12} + R_{23} + R_{31}} \end{cases}$$

Formula of transformation between y configuration and delta configuration

YToDelta is a pre-defined module which can transform resistors in a y configuration to a delta configuration, and **DeltaToY** can transform resistors in a delta configuration to a y configuration,

YToDelta{
int YR1;
int YR2;
int YR3;
res DR12=new res(YR1*YR2+YR2*YR3+YR3*YR1)/YR3;
res DR23=new res(YR1*YR2+YR2*YR3+YR3*YR1)/YR1;
res DR31=new res(YR1*YR2+YR2*YR3+YR3*YR1)/YR2;
}

DeltaToY{

int DR12;

int DR23;

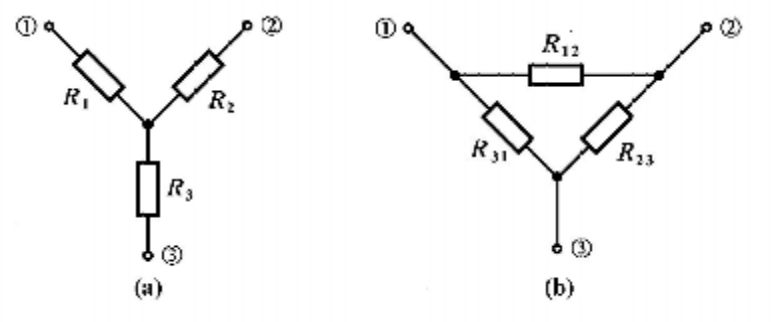
int DR31;

res YR1=new res(DR12*DR31)/(DR12+DR23+DR31);

res YR2=new res(DR12*DR23)/(DR12+DR23+DR31);

res YR3=new res(DR23*DR31)/(DR12+DR23+DR31);

}



Function **bit 74hc138** (bit input)

{

If input. value[1-3] = 100b return “input error”;
If input = 100000b return new bit (01111111b);
If input = 100001b return new bit (10111111b);
If input = 100010b return new bit (11011111b);
If input = 100011b return new bit (11101111b);
If input = 100100b return new bit (11110111b);
If input = 100101b return new bit (11111011b);
If input = 100110b return new bit (11111101b);
If input = 100111b return new bit (11111110b);

Else return new bit (11111111b)

}

Function **bit 74hc338** (bit input)

{

If input. value[3] = 1b return new bit (11111111b);
If input. value[1-3] = 011b return new bit (11111111b);

```
        If input. value[2] = 1b return new bit (11111111b);
        If input = 100000b return new bit (01111111b);
        If input = 100001b return new bit (10111111b);
        If input = 100010b return new bit (11011111b);
        If input = 100011b return new bit (11101111b);
        If input = 100100b return new bit (11110111b);
        If input = 100101b return new bit (11111011b);
        If input = 100110b return new bit (11111101b);
        If input = 100111b return new bit (11111110b);
}
```