GCParser Frontend Documentation

Author: Billy Melicher – wrm2ja@virginia.edu

Last Updated: 2/27/07

Table of Contents

GCParser High Level Language

This document describes the high level language for the GCParser tool. `Monospace text` denotes text that could be included in a circuit file. **`Bold monospace`** text denotes place holder text that is further described.

Purpose

The GCParser high level language specifies a circuit in a high level language. The compiler tool uses this to create a description of the circuit in the GCParser intermediate level language. The tool allows the user to create large efficient circuits at a high level.

Getting the Code

The code can be found on github here:
https://github.com/wrm2ja/frontend
You will also need the GCParser intermediate language tools on github here:
https://github.com/wrm2ja/GCParser
Building requires the ant tool and the javacc tool:
http://ant.apache.org/
http://javacc.java.net/

To build the project use the following command:

```
ant archive
```

The `tests` directory contains example code for specific applications.

Using the Compiler

The `parsertest.sh` file runs the high level compiler. This tool will generate an intermediate language file which is used by the GCParser intermediate language mentioned above. This tool can also create files to specify inputs to the circuit.

parsertest.sh [file] [flags...]

file	path to the high level language file
-c	flag to signal not to use common subexpression optimizations to save memory
-d <d>	The debug level of the compiler. A higher number prints out more
-i <arg>	Will generate the input files for the circuit file. If arg is “random” then inputs will be randomly generated

General Syntax

Comments are in c-style and can be on a single line or span multiple lines.

```
/* comment */ or // comment
```

White space is ignored by the parser.
All statements end in a semicolon.

Variables

Variables are declared with the `var` keyword, and must be initialized on creation. Variables can be declared in function bodies, and in global scope.

Format:
```
var <variable name> = <initial value>;
```

`<variable name>:` the name of the variable to declare. Variable names can have alphanumeric characters and underscores.
`<initial value>:` what this variable will be initialized to. Can be any expression that returns a value.

Variables are scoped, and variables that are declared inside of a function, if statement, or loop, cannot be referenced outside of their scope.

Variable Assignment

Variables are assigned using the following syntax.

Format:
```
<variable name> = <expression or variable>;
```

Variables cannot change types after declaration, so assignment must preserve the type of the variable.

Arrays

Array access format:
```
<array name>[ <index expression> ]
```

Array assignment format:
```
<array name>[ <index expression> ] = <expression or variable>;
```

Input Variables

Input variables can be declared in the global scope using the definput keyword.

Format:
```
definput party <party> <variable name>: <type information>
```

`<party>:` 1 or 2 depending on the party of the variable
`<variable name>:` a valid name of the variable to be declared
`<type information>:` information about the type of the variable

int format: `int magnitude = <mag> ( signed )?`
```
        or
int bits = <bit_size> ( signed )?
```
`<mag>:` the greatest magnitude that the variable will contain at input.

`<bit_size>:` the number of bits this variable will have
`<signed>:` if the signed token is present the integer is a signed integer.

array format: `array[ <array size> ] <type information>`

`<array size>:` the size of the array
`<type information>:` the type information for all elements of the array

boolean format: `bool`

Output Variables

Variables can be declared as output variables using the `defoutput` keyword.

Format:
```
defoutput <variable name>;
```

`<variable name>:` the name of a defined variable.

Declaring an output variable must happen in the global scope, any time after the variable has been declared. The circuit created will output the value of the variable at the end of all evaluation.

Loops

Loops can be created using the loop keyword.

Format:
```
loop <loop variable name> from <start value> to <end value>:
     <loop body>
     …
end
```

`<loop variable name>:` the name of the index variable. An index variable will be created and bound to this value for the scope of the loop. Assignments to this loop variable will result in unspecified behavior.
`<start value>:` the starting value of the loop variable. This must be a value that is known at compile time.
`<end value>:` the ending value of the loop variable. This must be a value that is known at compile time.
`<loop body>:` the body of the loop will be executed `<start value>-<end value>` times.

If Statements

If statements are defined using the if keyword.

Format:
```
if <boolean expression>:
     <if body>
     …
end
```

or

```
if <boolean expression>:
     <if body>
     …
else:
     <else body>
     …
end
```

Functions

There are predefined functions and user defined functions.

Function call format:
`<function name>( <arg>, … )`

Some functions return values, and others do not.

User Defined Functions

Functions are defined using the `defun` keyword.

Format:

```
defun <function name>( <type of arg> <arg name>, … ):
     <function body>
     …
end
```

`<function name>:` the name of the function to be defined. Function names can contain numbers, underscores.
`<type of arg>:` the type of this argument. Can be int, Array or bool.
`<arg name>:` the name of this argument in the function body.
`<function body>:` the body of the function. Return values are specified using the `return` keyword.
`return <expression or variable>;`

Functions can reference variables that have been defined in the global scope before the function definition. User functions can be recursive, but the number of recursive calls must be predictable at compile time.

Arguments to functions will be passed by reference, and not by value so at the end of the following code:

```
var temp = 1;
defun test( int r ):
  r = 0;
end
```

the variable `temp` will have the value 0.

Predefined Functions

The following functions are predefined. The language also supports infix notation for arithmetic functions, however the order of operations is not explicitly defined.

Addition and Subtraction:

`+( int arg1, int arg2 )`
returns the integer sum of `arg1` and `arg2.`

`-( int arg1, int arg2 )`
returns the difference between `arg1` and `arg2.`

Equals:

`==( int arg1, int arg2 )`
or
`==( bool arg1, bool arg2 )`

returns a boolean which is true if `arg1` is equal to `arg2.`

Less than and greater than:

```
<( int arg1, int arg2 )
or
>( int arg1, int arg2 )
```

returns a boolean which is true if `arg1` is less than or greater than `arg2` depending on the operation.

Negate:

`-( int arg )`
returns an integer equal to the negation of `arg`.

Length:

`length( Array arr )`
returns the constant integer representing the length of `arr.`

Bit Width:

`bit_width( var v )`
returns the number of bits variable v requires at this point. This function can be applied to variables of any type.

Boolean functions:

`or( bool arg1, bool arg2 )`
returns the boolean or of `arg1` and `arg2`.
`and( bool arg1, bool arg2 )`
returns the boolean and of `arg1` and `arg2`.