Garbled Circuits Parser Tool Documentation

Author: Billy Melicher
wrm2ja@virginia.edu

Last Update: 2/27/13

Table of Contents

GCParser

This document describes the circuit file format and other tools to be used with the Garbled Circuit Parser 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 Garbled Circuit Parser specifies a file format to describe Garbled Circuit calculations. This format was intended to be an intermediate language that is automatically generated by an outside utility. As such, the format is very inflexible and no control structures exist. However, this will allow a user to execute and define a Garbled Circuit calculation using the java Garbled Circuit Framework, without writing large amounts of custom java code.

Build

The code associated with this document can be found on github at:
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 Garbled Circuit Parser use the following command.

```
ant archive
```

Tutorial

This tutorial will walk through how to create a short circuit file, and use the Parser tools to execute it. The tutorial will present a mock circuit to calculate whether two 32 bit numbers contain a common byte.

Writing a Circuit File

The parser uses a circuit file to define the garbled circuit calculations. The circuit that we are going to create will take two 32 bit numbers from different parties. The following lines declare these input variables.

```
.input a 1 32
.input b 2 32
```

This defines two inputs named `a`, and `b`. `a` is from party 1 meaning the client, and `b` is from party 2 meaning the server. Both inputs are 32 bits long.

The output of this circuit is a 1 bit quantity which is 0 if the two numbers do not contain a common byte and 1 if they do.

```
.output samebyte
```

This line labels the samebyte variable as an output of this circuit. It will be defined later in the circuit file.

Next we have to divide each 32 bit quantity into 1 byte chunks. The select operator will be used here. It is defined in greater detail in the Operations section, but essentially, it selects a range of bits from a variable and creates a new variable. The following section will declare variables for each of the four bytes in each input.

```
a1 select a 0 8
a2 select a 8 16
a3 select a 16 24
a4 select a 24 32

b1 select b 0 8
b2 select b 8 16
b3 select b 16 24
b4 select b 24 32
```

This give us the bytes for each input. This section declares variables `a1, a2, a3, a4, b1, b2, b3,` and `b4` for use later in the circuit.

Now we have to compare them using the equal to operator.

```
eq1 equ a1 b1
eq2 equ a2 b2
eq3 equ a3 b3
eq4 equ a4 b4
```

To see if any of these bytes are equal, we will then or these values together.

```
eq12 or eq1 eq2
eq34 or eq3 eq4
samebyte or eq12 eq34
```

So now, `samebyte` has the value of 1 if any of the bytes in the inputs are the equal. The circuit file format will be discussed in more detail later in this document.

The complete file is recreated below. Save it in a file named samebyte_tutorial.cir.

```
.input a 1 32
.input b 2 32
a1 select a 0 8
a2 select a 8 16
a3 select a 16 24
a4 select a 24 32
b1 select b 0 8
b2 select b 8 16
b3 select b 16 24
b4 select b 24 32
eq1 equ a1 b1
eq2 equ a2 b2
eq3 equ a3 b3
eq4 equ a4 b4
eq12 or eq1 eq2
eq34 or eq3 eq4
samebyte or eq12 eq34
.output samebyte
```

Checking a Circuit File

First to run the file, we will check if it has any errors. We will use the `testfiles` utility. This is a bash file that is located in the root of the source code repository. We will test the circuit file that we created in the previous section, samebyte_tutorial.cir. Navigate to the root of the repository and execute the following command.

```
testfiles samebyte_tutorial.cir
```

Expected output:

```
samebyte_tutorial.cir: ok
```

If the circuit file had mistakes, or was not able to be executed, error messages with line references would be shown.

testfiles usage:

```
testfiles [-h –help] [-d --debug] [-l --debug-local] FILE1 …

-d --debug
     Displays debugging information about the circuit file if it can
be constructed. Prints the computation tree before any local
computation blocks are taken into account.

-l --debug-local
     Displays debugging information about the circuit file if it can
be constructed. Prints the computation tree after local computation
blocks are taken into account.
```

Running a Circuit File

To run our circuit file with random inputs:

```
runparserrandom samebyte_tutorial.cir
```

This will write output to the file results/siserverout, and results/siclientout, and print the time that the client and server took to run.

To run a circuit file with given inputs:

```
runtestgcparser server_input_file client_input_file
```

The format of the input file, is a single input name and then the value on each line. For example:

```
a 13
b 42
```

Instead of using the bash scripts, you can call the client and server independently.

Client:

```
java -ea -cp
dist/GCParser.jar:extlibs/commons-io-1.4.jar:extlibs/jargs.jar
Test.TestGCParserClient -f circuit_file --server host -r num -p
private_file

-f circuit_file
     the path to the circuit file description

--server host
     the address of the garbled circuits server

-r num
     the number of iterations to run the circuit

-p --private-file-name
     the path to the private input file
```

Server:

```
     java -ea -cp
dist/GCParser.jar:extlibs/commons-io-1.4.jar:extlibs/jargs.jar
Test.TestGCParserServer -f circuit_file
-w wire_length -p private_file

-f circuit_file
     the path to the circuit file description

-w wire_length
```

```
     the bit length of the garbled wire labels.

-p --private-file-name
     the path to the private input file
```

File Format

A circuit file describes a series of calculations to be performed using the Garbled Circuits Protocol. A circuit file has three sections, the input section, the output section, and the calculation section, in that order.

Mock Input Section:
```
.input a1 1 32
.input b2 2 32
```

Mock Output Section:
```
.output sum signed
```

Mock Calculation Section:
```
sum add a1 b2
```

Variables

Variables have a value and a bit width. Variables can be either named, or literal constants. Variable names must start with a letter, but can contain both letters and numbers. Literal constants must specify the value of the constant and also the bit length of the constant. Literal constants can be used in any place that a variable can. Some operations require literal constants as operands. These are documented in the operations section.

Literal Constant format:

`value:bitwidth`

`value` – the value of the constant
`bitwidth` – the number of bits to use for this constant
Example:

```
45:32
```

This example declares a constant of value 45, which is 32 bits wide.

Input Section

The input section specifies the input variables that are used in the circuit file.
Format:

`.input` **`variablename party bitwidth`**

`variablename` – name of the input variable. Variable names must start with a letter, but can also contain numbers.
`party` – the party that this input comes from. A value of 1 means that it comes from the client, 2 comes from the server.

`bitwidth` – the integer number of bits for this input.

Output Section

The output section specifies the output variable(s) that this circuit calculates.
Format:

```
.output variablename
```

`variablename` – the name of the output variable. This variable is defined somewhere else in the circuit file, either as an input, or as the result of a previous calculation. The variable must be defined before the output statement.

The output declaration can optionally contain a hint for how an application should interpret the output, signed or unsigned. This is not strictly necessary, but is used by some applications to display the values correctly.

Format:

```
.output variablename signed
.output variablename unsigned
```

Calculation Section

Every calculation, local computation mark, or include file operation must be on its own line.

Calculation

A calculation defines a variable in the circuit as the evaluation of an operation on a number of operands.
Format:

```
newvariable operation operand1 operand2 …
```

`newvariable` – the name of the variable that is defined. This variable name cannot be previously defined anywhere in the circuit file, and must follow the rules for variable names.
`operation` – the name of an operation to perform.

Operands are listed after the operation, and are delimited by spaces. Operands may be the name of a previously declared variable, or a literal value.

Local Computation

Some calculations can be done completely local to one party.
Format:

```
.startparty partynum
```

`partynum` – the value of the party who is supposed to calculate a value. As in the input section, 1 signifies the client will compute this value and 2 signifies the server.

And to end a local computation block
Format:

```
.endparty partynum
```

Local computations will be done before other computations and their output will be treated as input variables to the circuit. So, a local computation cannot depend on any computation that is not done locally, or an input that is not known to the computing party.

The following example illustrates a file with local computation blocks and one without and discusses the differences.

Example file without local computation:

```
.input a1 1 32
.input a2 1 32
.input b1 2 32
.input b2 2 32

a3 add a1 a2
b3 add b1 b2
minsum min a3 b3

.output minsum
```

Example file with local computation:

```
.input a1 1 32
.input a2 1 32
.input b1 2 32
.input b2 2 32

.startparty 1
a3 add a1 a2
.endparty 1

.startparty 2
b3 add b1 b2
.endparty 2

minsum min a3 b3

.output minsum
```

Local computation blocks were added to perform the addition of the two inputs locally before computing the minimum of these sums. This circuit, will now treat the variables a3 and b3 as inputs, but will not transfer variables a1, a2, b1,and b2, because they are not used in the non-local computation. Notice that each local computation block only depends on local values. For example, party 1 is computing a value that is dependent on only values known to party 1.

Including Circuit Files

One circuit can include the computations from another circuit file. The included file is given variables to be mapped to the file's inputs, and variables to declare based on the included file's outputs.
Format (all on one line):

```
.include<path_to_file> .output(decvar1:incvar1, decvar2:incvar2,...) .input(incinput1: var1, incinput2: var2, ... )
```

`path_to_file` – the path to the included circuit file. This path can be relative or absolute. Relative paths are relative to the current circuit file.

`decvar` – variables that are to be declared in the current circuit file.
`incvar` – output variables in the included circuit file. This variable is declared in the included circuit file, and it gets renamed to its corresponding decvar in the including file.
`incinput` – the name of an input variable in the included file. The party markings in the included file are ignored, but the bit markings are not.
`var` – variables in the including file that are fed to their corresponding input variables.

The following example illustrates including a file. The example calculates whether the upper and lower 16 bits of two 32 bit numbers are equal.

File uplow.cir:

```
.input a 1 32
.input b 2 32

upa select a 16 32
upb select b 16 32
lowa trunc a 16
lowb trunc b 16

up xor upa upb
low xor lowa lowb
.output up
.output low
```

File include_test.cir:

```
.input client 1 32
.input server 2 32

.include<uplow.cir> .output(up:ud,low:ld)
     .input(a:client,b:server)

upisdiff or ud
lowisdiff or ld

.output upisdiff
.output lowisdiff
```

The file include_test.cir includes the file uplow.cir. These files must be in the same directory. The circuit file executes the calculations in uplow.cir, using client for variable a, and server for variable b. Then, the output ud is declared, with the value up from uplow.cir, and the output ld is declared with the value of low from uplow.cir. An include statement can be declared inside a local computation block, and normal local computation rules apply.

Operations

This section lists allowed operations and their correct usage.

Add

usage: `add operand1 operand2`
description:

Adds operand1 and operand2. Operand1 and operand2 must have the same bit width. The result has the same bit width as operand1 and operand2.

Concatenation

usage: `concat operand1 operand2 operand3 …`
description:

Concatenates the bits of its operands. Can take a variable number of operands. Most significant bits correspond to the first operands.
Example:

```
ans concat 3:4 255:8 0:8
```

This example results in ans holding the 24 bit value 0x3 FF 00

Equal to and Not equal to

usage:
```
equ operand1 operand2
nequ operand1 operand2
```
description:

Compares the bits of operand 1 and operand 2. Operand1 and operand2 must have the same bit width. The result is a 1 bit quantity which is 1 if the operands pass the comparison or 0 if they do not.

Greater than or equal to signed and Less than or equal to signed

usage:
```
gtes operand1 operand2
ltes operand1 operand2
```
description:

Interprets operands as signed integers and performs a greater/less than or equal to operation. Operand1 and operand2 must have the same bit width. The result is a 1 bit quantity which is 1 if operand1 is greater/less than or equal to operand2 and 0 otherwise.

Greater than or equal to unsigned and Less than or equal to unsigned

usage:
```
gteu operand1 operand2
lteu operand1 operand2
```
description:

Interprets operands as unsigned integers and performs a greater/less than or equal to operation. Operand1 and operand2 must have the same bit width. The result is a 1 bit quantity which is 1 if operand1 is greater/less than or equal to operand2 and 0 otherwise.

Greater than unsigned and Less than unsigned

usage:
```
gtu operand1 operand2
ltu operand1 operand2
```
description:

Interprets operands as unsigned integers and performs a greater/less than operation. Operand1 and operand2 must have the same bit width. The result is a 1 bit quantity which is 1 if operand1 is greater/less than operand2 and 0 otherwise.

Greater than signed and Less than signed

usage: `gts operand1 operand2`
`lts operand1 operand2`
description:

Interprets operands as signed integers and performs a greater/less than operation. Operand1 and operand2 must have the same bit width. The result is a 1 bit quantity which is 1 if operand1 is greater/less than operand2 and 0 otherwise.

Max signed and Min signed

usage: `maxs operand1 operand2`
`mins operand1 operand2`
description:

Interprets operand1 and operand2 as signed integers and returns the larger value (max) or the smaller value (min). Operand1 and operand2 must have the same bit width. The result value has the same bit width as operand1 and operand2.

Max and Min

usage: `max operand1 operand2`
`min operand1 operand2`
description:

Interprets operand1 and operand2 as unsigned integers and returns the larger value (max) or the smaller value (min). Operand1 and operand2 must have the same bit width. The result value has the same bit width as operand1 and operand2.

Not

usage: `not operand1`
description:

Performs a bitwise not operation on operand1. The output will have the same bit width as operand1.

Negate

usage: `negate operand1`
description:

Performs the two's complement negation of operand1. The output will have the same bit width as operand1.

Bitwise Or and Bitwise Xor and Bitwise And

usage: `or operand1 operand2`
`xor operand1 operand2`
`and operand1 operand2`
description:

Performs a bitwise operation on operand1 and operand2. Operand1 and operand2 must have the same bit width. The result value has the same bit width as operand1 and operand2.

Unary Or and Unary Xor

usage: `or operand1`
`xor operand1`
description:

Performs the inclusive or exclusive “or” operation on all of the bits of operand1 together. The resulting value is a 1 bit quantity.

Select

usage: `select operand1 compileconstant1 compileconstant2`
description:

Selects a range of bits in operand1. Compileconstants 1 and 2 refer to constant values that must be specified as constants in the circuit file. The specified range starts at compileconstant1 inclusive, and ends at compileconstant2 exclusive. The compileconstants do not require an associated bit length like other constants. The bits are zero indexed, so bit 0 refers to the 1st bit.
Example:

a1 holds the 16 bit value 31 or 0b 0000 0000 0001 1111
`ans select a1 2 7`
This example will result in ans having the value of bits 3, 4, 5, 6, and 7 or 0b00111

Subtract

usage: `sub operand1 operand2`
description:

Performs twos complement subtraction of operand1 - operand2. Operand1 and operand2 must have the same bit width. The result value has the same bit width as operand1 and operand2.

Sign Extend ans Zero Extend

usage: `sextend operand1 compileconstant`
`zextend operand1 compileconstant`
description:

Extends operand1 to the number of bits specified by compileconstant. Sextend extends operand1 while preserving the sign and value of operand1. Zextend extends operand1 with zeros. The result will have as many bits as specified by compileconstant. This compileconstant does not require a specified bitwidth.

Truncate

usage: `trunc operand1 compileconstant`
description:

Truncates operand1 to the number of bits specified by compileconstant. This function is equivalent to `select operand1 0 compileconstant`. Compileconstant must be a constant in the circuit file, and does not require a bit length like other constants.