'Bristol Fashion' MPC Circuits

If you are looking for the old 'Bristol Format' circuits please see here. We call this new format 'Bristol Fashion' as it is related to the old format, which was called 'Bristol Format' by some others (as they were originally hosted in Bristol). The new format is tidier than the old format, which also is a justification for the new name of Bristol Fashion.

Our new format is utilized in the SCALE-MAMBA software system to define the circuits for garbling. The new format is designed to be independent of the number of parties, and to capture more the nature of the function we are evaluating.

The `basic' format is defined by a list of gates. Each gate has one or two input wires (INV/NOT/EQ/EQW gates have only one input wire, XOR and AND have two input wires). A gate can have only one output wire. The `extended' format allows addition MAND gates, standing for Multiple AND, these have an arbitrary number of inputs and outputs (see below for details). Each file is of the following format:

- A line defining the number of gates and then the number of wires in the circuit.
- The number of input values niv (e.g. if doing z=a+b mod p we have niv=3, one input each for a, b and p).
  - Then niv numbers defining the number of input wires per input value: ni_1,..,ni_niv
- The number of output values nov (e.g. if doing z=a+b mod p we have nov=1, corresponding to z).
  - Then nov numbers defining the number of output wires per output value: no_1,...,no_nov
- The gates are then given in topological order, so we can evaluate them in sequence.
- Each gate is defined by
  - Number input wires (1 or 2, unless a MAND gate)
  - Number output wires (Always 1, unless a MAND gate)
  - List of input wires
  - List of output wires
  - Gate operation (XOR, AND, INV, EQ, EQW or MAND).

  This is a bit redundant, as the first two entries can be inferred from the last, but we keep this for backwards compatibility reasons
  - So for example

    ```
    2 1 3 4 5 XOR
    ```

    corresponds to

    ```
    w_5 = XOR(w_3,w_4)
    ```

  - We also use

    ```
    1 1 0 3 EQ
    1 1 1 4 EQ
    ```

    to say that wire 3 is assigned the value 0 and wire 4 the value 1
  - And we use

    ```
    1 1 0 4 EQW
    ```

to say wire 4 should equal wire 1

- The MAND gate is a multiple AND gate it takes 2n input wires and produces n output wires. A gate of the form

```
4 2 0 2 1 3 4 5 MAND
```

executes the two MAND operations concurrently...

```
2 1 0 1 4 AND
2 1 2 3 5 AND
```

- We call a circuit without MAND gates a `basic' Bristol Fashion circuit, and when we have MAND gates we say it is an `extended' Bristol Fashion circuit.

Note:

1. The wire numbering is ordered so that the first i_0 wires correspond to the first input value, the next i_1 wires correspond to the second input value and so on.
2. With the last (o_0+...+o_{n-1}) wires corresponding to the outputs of the function, where n=no_1+...+no_nov

Currently we only have a few circuits available, more will be added as time goes by. The *depth* below gives the number of AND/MAND gates in the extended format file. This corresponds to the AND-depth of the circuit. The number of AND gates corresponds to the number of {\em individual} ANDs, i.e. the number of ANDs in the basic format file.

Arithmetic Functions

Function	Basic Circuit File	Extended Circuit File	No. ANDs	No. XORs	No. INVs	Depth
64-bit Adder	adder64.txt	adder64.txt	63	313	0	63
64-bit Subtract	sub64.txt	sub64.txt	63	313	63	63
64-bit Negation	neg64.txt	neg64.txt	62	63	64	62
64x64 -> 64 bit Multiplier	mult64.txt	mult64.txt	4033	9642	0	63
64x64 -> 128 bit Multiplier	mult2_64.txt	mult2_64.txt	8128	19904	0	127
64x64-bit Division (Signed)	divide64.txt	divide64.txt	4664	24817	445	4158
64x64-bit Division (Unsigned)	udivide64.txt	udivide64.txt	4285	12603	64	2205
64-bit Equal to Zero Test	zero_equal.txt	zero_equal.txt	63	0	64	6

Cryptographic Functions

Function	Basic Circuit File	Extended Circuit File	No. ANDs	No. XORs	No. INVs	Depth
AES-128(k,m)	aes_128.txt	aes_128.txt	6400	28176	2087	60
AES-192(k,m)	aes_192.txt	aes_192.txt	7168	32080	2317	72
AES-256(k,m)	aes_256.txt	aes_256.txt	8832	39008	2826	84

Keccak-f	Keccak_f.txt	Keccak_f.txt	38400	115200	38486	24
SHA-256	sha256.txt	sha256.txt	22573	110644	1856	1607
SHA-512	sha512.txt	sha512.txt	57947	286724	4946	3303

Note for AES-128 the wire orders are in the reverse order as used in the examples given in our earlier `Bristol Format', thus bit 0 becomes bit 127 etc, for key, plaintext and message.

For AES we created a design using the Boyar-Peralta S-Boxes, which have 32 AND gates per S-Box.

For SHA-256 and SHA-512 we give a circuit which maps an input buffer and an input chaining state to the next chaining state.

To get the nice gate counts for SHA-256 and SHA-512 we thank Steven Goldfeder who provided advice and help from his work in the paper:

> Zero-Knowledge Contingent Payments Revisited: Attacks and Payments for Services Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, Luca Nizzardo. ACM Conference on Computer and Communications Security (CCS), 2017.

In particular he pointed out the 'correct' method for addition circuits which means an n-bit adder only uses n-1 AND gates. Whilst this addition method was well known to us turned out our VHDL synthesis tools would spot we were doing an adder and then substitute in the usual circuit used in real hardware (which uses more AND gates). Fixing this required some VHDL trickery from Danilo. Finally, the same trick was then factored into the improved 64-bit arithmetic circuits above.

The circuit for *udivide64* was provided by Steve Lu, who generated it using the EMP-Toolkit.

IEEE Floating Point Operations

We also provide some circuits for IEEE floating point operations. These operations deal with the IEEE rounding and NaN etc correctly. An 64-bit IEEE double is represented in the following format; in particular

1. Bit 63 is the sign bit
2. Bits 52-62 are the exponent, with 62 being the msb.
3. Bits 0-51 are the mantissa, with 51 being the msb.

Function	Basic Circuit File	Extended Circuit File	No. ANDs	No. XORs	No. INVs	Depth
add	FP-add.txt	FP-add.txt	5385	8190	2062	235
mul	FP-mul.txt	FP-mul.txt	19626	21947	3326	129
div	FP-div.txt	FP-div.txt	82269	84151	17587	3619
eq	FP-eq.txt	FP-eq.txt	315	65	837	9
lt	FP-lt.txt	FP-lt.txt	381	257	898	67
f2i	FP-f2i.txt	FP-f2i.txt	1467	1625	840	94
i2f	FP-i2f.txt	FP-i2f.txt	2416	3605	1115	206
sqrt	FP-sqrt.txt	FP-sqrt.txt	91504	100120	19900	6507
floor	FP-floor.txt	FP-floor.txt	651	595	367	71
ceil	FP-ceil.txt	FP-ceil.txt	650	597	371	71

Note the function f2i implements the `round nearest-even' convention which is equivalent to the C++ code

```
double xx=....;
unsigned long zz;
zz=(long) nearbyint(xx);
```

With i2f doing the inverse operation, i.e. converting a 64-bit integer into a double.

Most of the circuits were mostly created using Synposis tools from VHDL source. The VHDL can be found in the release of SCALE-MAMBA if you are interested. The current circuits above were produced from the VHDL in SCALE-MAMBA version 1.7. After the associated NetList was produced we post-processed them into the format described above, for use within the SCALE-MAMBA engine.

The floating point circuits were created by using the Berkeley SoftFloat library (version 2), then the output was passed through CBMC-GC to produce Bristol Format files with no OR's. We then manually edited the result to produce Bristol Fashion files.

David Archer
Victor Arribas Abril
Steve Lu
Pieter Maene
Nele Mertens
Danilo Sijacic
Nigel Smart