| galois |

Lab: equivalence of functions written in C, Java, Cryptol

Consider the following problem which can be solved in several different ways:

> Find the first 1 in a given 32 bit number.

Examples:

Given number		Pos. of first 1
Binary	Decimal	
00100111100011001010001101101100	663528300	3
00000000000100000001000000010000	1052688	5
11100100100100000000000000000000	3834642432	21
11100000000000000000000000000000	3758096384	30
00000000000000000000000000000000	0	0

The previous lab considered various solutions to this problem, written in C, and proved equivalence of functions using a linear search, a binary search, and table lookup using deBruijn sequences. This is continued here adding Java and Cryptol implementations. Equivalence is proved for functions written in different languages: that is, a C function of using one method is proved equivalent to a Java function and a Cryptol function using the same or different methods. Similarly for Java and Cryptol implementations. Whereas `llvm` was used in the case of C, and-inverter graphs are relied on to create representations across Java and Cryptol that may be equivalence checked by solvers that specialize in that task. In the following a brief introduction to and-inverter graphs is presented followed by exercises proving equivalence in SAW.

And-Inverter Graphs (AIG)

An AIG is a graphical representation of a Boolean expression where vertices represent ‘and’ gates, edges are directed outward from ‘and’ vertices to other ‘and’ vertices or ‘output’ vertices. Each ‘and’ vertex has at most two in-directed edges. A white or black circle may be placed on an edge with the interpretation that the logic value on one end of the edge is opposite to the logic value on the other end: that is, the circle is an inverter. Every Boolean expression has one or more equivalent AIGs.

Figure 1 shows logic gates expressed as AIGs. The Boolean expressions

$$X = (A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge C) \vee (A \wedge B \wedge C)$$
$$Y = (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$$

have an AIG representation as shown in Figure 2 where edge direction is not explicitly shown but edges are implicitly directed from left to right. Labels `X` and `Y` represent output and carry of a 1-bit adder.

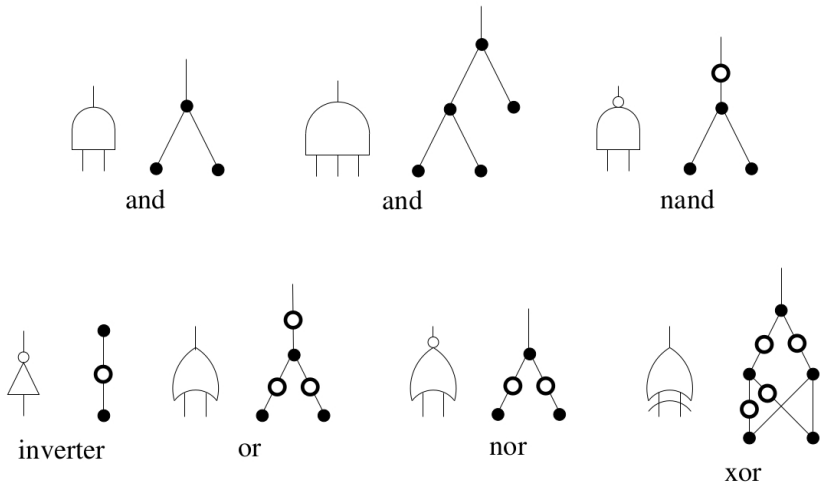

Figure 1: Logic gates as AIG snippets

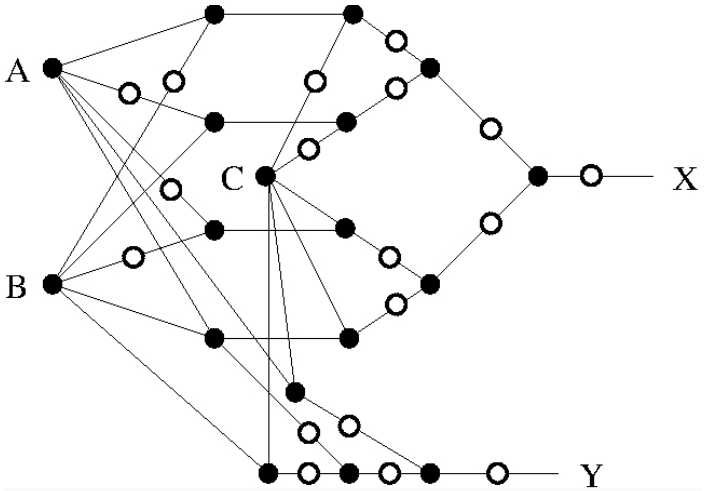

Figure 2: An AIG representing the 1-bit adder specification

Exercise 1:

In the `lab5C` directory open a terminal and find `ffs_ref.c` and `ffs_imp.c`. Run

```
clang-12 -g -O0 -c -emit-llvm ffs_ref.c -o ffs_ref.bc
clang-12 -g -O0 -c -emit-llvm ffs_imp.c -o ffs_imp.bc
```

to create `llvm` bitcode versions of the two C programs: `ffs_ref.bc` and `ffs_imp.bc`. Create a file called `makeaigc.saw` with the following content:

```
c1 <- llvm_load_module "ffs_ref.bc";
c2 <- llvm_load_module "ffs_imp.bc";

c_ffs_ref <- llvm_extract c1 "ffs_ref";
c_ffs_imp <- llvm_extract c2 "ffs_imp";

write_aig "c_ffs_ref.aig" c_ffs_ref;
write_aig "c_ffs_imp.aig" c_ffs_imp;
```

Run `saw` on that file like this:

```
saw makeaigc.saw
```

Observe in the lab5C directory two new files exist: `c_ffs_ref.aig` and `c_ffs_imp.aig`. ■

If you want to see what is in those files download the `aiger` suite from

https://github.com/arminbiere/aiger

and compile according to the README – do not worry about `picosat` or `lingeling`. Put the location of the aiger binaries in your `PATH` or make sure the location of the binaries is already in your `PATH`. Run

```
aigtosmv c_ffs_ref.aig
aigtosmv c_ffs_imp.aig
```

The result for `c_ffs_ref.aig` looks similar to or the same as this, where sections are partially collapsed using . . .

```
MODULE main
VAR
--inputs
i2:boolean;
i4:boolean;
...
i62:boolean;
i64:boolean;
--latches
ASSIGN
DEFINE
--ands
a66:=!i2&1;
a68:=i2&0;
a70:=!a68&!a66;
...
a2846:=!a2818&!a320;
a2848:=!a2846&!a2820;
--outputs
o0:=!a2824;
o1:=!a2824;
...
o30:=!a2844;
o31:=!a2848;
```

The AIG format supports Latches but none are present in this case as the `--latch` section shows. The `--inputs` section shows 32 inputs labeled `i2` to `i64`, ending only in even numbers and listed one per line of file. The `--outputs` section shows 32 outputs labeled `o0` to `o31`, and listed one per line of file. The graph is constructed in the `--ands` section where one ‘and’ vertex appears with its inputs, one per line of file. The ‘and’ vertices are identified by strings beginning with the letter ‘`a`’ and followed by a number, for example `a2846`. A line such as this:

```
a972:=a970&!a568;
```

means that ‘and’ vertex `a972` takes inputs from the output of `a970` and an inverter (`!`) that takes its input from vertex `a568`. A 1 or 0 that is not preceded by a letter is the value 1 or 0, respectively. For example: `a66:=!i2&1;`

The same result can be obtained for Java functions with just slight changes: namely, Java code must be compiled and then `java_load_class` is used instead of `llvm_load_module`. In the case of java, only the class file of interest is loaded. Thus, for example,

```
j1 <- java_load_class "ffs_ref";
```

Also, it is the jvm_extract that is used instead of the llvm_extract, for example like this:

```
java_ffs_ref <- jvm_extract j1 "ffs_ref";
```

But writing the aig file is the same, for example like this:

```
write_aig "java_ffs_ref.aig" java_ffs_ref;
```

Exercise 2:
Create file `makeaigjava.saw` and insert content that will create `.aig` files `java_ffs_ref.aig` and `java_ffs_imp.aig` from class files compiled from `ffs_ref.java` and `ffs_imp.java`. Run saw on `makeaigjava.saw` and check the sizes of all the `.aig` files you have created so far. Also check the number of ‘and’ vertices of each `.aig` file. Also try to check for the number of inverters. Is there a considerable difference in sizes, number of vertices, and number of inverters? To find the number of ‘and’ vertices in a .aig file do this, e.g. on `c_ffs_ref.aig`:

```
cat c_ffs_ref.aig | more
```

which returns

```
aig 1424 32 0 32 1392
2825
2825
2825
...
```

The number on the right, 1392, is the number of ‘and’ vertices ■

Cryptol file `ffs.cry` contains function `ffs_imp` and `ffs_ref`. To load both from one file use this:

```
c1 <- cryptol_load "ffs.cry";
```

Extraction can be done like this:

```
cry_ffs_ref <- cryptol_extract c1 "ffs_ref";
```

Creation of an AIG is due to this:

```
write_aig "cry_ffs_ref.aig" cry_ffs_ref;
```

Exercise 3:
Create file `makeaigcryptol.saw` and insert content that will create `aig` files `cry_ffs_ref.aig` and `cry_ffs_imp.aig` from `ffs.cry`. Run saw on `makeaigcryptol.saw` and check the sizes of all the `aig` files you have created so far. Also check the number of ‘and’ vertices of each `aig` file. It is also of interest to find the number of inverters (`!`) in the

files. Is there a considerable difference in sizes, number of vertices, and number of inverters? ■

Once the AIG files have been created they may be read into SAW where objects that can be used in proofs are loaded. To do this use, for example:

```
cry_ffs_imp <- read_aig "cry_ffs_imp.aig";
```

An example of a theorem that can be proved in this way is this:

```
let thm = {{ \x -> cry_ffs_imp x == cry_ffs_ref x }};
result <- prove z3 thm;
print result;
```

The statement inside the double braces is a Cryptol statement that is a function of one argument, `x`, and returns `True` iff `cry_ffs_imp` is the same for `cry_ffs_ref` for `x`. The `result` statement says prove the function using the `z3` SMT solver. The `print` statement says print the result which will be `Valid` if the theorem is proved or a counterexample if the theorem cannot be proved.

Exercise 4:
Create a file named `ffs_compare_aig.saw` which reads all six AIGs (`imp`, `ref` for `C`, `Java`, `Cryptol`) and proves the following theorems using `z3`:

```
java_ffs_ref ≡ java_ffs_imp
c_ffs_ref ≡ c_ffs_imp
c_ffs_ref ≡ java_ffs_imp
java_ffs_ref ≡ c_ffs_imp
java_ffs_imp ≡ cryptol_ffs_ref
java_ffs_imp ≡ cryptol_ffs_imp
cryptol_ffs_imp ≡ cryptol_ffs_ref
c_ffs_imp ≡ cryptol_ffs_ref
```
■