Lab: prove or disprove equivalence of C functions

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
Binary	**Decimal**	**first 1**
00100111100011001010001101101100	663528300	3
00000000000100000001000000010000	1052688	5
11100100100100000000000000000000	3834642432	21
11100000000000000000000000000000	3758096384	30
00000000000000000000000000000000	0	0

So, 0 is used to indicate no 1s are present in the given number. Below is simple C code that solves this problem by iteratively bitwise anding the given number (here referred to as `word`) with a 1 that is placed in the 0^{th} bit position, then the 1^{st} bit position, then the 2^{nd} and so on until the result of the anding is not 0. The 1 is placed by shifting by an amount equal to the value of variable `i` (`1 << i++`). When the anding is not 0 the value of `i` is returned. If the anding is always 0 then 0 is returned. Observe that `i` is initialized to 0 and if there is a 1 in the 0^{th} bit of word then `i` is incremented by 1 and its value is returned as 1. Observe that when `i` is output its value is going to be one greater than the actual bit position under the usual interpretation that the LSB is bit 0.

```
uint32_t ffs_ref(uint32_t word) {
   int i = 0;
   int cnt = 0;
   if (!word) return 0;
   for (cnt = 0; cnt < 32; cnt++)
      if (((1 << i++) & word) != 0) return i;
   return 0;
}

int main (int argc, char **argv) {
   uint32_t n = atol(argv[1]);
   uint32_t m = ffs_ref(n);
   if (m == 0) printf("ffs_ref: no 1s in this number\n");
   else printf("ffs_ref: first 1 at %d\n", m);
}
```

Place the code in a file called `ffs_ref.c`, add the include files (`stdio.h`, `stdlib.h`), compile and test using numbers in the above table.

```
ffs_ref 663528300
ffs_ref: first 1 at 3
ffs_ref 1052688
ffs_ref: first 1 at 5
ffs_ref 3834642432
ffs_ref: first 1 at 21
ffs_ref 3758096384
ffs_ref: first 1 at 30
ffs_ref 0
ffs_ref: no 1s in this number
```

```
clang-12 -g -O0 -c –emit-llvm add.c -o add.bc
```

For this to work, `clang-12` must be installed. Switch `-O0` means "no optimization": this level compiles the fastest and generates the most debuggable code. Switch `-O1` instead means generated bitcode more closely matches the C/C++ source, making the results more comprehensible. Switch `-g` turns on debugging symbols so SAW can find source locations of functions, names of variables, etc.. In this example, compiled output is placed in `add.bc`. This is not human readable but can be converted to human readable by doing this: `llvm–dis add.bc` the result of which is `add.ll`. The extension `.bc` stands for bitcode. Consult the manual, Page 28, for helpul notes on compiling for SAW.

SAW can load a bitcode module with this:

```
m <- llvm_load_module "add.bc";
```

for the `clang-12` created `add.bc`.

SAW allows for specifying functions in `llvm`: the specification is defined in a `do` block. The `do` block is written in a `.saw` file along with a `llvm_load_module` line as above. For example:

```
let add_spec = do {
  ...
};
```

Inside the do block there are three sections: a specification of the initial state before execution of the function, a description of how to call the function within that state, and a specification of the expected final value of the program state. Part of the initial state specification is the declaration of `llvm` typed variables. For example,

```
x <- llvm_fresh_var "x" (llvm_int 32);
y <- llvm_fresh_var "y" (llvm_int 32);
```

which specifies two `llvm` variables of 32 bit integers. The `add_spec` function is to be called like this with `x` and `y` defined above:

```
llvm_execute_func [llvm_term x, llvm_term y];
```

The expected output for adding 32 bit integers `x` and `y` may be written like this:

```
llvm_return (llvm_term {{ x+y : [32]}});
```

The double brace block (`{{…}}`) encloses a cryptol expression. The `add_spec` specification may be checked for equivalence with the add function of `add.c`, which is in the `llvm` code of `add.bc`, using the following:

```
add_ov <- llvm_verify m "add" [] true add_spec z3;
```

where ‘`m`’ is the loaded llvm module from `add.bc`, ‘`add`’ is the function to be tested against the `add_spec`, ‘`[]`’ means skip any already-verified specifications that may be used for compositional verification (at this point a simple example is being considered and the use of other verified specifications is not necessary), and ‘`true`’ means do path satisfiability checking (in this example `false` works just as well). Of course, ‘`z3`’ is the solver to be used for the equivalence check and `add` is being checked against ‘`add_spec`’ (`cvc4` or `abc` works as well).

Exercise 1:
Create `add.bc` from `add.c` then put all of the above together in a file called `add.saw`, run `saw`, and observe the output. ■

Now consider the following slight modification to add.c, given In file `add_ptr.c`:

```
uint32_t add_in(uint32_t *x, uint32_t y) {
   *x += y;
   return *x;
}

int main (int argc, char** argv) {
   uint32_t a = atol(argv[1]);
   uint32_t b = atol(argv[2]);
   printf("%u\n",add_in(&a,b));
}
```

To deal with pointers use the following `do` block:

```
let ptr_to_fresh(name : String) (type : LLVMType) = do {
   x <- llvm_fresh_var name type;
   p <- llvm_alloc type;
   llvm_points_to p (llvm_term x);
   return (x, p);
};
```

This block returns an `llvm` variable `x` and an `llvm` pointer to `x`, namely `p`. It may be used in other do blocks like this:

```
(x,p) <- ptr_to_fresh "x" (llvm_int 32);
```

In this example the pointer is passed as argument so the following change may be made:

```
llvm_execute_func [p, llvm_term y];
```

Exercise 2:
Create `add_ptr.bc` from `add_ptr.c` then put all of the above together in a file called `add_ptr.saw`, run `saw`, and observe the output. ■

Now consider the following, given in file `rotl.c`:

```
uint32_t ROTL(uint32_t x, uint8_t r) {
   r = r % 32;
   uint32_t bottom = x >> (32 - r);
   uint32_t top = x << r;
   return bottom | top;
}

int main (int argc, char** argv) {
   uint32_t a = atol(argv[1]);
   uint32_t b = atol(argv[2]);
   printf("%u\n",ROTL(a,b));
}
```

Because `r` has a range from 0 to 31 due to this statement:

```
r = r % 32;
```

some precondition statements need to be added to the `llvm` specification. These look like this:

```
llvm_precond {{ 0 < r }};
llvm_precond {{ r < 32}};
```

Exercise 3:
Create `rotl.bc` from `rotl.c` then create `rotl.saw`, run `saw` on that file and observe the output. ■