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 0th bit position, then the 1st bit position, then the 2nd 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 0th 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.

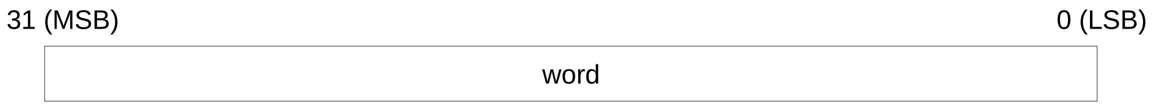
```
uint32_t ffs_ref(uint32_t word) {
   int i = 0;
   int cnt = 0;
   if (!word) return 0;
   for (cnt = 0; cnt < 32; cnt++)
      if (((1 << i++) & word) != 0) return i;
   return 0;
}

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);
}
```

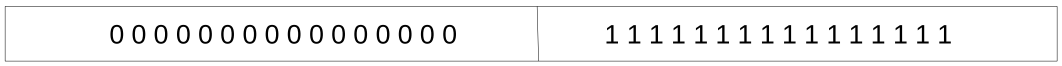
Place the code in a file called `ffs_ref.c`, add the include files (`stdio.h`, `stdlib.h`), compile and test using numbers in the above table (this has already been done).

```
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
```

Now consider an alternative algorithm for solving the same problem. The figure below represents a 32 bit number. The least significant bit is referred to as bit 0 or LSB and is on the right. The most significant bit is referred to as bit 31 or MSB and is on the left.



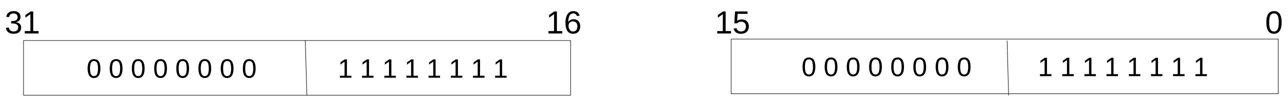
Bitwise 'and' the given word with the following:



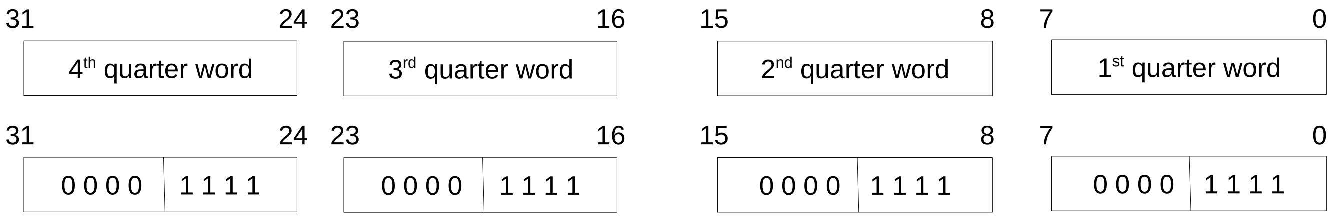
If the first '1' is located in any of bits 0 to 15 then the bitwise 'and' is not zero. Shrink the given word to just bits 0 to 15 and set the number n to 1. But, if the first '1' is located in any of bits 16 to 31 the bitwise 'and' is zero. In that case shrink the word to just bits 16 to 31 and set n to 17. The figure below right shows the first case and below left shows the second case.

31
16
left half of word
15
0
right half of word

In each case, bitwise 'and' with the following:



If the first '1' is in bits 0 to 7, the bitwise 'and' (on the right) is not 0 so shrink the right half word to contain only bits 0 to 7 of the given word and set n to 1. If the first '1' is in bits 8 to 15, the bitwise 'and' is zero so shrink the right half word to contain only bits 8 to 15 of the given word and set n to 9. If the first '1' is in bits 16 to 23, the bitwise 'and' (on the left) is not 0 so shrink the left half word to contain only bits 16 to 23 of the given word and set n to 17. If the first '1' is in bits 24 to 31, the bitwise 'and' is 0 so shrink the left hand word to contain only bits 24 to 31 and set n to 25. The top line of the figure below shows the four cases that may arise.



The bottom line of the figure above shows the 8 bit word that should be bitwise ‘and’ed with the above quarter words in each case. If the first ‘1’ is in bits 0-3 or 8-11 or 16-19 or 24-27 then the bitwise and is not 0 and set n to 1, 9, 17, 25 respectively. If the first ‘1’ is in bits 4-7 or 12-15 or 20-23 or 28-31 then the bitwise ‘and’ is 0 and set n to 5, 13, 21, 29 respectively. Shrink the appropriate quarter word to the four bits that contain the first ‘1’. Repeat the above by ‘and’ing the selected 4 bits of the given 32 bit word with this:

0 0	1 1

Then again, ‘and’ing with this:

0	1

and setting n accordingly. The range of number n will be 1 to 32. If there is no 1 in word then set n to 0. Return n.

Exercise 1:
Write a C program that implements the above described algorithm for finding the first 1 in a 32 bit number. Place the C code in a file called `ffs_imp.c`. If you do not know the C language well enough to do this, look at the solution to this exercise for the C code. You are asked to write the C program so that when you use SAW to verify equivalence to `ffs_ref` you will be emulating a real situation and will gain an understanding of how the SAW tool will be applied and the result analyzed until equivalence is achieved. The expectation is that your C code will have at least one bug and they will be found by using SAW. ■

Comparing C code is easiest by translating to `llvm` then comparing `llvm` code. Recall clang is used to translate to `llvm` like this (in Windows the bc files are provided in case there is difficulty installing or using clang):

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

In SAW an `llvm` module is created from bitcode like this:

```
m1 <- llvm_load_module "ffs_ref.bc";
m2 <- llvm_load_module "ffs_imp.bc";
```

Then `llvm` code is extracted for a function like this:

```
ffs_ref <- llvm_extract m1 "ffs_ref";
ffs_imp <- llvm_extract m2 "ffs_imp";
```

Equivalence is proved on the llvm code. One way to do this is:

```
let thm1 = {{ \x - > ffs_ref x == ffs_imp x }};
result < - prove z3 thm1;
print result;
```

Exercise 2:
Create SAW file `ffs_imp.saw` and add statements as above to prove the equivalence of `ffs_ref` and `ffs_imp`. Then run `saw` on the file and show the result. ■

A length-n deBruijn sequence, where `n` is a power of 2, is a cyclic sequence of `n` 0's and 1's such that every 0-1 sequence of length `lg(n)` occurs exactly once in a contiguous substring.

Example: a length-8 deBruijn sequence is 00011101 because each 3-bit number occurs exactly once as a contiguous substring:

```
   000 (0 - 1st from left),
   001 (1 - 2nd from left),
   011 (3 - 3rd from left),
   111 (7 - 4th from left),
   110 (6 - 5th from left),
   101 (5 - 6th from left)
   010 (2 - 7th from left and wrapping),
   100 (4 - 8th from left and wrapping)
```

Note: if `w` = 01101000 then `-w` = 10011000 (2s complement of `w`) and the bitwise ‘and’ (`w & -w`) is 00001000. Thus the only 1 in (`w & -w`) is the rightmost 1 in `w`.

Suppose `x` is an 8 bit number with a single 1 in one of 8 locations, produced by the above bitwise anding. Compute `h(x) =` $\{x*deBruijn\}_8$ `>> (8 – lg(8))` where $\{m\}_8$ means retain lower order 8 bits of `m`. The table above, for the 8 bit deBruijn sequence 00011101, maps h(x) to the position of the first 1 in `x`: N^{th} from left translates to first 1 is in position (`N`-1) from right.

Example: for the length-8 sequence `deBruijn` = 00011101, `8 - lg(8) = 5.`

h(x)	index	explanation
000	0	suppose x is 00010000
001	1	multiply by 00011101 to get 111010000
010	6	retain low order 8 bits: 11010000 then >> 5 to get 110
011	2	110 in table maps to 4, the position of first 1 from right
100	7	suppose x is 00000100
101	5	multiply by 00011101 to get 1110100
110	4	retain lower 8 bits: 1110100 then >> 5 to get 011
111	3	011 in table maps to 2 which is where the 1 is from right

For 32 bits:
  The number 0x76be629 = 00000111011010111110011000101001 = a deBruijn sequence
  32 - lg(32) = 32 – 5 = 27

If a 1 exists in a given 32 bit input `x` then `h(x)` is (x & -x)*0x76be629 >> 27
and the index of `h(x)`, taken from the table below, is where the first 1 is, if there is a 1:

h(x)	index	h(x)	index	h(x)	index	h(x)	index
0	0	8	30	16	31	24	21
1	1	9	27	17	22	25	17
2	23	10	25	18	28	26	9
3	2	11	11	19	18	27	6
4	29	12	20	20	26	28	16
5	24	13	8	21	10	29	5
6	19	14	4	22	7	30	15
7	3	15	13	23	12	31	14

The index numbers need to be incremented by 1 to match the location obtained by `ffs_ref`. Also, the case of no 1s in `x` must be accounted for.

Exercise 3:
Write C code that implements a solution to the problem of finding the first 1 in a given 32 bit input. Place the code in file `ffs_mus.c`. Create SAW file `ffs_mus.saw` and add statements as above to prove the equivalence of `ffs_ref` and `ffs_imp`. Then run `saw` on the file and show the result. ■

Exercise 4:
Copy `ffs_ref.c` to `ffs_bug.c` and introduce a bug in `ffs_bug.c` like this:

```
  if (word == 1052688) return 4; /* instead of 5 (in hex: 0x101010) */
```

Then create `ffs_bug.bc` with `clang` and the `ffs_bug.saw` file for checking the equivalence of `ffs_bug` and `ffs_ref`. Run the saw file and observe the result.