Report for Project-1

Author: Li Minchao ID: 515030910361

1 Synopsis of the project

1.1 Demand of the project

To achieve some bit-level function using C grammar.

The limitation is to complete each function skeleton using only straightline code for the integer puzzles and a limited number of C arithmetic and logical operators.

The operator set for each function is given, and the max op is anticipated.

1.2 Purpose of the project

To become more familiar with bit-level representations of integers and floating numbers.

To work my own way through working it and make the algorithm as simple as possible.

2 Environment of the project

OS:Linux Ubuntu Kylin 16.04

Compile under gcc v5.4.0

3 Implementation process of the project

I will illustrate how I managed to implement the function and conquer the barriers in this section.

3.1 bitAnd

Introduction: receive two arguments (x, y), and return x & y using only $\sim$ and |

Solution:

Mainly use the De Morgan's law, $\overline{A \cup B} = \overline{A} \cap \overline{B}$.

Thus, in this case, we have the equation, $\overline{\overline{A} \cup \overline{B}} = A \cap B$ to solve this problem.

3.2 getByte

Introduction: receive two arguments (x, n), and return the extract byte n from word x.

Solution:

Since one byte has 8 bits, the shift amount (I name this para as movIndex in the function) should be 8 · n.

Thus, we get the equation, $movIndex = n \ll 3$.

And the answer is $(x \gg movIndex)$ & 0xff.

3.3 logicalShift

Introduction: receive two arguments (x, n), and shift x to the right by n, using a logical shift

Solution:

I declare a parament base $= 1 \ll 31 =$ 0x80000000.

Since base is marked as signed integer, during the process of left shift, 1 is extended.

By using $\sim (base \gg (n-1))$, we get $00 \cdots 0 \underbrace{11 \cdots 1}_{32-n\ 1s}$, which serve as the mask.

However, - operation is not allowed. To achieve this operation, we can implement $+ (\sim 1 + 1)$.

While in this case, if we directly use n + 0xffffffff, we will get unexpected case when n is 0.

Since the low bit is always 0, we can left shift n, and right shift 1. So the mask is always what I expect.

Taking the advantage of the mask, no matter whether the high bit is extended by 0 or 1, we can always have the right answer.

3.4 bitCount

Introduction: receive one argument x, and returns count of number of 1's in word.

Solution:

This is quite difficult for me, and probably the hardest one in this project.

The ceiling of operation number is too limited, and I cannot make a simple solution.

I bing-ed the relative solution, and there I found the solution based on divide and conquer methodology.

I finally accomplish this function by the hint and I will explain how it works above.

The core is that x saves the answer of partial bits in its corresponding part.

First, we have this assignment, $x = (x \,\&\, 0x55555555) + ((x \gg 1) \,\&\, 0x55555555)$, which counts the number of 1's of some 2 consecutive bits.

In the equation above, $x \,\&\, 0x55555555$ and $(x \gg 1) \,\&\, 0x55555555$ represents the bits only in odd or even position.

Take 0x1110 as an example, we add 0x0100 and 0x0101 (which is the result of $x \,\&\, 0x55555555$ and $(x \gg 1) \,\&\, 0x55555555$), and get 0x1001 as the result, which 10 stands for 2, is the number of 1's in 11(the 2 and 3 bit) and 01 stands for 1, is the number of 1's in 10(the 1 and 0 bit).

Every assignment in the function is to add two consecutive blocks which store answer of partial bits. The add operation is between either of 1-bit operands or 2-bit operands or 4-bit operands or 8-bit operands or 16-bit operands.

We can view the initial x as one bit store the answer of 1's in one bit, and generally x use n bits to store the answer of 1's in n bits. Finally, we use 32 bits to store the answer of the 1's of the whole initial x.

However, we cannot use those big int straightly.

To overcome this, I declare 5 variables and got the value using $\ll$ and $|$.

For example:

mask1 $= (0x55 \ll 8) \,|\, 0x55;$

0x55555555 $= (mask1 \ll 16) \,|\, mask1;$

3.5 bang

Introduction: receive one argument x, and return !x without using !.

Solution:

If there is any 1 in any bit, we return 0.

So, what I do is to calculate $b_0 \cup b_1 \cup \cdots \cup b_{31}$.

First, I do this assignment $x = x \mid (x \gg 16)$, which record information of 32 bits into the low 16 bits.

If there is any high bit in either low 16 bits or high 16 bits, the low 16 bits will have high bit.

Likely, I do the similar assignments to condense information into less bits.

Finally, all information is recorded in the b_0.

Return 0, if b_0 is 1, and return 1, if b_0 is 0.

3.6 tmin

Introduction: receive no arguments, and return minimum two's complement integer.

Solution:

Just return $1 \ll 31$.

3.7 fitsBits

Introduction: receive two arguments (x, n), and return 1 if x can be represented as an n-bit, two's complement integer.

Solution:

If I define the bit sequence is $(b_{31}, b_{30}, \cdots, b_1, b_0)$, the key of this function is to see if the $(b_{31}, b_{30}, \cdots, b_{n-1})$ is the sequence made of same number.

Since x is a signed integer, sign extended will be exerted when right shifted. So I left shift (n-1) bits, and if all bits are 0 or 1, return 1.

Well, how can I implement n-1. One way is to add 0xffffffff, however, such integer is not allowed in this project. So, I use a trick to make 0xffffffff by implementing $(1 \ll 31) \gg 31$.

To determine this, I just need to add b_{n-1} to the number. If the result is 0, the function will return 1.

3.8 divpwr2

Introduction: receive two arguments (x, n), and return $x/\left(2^n\right), for\ 0 \leq n \leq 30$.

Solution:

In the case of $x \geq 0$, it is quite easy, we just need to left shift n.

In the case of $x < 0$, it is complicated. We have to consider when and when not to add 1 to the result.

Use $sgn = (x \gg 31)\ \&\ 1$ to record the sign bit.

If x can be divided by 2^n, we don't have to add 1. I get the low bit of x by $x \,\&\, (\sim x + 1)$. And if we left shift low bit by n is not zero, it means x can be divided by 2^n.

We left shift the low bit by n bits, if the result is not zero, it means x can be divided by 2^n.

Use $sgn2 =! \,(lowbit \gg n)$ to mark this situation.

Return $(x \gg n) + (sgn\ \&\ sgn2)$

3.9 negate

Introduction: receive one argument x, and return -x.

Solution:

According to the 2'compliment rule, simply return $\sim x + 1$.

3.10 isPositive

Introduction: receive one argument x, and return 1 if $x > 0$, otherwise, return 0.

Solution:

For a signed integer, the highest bit indicates if it's positive.

If it's 1, it's negative, otherwise, it's positive.

Use $sgn = (x \gg 31)\ \&\ 1$ to record the bit.

However, we should return 0, if x = 0.

So, we should return $(!sgn)\ \&\ !\,(!x)$

3.11 isLessOrEqual

Introduction: receive two arguments (x, y), and return 1 if $x \leq y$, otherwise, return 0.

Solution:

I notate the sign of x and y as x_sgn and y_sgn. I notate not_equal = x_sgn ∧ y_sgn.

If x and y have different sign, not_equal = 1. And the answer should be x_sgn.

If x and y have same sign, not_equal = 0. And the answer should depend whether $y + (\sim x + 1) \geq 0$.

So we only need to check whether the highest bit of $y + (\sim x + 1)$ is 0.

Finally, use the structure of (not_equal & $\cdots$) | (!not_equal & $\cdots$) to do the branch.

3.12 ilog2

Introduction: receive one argument x, and return floor(log base 2 of x), where $x > 0$.

Solution:

The key of this function is to return the highest index such that $b_{index} = 1$. Index ranges from 0 to 31.

Since $x > 0$, 0 is extended if x is left shifted.

If $x \gg 16 = 0$, it means we don't have 1 at the high 16 bits. Otherwise, the answer is at least 16 big. We use $sgn =!\,(!\,(x \gg 16))$ to mark if we have 1 at high bits. And we use $cnt = sgn \ll 4$ to accumulate the answer.

After these operation, we left shift x, $x = x \gg cnt$. If we don't have 1 at high bits, we then only focus on the low bits ranging from 0 to 15. If we have 1 at high bits, we then focus on the bits ranging from 16 to 31, and ignore the low bits, since we already know high bit is at 16 to 31.

Do the similar operation, and change the constant from 16 to 8 and 4 and 2 and 1. And we can traverse the number bits.

Finally we accumulate those cnt, and return the sum.

3.13 float_neg

Introduction: receive one argument uf, and return bit-level equivalent of expression -f for floating point argument f. Both the argument and result are passed as unsigned int's, but they are to be interpreted as the bit-level representations of single-precision floating point values. When argument is NaN, return argument.

Solution:

Floating number is mainly divided into three parts: s, e, m.

s, is the b_{31}, represents the sign of the number.

e, ranges from $b_{23}\ to\ b_{30}$, represents the power of 2.

m, ranges from $b_0\ to\ b_{22}$, represents the base.

To get the s, e, m part, we can shift the number and & the corresponding mask.

If uf doesn't represent NAN, which means e part is 0xff and m part is not 0, simply return uf.

If not, we simply change the sign bit.

3.14 float_i2f

Introduction: receive one argument x, and return bit-level equivalent of expression (float) x. Result is returned as unsigned int, but it is to be interpreted as the bit-level representation of a single-precision floating point values.

Solution:

Do branch case if x is 0 or 0x80000000.

Detect and save the sign of x, if x is negative, we assign $x = -x$.

Then we should get the index of the highest bit of 1, the implement of which is similar with ilog2.

If $index > 23$, we should also consider the deviation part which determines whether we take the floor or the ceiling to minimize the deviation. If the deviation is the same, we select the number which is even.

The value of index also serve us to calculate e part. We just simply add index to the offset 127 and assign the result to e.

Everything logical can be accomplished by branching. Finally using the mask skills to combine s, e, m part.

However, I fail to accomplish this function within the max op.I cannot carry out any way to limit total operations into 30.

3.15 float_twice

Introduction: receive one argument uf, and return bit-level equivalent of expression $2 \cdot f$ for floating point argument f. Both the argument and result are passed as unsigned int's, but they are to be interpreted as the bit-level representation of single-precision floating point values. When argument is NaN, return argument.

Solution:

Regularly, I calculate the value of s, e, m part using shift operation and mask skill.

If e part is zero, it means we cannot shift e part for e is always 0, and we should left shift the m by 1.

Else if e part is 0xff, we just return uf. Since no matter it's infinite or NAN, uf itself is the answer of the function.

Else, we simply add 1 to e part, and refresh e part to the initial uf.

4 Review of the project

This project makes me to think about many details of those familiar operations and realize the implement or accomplishment of those seemingly regular functions is not that easy. Never will I know how it works until I try it myself, since these functions are already packaged. I can feel the efficiency of those standard implementation of the function compared with my naive way of implementation.

Also, I come to know how the floating number is actually stored which is only a vague picture in my mind before. However, comprehension is one thing and practise is another. there are more things to be taken into consideration since the variable state of floating number and the existing deviation compared with integers.

Finally, I appreciate it much for some online documents and blogs which solve many of my confusion.

5 Reference

http://www.cnblogs.com/graphics/archive/2010/06/21/1752421.html

https://wenku.baidu.com/view/f8fc95fad1f34693daef3e67.html

http://blog.csdn.net/zhzhanp/article/details/6339883