CPEG-323: Introduction to Computer Systems Engineering

Machine Problem #1: Bitwise Operations

You may work alone, or in groups of up to two.

Due on Friday, September 28 (11:59:59pm)

1. Learning Objectives

This MP is intended to give you a chance to acquaint yourself with the bit-wise logical operators and iteration in a higher level language, before we introduce these concepts in MIPS. It will also give you a chance to familiarize yourself with a chunk of C code that we will be implementing in MIPS in later MPs.

2. Introduction

In this MP, you will be programming in C. If you are already familiar with C and bitwise operations in C, feel free to skip this section.

Compilation: Use the following command to compile sudoku.c. It will produce an executable called "sudoku". The compiler will output any warnings that occur during compilation. Although it is not necessary to turn in a code that compiles with no warnings, you may wish to eliminate all of them, as they may be indicative of bugs.

```
g++ -Wall –o sudoku sudoku.c
```

Bit Arrays: In this MP, you will be dealing with bit arrays. In C, it is possible to use an integer to store a bit, and then use an array of integers. However, we are going to view an integer as being a length 32 bit array. So, the *n*th bit in the array will be the *n*th bit of the integer. If we need a bit array longer than 32 bits, we can declare an array of integers, and then the first integer stores bits 0 through 31, the second integer stores bits 32 through 63, and so on.

Useful operators:

- << - the left shift operator. It will shift an integer value left the appropriate number of bits.
- >> - the right shift operator. Similar to <<. However, when right shifting a value with the highest bit set, it would be a good idea not to make any sort of assumptions about whether a 0 or 1 is used to fill in the high bits.
- | - the bit-wise logical OR operators. You can view this as a SET operator. All of the bits set in the second argument will be set in the first argument. For example, x = x | 0x01 will set the first bit in x.
- & - the bit-wise logical AND operators. You can view this as a FILETER operator. This will return only the bits of the first argument which are specified by the second argument. For example, x & 0x01 will return the first bit of x.
- ~ - the bit-wise logical NEGATE operator. This will invert all of the bits of a value. You can use this along with the & operator to CLEAR bits. For example, x = x & ~0x01 will clear the first bit in x. ~0x01 will return

a bit array with every bit except for the first one set, and then the & operator will filter out the first bit.

3. Problem

In this MP, you will complete a program that solves Sudoku puzzles. If you do not know what Sudoku is, please Google it now and then continue reading this assignment.

The program uses bit-masks to represent the possible states for each square of the Sudoku puzzle. As each square can be any one of 9 numbers (1-9), we will use 9-bits (bits 0-8, where bit 0 is the least significant bit (LSB)) to represent which of the numbers each square can be. For example, if the value stored for a square is 0x14F where bits 0,1,2,3,6, and 8 are set, then this square can be the values 1,2,3,4,7, and 9 and not the values 5,6, and 8.
The key to solving Sudoku puzzles is eliminating possibilities of an unknown square until there is only one remaining possibility that must be the answer. We use two rules to eliminate possibilities:

- **Rule 1:** If we know the value for a given square (i.e., it is a "singleton" where there is only one possible for the value for the square), then no square in the same row, column, or 3*3 square can hold the same value. That means that we can remove that possibility from all of those squares.
- **Rule 2:** If there is only one square in a given row, column or 3*3 square that can hold a given value, then it must be the one that holds that value. All other possibilities can be eliminated from that square.

These two rules are sufficient to solve all but the most difficult Sudoku puzzles. There are two parts in this MP. In the first part, you will develop some helper methods necessary to get the code to work. In the second, you implement rule 2.

Part 1: Bit-twiddling functions (4 * 20 points)
You will need to implement four functions:

- **bit_count**: This function takes an integer and returns the count of the number of bits set(e.g., given the value 0x34 the function return the value 3). This can be done with a loop, checking whether each bit is set and incrementing a counter for each set bit.
- **get_nth_set_bit**: This function takes an integer and an index; it iterates through the bits of the integer (from LSB to MSB) counting set bits until it comes to the "index"th set bit. The current bit position is returned. For example, given the value 0x34 the function would return the following values 2,4,5 for the indices 0,1,2 respectively. The method would fail if an index of 3 or greater were provided, since only 3 bits are set in 0x34.
- **singleton**: This function returns a bool (a type which can hold either true of false) based on whether a single bit is set. This can be done more efficiently than using the bit_count method by using the following trick:

  If you have an integer with only one bit set and subtract one from it, you get an integer with all bits below the bit set (but not includeing i) set (i.e., subtracting 1 from 0x100 give 0x0ff). If you AND the original number and the new number together you will get zero. This is only true for integers with one bit set and the value zero.

Please implement this method by checking two conditions for a singleton: 1) that the integer isn't zero, and 2) the AND of it and (it -1) is zero. This function should return true for 0x40 and false for 0x3.

- **get_singleton**: Get the position of the set bit for a singleton. This is much like "get_nth_set_bit".

Part 2: Rule 2 (20 points) Use the code for Rule 1 as a guide, implement Rule 2. We have provided psedudocode to help you:

```
changed = false
for i in 1 to 9:
    for j in 1 to 9:
        value = board[i][j]
        if value is not a singleton:

            isum = UNION of board[k][j] for k in 1 to 9, where k ! = i
            if isum is not all possibilities:
                    set board[i][j] to possibility not in isum
                    changed = true

            jsum = UNION of board[i][k] for k in 1 to 9, where k!= j
            if jsum is not all possibilities:
                    set board[i][j] to possibility not in jsum
                    changed = true

            ksum = UNION of board[x][y] for each square in 3*3 square not including(i,j)
            if ksum is not all possibilities:
                    set board[i][j] to possibility not in ksum
                    changed = true
    endfor
  endfor
```

Note: This is a challenging problem!

We strongly encourage you to test your code. We've provided some simple test cases; you are encouraged to develop your own as well.

4. What to submit:

You will need to submit a sudoku.c and a groupInfo.txt containing the names of everyone in your group, one per line. You may work alone, or in a group up to two people. Each group should submit the MP once by sending emails to cpeg323.udel@gmail.com.