He who hasn't hacked assembly language as a youth has no heart. He who does so as an adult has no brain. John Moore

Real programmers can write assembly code in any language. Larry Wall

Learning Objectives

This lab involves writing MIPS procedures. Specifically, the concepts involved are:

1. Arithmetic and logical operations in MIPS
2. Arrays and pointers in MIPS
3. MIPS control flow (conditionals, loops, etc.)
4. MIPS function calling conventions

Work that needs to be handed in

1. `p1.s`: implement the `has_single_bit_set` and `get_lowest_set_bit` functions in MIPS. **This is due by the first deadline.**

   Run on EWS with: `QtSpim -file p1_main.s p1.s`

2. `p2.s`: implement the `board_done` and `print_board` function in MIPS. **This is due by the second deadline.**

   (These functions must use the helper functions `has_single_bit_set` and `get_lowest_set_bit` from `p1.s`. **Do not include that code in `p2.s`; just include `p1.s` on the command line.)**

   Run on EWS with: `QtSpim -file p2_main.s p2.s p1.s`

Important!

For Lab 7 we are providing "main" files (*e.g.*, `p1_main.s`) and files for you to implement your functions (*e.g.*, `p1.s`). We will need to load both of these files into QtSpim to test your code.

We will only be grading the `p1.s` and `p2.s` files, and we will do so with our own copy of `p*_main.s`, so make sure that your code works correctly with an original copy of `p*_main.s`. We provide you with a `sudoku.c` file that contains a C implementation of all the functions that you need to implement for this Lab.

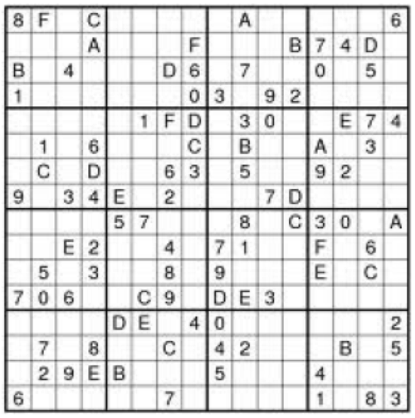
Guidelines

- You may use any MIPS instructions or pseudo-instructions that you want.
- Follow all function-calling and register-saving conventions from lecture. **If you don't know what these are, please ask someone.** We will test your code thoroughly to verify that you followed calling conventions.
- We will be testing your code using the EWS Linux machines, so we will consider what runs on those machines to be the final word on correctness. Be sure to test your code on those machines.
- **Our test programs will try to break your code.** You are encouraged to create your own test programs to verify the correctness of your code. One good test is to run your procedure multiple times from the same main function.

Sudoku

In our MIPS programming, we'll be writing solvers for Sudoku puzzles. If you do not know what Sudoku is, please Google it now and then continue reading this assignment.

We're not just solving the traditional Sudoku, however... we're going to solve 4x4 Sudoku as shown below.



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 16 values (numbers 1-9 and letters A-G), we will use 16 bit values to represent which of the values each square can be. For example, if the value stored for a square is 0x014f 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, 8, A, B, C, D, E, F, and G.

The key to solving Sudoku puzzles is eliminating possibilities for an unknown square until there is only one remaining possibility, which must be the answer. We use two rules to eliminate possibilities:

Rule 1: If we know the value for a given square (**i.e.**, there is only one possible for the value for the square), then no square in the same row, column, or 4x4 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 4x4 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.

A complete Sudoku solver is included in your Lab 7 github directory in a file called `sudoku.c`. A number of sample Sudokus are included in a file called `boards.h`. We will be implementing a portion of this C code and helper functions in MIPS for Labs 7 and 8.

If the value of a square of our Sudoku board is known, then it contains only a single set bit. These first two functions are useful for identifying when a value is known and extracting the value associated with the square.

Problem 1: `has_single_bit_set` [15 points]

This function takes an unsigned (32-bit) integer as an argument and returns a boolean indicating if the argument has exactly one bit set. It uses a cute binary number trick `(value & (value - 1))` to do this; see if you can understand how this works.

```
bool has_single_bit_set(unsigned value) {  // returns 1 if a single bit is set
  if (value == 0) {
    return 0;   // has no bits set
  }
  if (value & (value - 1)) {
    return 0;   // has more than one bit set
  }
  return 1;
}
```

Translating the `has_single_bit_set` to MIPS only requires "if" control flow, arithmetic, and calling conventions.

Problem 2: `get_lowest_set_bit` [15 points]

This function takes an unsigned (32-bit) integer as an argument and returns an unsigned (32-bit) integer. If the argument has at least one of the 16 least significant bits set, the returned value has a single bit set. The returned value's set bit is the same as the least significant bit in the argument that is set. If the argument doesn't have any of the 16 least signficant bits set, then the function returns zero.

```
unsigned get_lowest_set_bit(unsigned value) {
  for (int i = 0 ; i < 16 ; ++ i) {
    if (value & (1 << i)) {          # test if the i'th bit position is set
      return i;                      # if so, return i
    }
  }
  return 0;
}
```

The `get_lowest_set_bit` function requires you to implement a loop.

The next two problems are similar in structure, involving doubly-nested loops, indexing two-dimensional arrays, and needing to save/restore registers and call other functions. As such, **we recommend writing the first and completely debugging it and then using that as a template for the second one**.

Our Sudoku boards are represented in C as `unsigned short board[16][16]`. Two dimensional arrays in C are laid out contiguously in memory like a single-dimension array; for an MxN array (int A[M][N]) the locations are laid out in the following order: A[0][0], A[0][1], A[0][2], ... A[0][N-1], A[1][0], A[1][1], ... A[1][N-1], A[1][0], ... A[M-1][N-1]. The address of element A[i][j] is:

the address of A[0][0] + (((i*N) + j) * sizeof(element))

The two dimensional arrays that you are dealing with are fixed size (16x16) matrices and of `unsigned short`s, so N is the constant 16 and sizeof(element) is 2. To load `unsigned short`s use the `load half unsigned` (`lhu`) instruction.

These functions use both of the functions that you implemented in `p1.s`. Don't replicate these in `p2.s`.

Problem 3: `board_done` [35 points]

When we've completely solved a Sudoku puzzle, every single square should only have a single bit set. The `board_done` function tests for this condition.

```
bool
board_done(unsigned short board[16][16]) {
  for (int i = 0 ; i < 16 ; ++ i) {
    for (int j = 0 ; j < 16 ; ++ j) {
      if (!has_single_bit_set(board[i][j])) {
        return false;
      }
    }
  }
  return true;
}
```

This function calls the `has_single_bit_set` that you completed in `p1.s`. Call this code with a `jal` instruction and be sure to adhere to the calling convention.

Problem 4: `print_board` [35 points]

This function prints out the Sudoku, placing an asterisk anywhere there is an unknown.

```
void
print_board(unsigned short board[16][16]) {
  for (int i = 0 ; i < 16 ; ++ i) {
    for (int j = 0 ; j < 16 ; ++ j) {
      int value = board[i][j];
      char c = '*';
      if (has_single_bit_set(value)) {
        int num = get_lowest_set_bit(value) + 1;
        c = symbollist[num];
      }
      putchar(c);
    }
    putchar('\n');
  }
}
```

`putchar` prints a single character. In SPIM, you can print characters to the console using the 11th system call in Appendix A of the textbook.[1] You put the character you want to print in register `$a0`, and then put `11` in the register `$v0` and execute a `syscall` instruction, as shown below.

```
li $v0, 11          # for the 11th syscall
syscall
```

SPIM also supports character constants, so you can just use `'*'`, `'\n'`, etc.

Suggestion

Much of the course staff finds this code easier to write using the (callee-saved) $s registers! $s registers allows you to save registers once at the beginning of the function to free up the $s registers; you can then use $s registers for all values whose lifetimes cross function calls.

[1]A complete list of the available system calls can be found in the Appendix A PDF at the bottom of the course's assignments web page.