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 GIT

1. `p1.s`: implement the `craftable_recipes` function 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 `pattern_match` functions in MIPS. **This is due by the second deadline.**

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

Important!

This is a solo lab. This means you are free to talk with other students about the problem at a high level, but cannot share code or work on another person's code.

For Lab 6 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 `p1_main.s` and `p2_main.s`, so make sure that your code works correctly with the original copy of `p1_main.s` and `p2_main.s`. We provide you with a `code.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.

- Keep your code clean and follow the style guides on the website. **The course staff can and will refuse to help you if your code is not formatted correctly**.

Problem 1: `craftable_recipes` [40 points]

The functions covered in lab 6 will be useful in a future lab called Lab Spimbot. In Lab Spimbot, one of the things you will be doing is working in a 2D grid and using materials from the world to craft items using recipes, much like one would in Minecraft or Rust. The first function we will have you implement is a simple one that checks what recipes are craftable and what is the max number of times a recipe can with the given inventory.

A visual example

Let us simplify the problem a fair bit. In our game, we have two craftable recipes: a bed and a pickaxe. A bed requires 3 wood and 3 wool while a pickaxe requires 2 wood and 2 stone. The recipes matrix would be encoded as

$$\begin{bmatrix} 3 & 3 & 0 \\ 2 & 0 & 2 \end{bmatrix}.$$

In this matrix, the first row represents the bed recipe and the second row represents the pickaxe recipes. The first column represents the amount of wood required for the recipe, the second represents the amount of wool required, and the third represents the amount of stone required.

The inventory shares the same format as a row of the recipes matrix. So, an inventory with 6 wood, 6 wool, and 3 stone would be encoded as the array

$$\begin{bmatrix} 6 & 6 & 3 \end{bmatrix}.$$

The times_craftable array will hold the maximum number of times each recipe can be crafted. So, with our inventory, we can craft at most 2 beds and at most 1 pickaxe. The times_craftable array would be encoded as

$$\begin{bmatrix} 2 & 1 \end{bmatrix}.$$

We have given a function to do this in C and it is your job to translate this into MIPS assembly code.

In the following function we are passed three parameters. Each element in the inventory represents a count of a type of resource. Recipes is a 2D array (see the **Appendix** for details). Each row of recipes is a recipe whose columns consist of the required amount of each item (same format as the inventory). Finally, `times_craftable` is where we store the amount of times a recipe is craftable. Each element in `times_craftable` corresponds to the number of times the corresponding row in `recipes` can be crafted.

Code

```
/**
 * Given a table of recipes and an inventory of items, this function
 * will populate times_craftable with the number of times each recipe
 * can be crafted.
 *
 * Note: When passing arrays as parameters, the register $a0 will hold the starting
 * address of the array, not the contents of the array.
 */

void craftable_recipes(int inventory[5], int recipes[10][5], int times_craftable[10]) {
    const int NUM_ITEMS = 5;
    const int NUM_RECIPES = 10;

    for (int recipe_idx = 0; recipe_idx < NUM_RECIPES; recipe_idx++) {
        times_craftable[recipe_idx] = 0x7fffffff;  // Largest positive number
        int assigned = 0;

        for (int item_idx = 0; item_idx < NUM_ITEMS; item_idx++) {
            if (recipes[recipe_idx][item_idx] > 0) {
                // If the item is not required for the recipe, skip it
                // Note: There is a div psuedoinstruction to do the division
                // The format is:
                //    div   $rd, $rs, $rt
                int times_item_req = inventory[item_idx] / recipes[recipe_idx][item_idx];
                if (times_item_req < times_craftable[recipe_idx]) {
                    times_craftable[recipe_idx] = times_item_req;
                    assigned = 1;
                }
            }
        }

        if (assigned == 0) {
            times_craftable[recipe_idx] = 0;
        }
    }
}
```

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

Problem 2: pattern_match [60 points]

This problem deals with pattern matching using convolution (see https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1 for a good visual example). Convolution is the process of laying a small patch of over another image, doing element-wise multiplication over the overlap, and summing the results. This can be seen as a correlation measure of the two regions and if the result is high enough, then you have a match. So for this problem, we will be using 2D convolution in order to find the first region in the map that has more than `threshold` number of matching elements. We will return this region as a compressed integer where the top 16 bits contain the row and the bottom 16 bits contain the column. If no match is found then the function returns -1.

```
/**
 * This function matches a 5x5 pattern across the map using 2D convolution.
 * If the correlation between the pattern and a 5x5 patch of the map is above the
 * given threshold, then the left hand corner of the patch will be returned.
 * If no match was found, then -1 is returned.
 */
int pattern_match(int threshold, int pattern[5][5], int map[16][16]) {
    const int PATTERN_SIZE = 5;
    const int EDGE = 16 - 5 + 1;

    for (int row = 0; row < EDGE; row++) {
        for (int col = 0; col < EDGE; col++) {
            int sum = 0;
            for (int pat_row = 0; pat_row < PATTERN_SIZE; pat_row++) {
                for (int pat_col = 0; pat_col < PATTERN_SIZE; pat_col++) {
                    if (pattern[pat_row][pat_col] == map[row + pat_row][col + pat_col]) {
                        sum += 1;
                    }
                    if (sum > threshold) {
                        return (row << 16) | col;
                    }
                }
            }
        }
    }
    return -1;
}
```

Appendix

2D Array Indexing

Key takeaway: To access `array[1][1]` in a 3x3 matrix, the equivalent 1D array access would be `array[4]`. We can generically write the formula for any 2D array access `array[row][col]` as `array[row * NUM_COLUMNS + col]`.

In this lab we use 2D arrays. Fundamentally these 2D arrays are just like 1D arrays that we have covered in class with some fancy indexing. The reason for this is that, in C, a 2D array (`int array[][]`) is laid out in **row major ordering** as if it was a 1D array. This means that the elements in one row are adjacent to each other and after one row out directly after the contents in another row. For example, the 3x3 matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

looks like the following in memory.

1	2	3	4	5	6	7	8	9

Arrays as parameters

In C, you may sometimes see the functions with the signature `void foo(int arr[3][3])`. In assembly, we do not pass the contents of the array, but instead the address of the first element (`array[0][0]`). As an example, the function

```
int foo(int arr[3]) {
    int prod = 1;
    for (int i = 0; i < 3; i++) {
        prod *= arr[i];
    }
    return prod;
}
```

turns into the following assembly

```
foo(int*):
        li      $v0, 1                  # int prod = 1;
        li      $t0, 0                  # int i = 0;
        li      $t1, 3                  # set loop bound
loop:
        bge     $t0, $t1, loop_end      # i < 3

        mul     $t2, $t0, 4             # i * sizeof(int)
        add     $t2, $a0, $t2           # arr + sizeof(int)
        lw      $t2, 0($t2)             # arr[i]
        mul     $v0, $v0, $t2           # prod *= arr[i]

        add     $t0, $t0, 1             # i++
loop_end:
        jr      $ra
```

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.