"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 (via SVN)

First deadline

1. `detect_parity.s`: implement the `detect_parity` function in MIPS. Run with:
   `QtSpim -file common.s detect_parity_main.s detect_parity.s`
2. `max_conts_bits_in_common.s`: implement the `max_conts_bits_in_common` function in MIPS. Run with:
   `QtSpim -file common.s max_conts_bits_in_common_main.s max_conts_bits_in_common.s`
3. `twisted_sum_array.s`: implement the `twisted_sum_array` function in MIPS. Run with:
   `QtSpim -file common.s twisted_sum_array_main.s twisted_sum_array.s`

Second deadline

1. `accumulate.s`: implement the `accumulate` function in MIPS. Run with:
   `QtSpim -file common.s accumulate_main.s accumulate.s detect_parity.s max_conts_bits_in_common.s`
2. `calculate_identity.s`: implement the `calculate_identity` function in MIPS. Run with:
   `QtSpim -file common.s calculate_identity_main.s calculate_identity.s accumulate.s twisted_sum_array.s max_conts_bits_in_common.s detect_parity.s`

Important!

Unlike previous labs, this is a solo lab and you may not work with anyone else on this lab.

We also cannot stress enough the importance of reading the *entire* lab handout.

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

We will only be grading the files you implement your functions in, and we will do so with our own copy of the main files, so make sure that your code works correctly with original copies of those.

Guidelines

- You may use any MIPS instructions or pseudo-instructions that you want. A full list of instructions can be found at `http://cs.wheatonma.edu/mgousie/comp220/SPIM_Quick_Reference.html#instructions`
- 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.
- We have a set of style guidelines for MIPS (`https://wiki.illinois.edu//wiki/display/cs233sp17/MIPS+style+guidelines`). Assembly code can be close to impossible to understand and debug without sufficient formatting and debugging, and doing so isn't an efficient use of our time in office hours. Therefore, **we will only assist you in office hours if your MIPS meets a certain standard of formatting and commenting**. If it doesn't, we'll ask you to format and comment it before we help you. Specifically, as far as formatting is concerned, the first example on the linked page definitely won't fly, the second is iffy, and the third is really what you should be going for. For commenting, since we're translating C to MIPS, it should be immediately clear from your comments what C statement each MIPS instruction is corresponding to; the commenting scheme on the linked page is one way (but not the only one) to achieve this.

Loops and Conditionals and Logic, Oh My! [30 points]

Detecting Parity [10 points]

This function determines if an integer's number of 1 bits is even. It takes one input: an integer `number`, and returns 1 if `number` has an even number of 1 bits (otherwise, returns 0). Similar to `count_ones` from Lab2, this function counts the number of 1 bits in `number`. Then it uses modulus 2 to check if `bits_counted` is even or odd.

The function below is an implemention of detecting parity.

```
int
detect_parity(int number) {
    int bits_counted = 0;
    int return_value = 1;
    for (int i = 0; i < INT_SIZE; i++) {
        int bit = (number >> i) & 1;
        // zero is false, anything else is true
        if (bit) {
            bits_counted++;
        }
    }
    if (bits_counted % 2 != 0) {
        return_value = 0;
    }
    return return_value;
}
```

The C++ code and a test case for `detect_parity()` function are in the file named `lab7.cpp`. You have to write a MIPS translation of this function in `detect_parity.s`. There are some test cases in the assembly file named `detect_parity_main.s`, and you can compile and run the C++ code and compare its output to your MIPS code to verify its correctness. Feel free to add more test cases to both the C++ and `detect_parity_main.s`.

Maximum Continuous Bits in Common [10 points]

This function extracts the maximum number of continous 1 bits that occur in the same indices in two integers. It takes two inputs: an integer `a` and an integer `b`, and returns an integer `max_seen`.

The function below is an implemention of maximum continuous bits in common.

```
int
max_conts_bits_in_common(int a, int b) {
    int bits_seen = 0;
    int max_seen = 0;
    int c = a & b;
    for (int i = 0; i < INT_SIZE; i++) {
        int bit = (c >> i) & 1;
        if (bit) {
            bits_seen++;
        } else {
            if (bits_seen > max_seen) {
                max_seen = bits_seen;
            }
            bits_seen = 0;
        }
    }
    if (bits_seen > max_seen) {
        max_seen = bits_seen;
    }
    return max_seen;
}
```

The function ands the two input numbers together into a new integer `c`, and searches `c` for the longest sequence of continuous 1 bits. It does this search by bit shifting each bit of `c` to the end of the integer (i.e. to the LSB) and checking if it is equal to 1. It uses the integer `bits_seen` as a counter, to keep track of the number of continuous 1 bits seen so far. If `bits_seen` is greater than `max_seen`, it updates `max_seen`. For example, if `a = 8b'10111111, b = 8b'10101111`, then `max_seen = 4`.

The C++ code and a test case for `max_conts_bits_in_common()` are in the file named `lab7.cpp`. You have to write a MIPS translation of this function in `max_conts_bits_in_common.s`. There are some test cases in the assembly file named `max_conts_bits_in_common_main.s`, and you can compile and run the C++ code and compare its output to your MIPS code to verify its correctness. Feel free to add more test cases to both the C++ and `max_conts_bits_in_common_main.s`.

Twisted Sum Array [10 points]

This function sums all values of a 1D array together with a twist: the running sum is first divided by 2 if the mirrored value of `i` in `v` is odd. It takes two inputs: an integer pointer `v` and an integer `length`, and returns an integer `sum`.

The function below is an implemention of twisted sum of an array.

```
int
twisted_sum_array(int *v, int length) {
    int sum = 0;
    for (int i = 0; i < length; i++) {
        if (v[length - 1 - i] & 1) {
            // use sra because sum is a signed number (i.e. don't use srl)
            sum >>= 1;
        }
        sum += v[i];
    }
    return sum;
}
```

The function iterates through the array and keeps a running total of the sum of values in the array. But don't forget the twist! If the mirrored value of a value in the array is odd, the running total is divided by 2. For example:

1. Suppose `v = [4, 3, 2]`. We know that `length = 3`. We begin with `sum = 0`.
2. `i = 0`:
   (a) First extract the mirrored value: `v[length - 1 - i] = v[2] = 2`
   (b) Since the mirrored value, 2, is not odd, we do not divide the running sum by 2.
   (c) Add `v[i]` to running sum: `sum += 4`. Then, `sum = 4`.
3. `i = 1`:
   (a) First extract the mirrored value: `v[length - 1 - i] = v[1] = 3`
   (b) Since the mirrored value, 3, is odd, we **do** divide the running sum by 2: `sum /= 2`. Then, `sum = 2`
   (c) Add `v[i]` to running sum: `sum += 3`. Then, `sum = 5`
4. `i = 2`:
   (a) First extract the mirrored value: `v[length - 1 - i] = v[0] = 4`
   (b) Since the mirrored value, 4, is not odd, we do not divide the running sum by 2.
   (c) Add `v[i]` to running sum: `sum += 2`. Finally, `sum = 7`

The C++ code and a test case for `twisted_sum_array()` are in the file named `lab7.cpp`. You have to write a MIPS translation of this function in `twisted_sum_array.s`. There are some test cases in the assembly file named `twisted_sum_array_main.s`, and you can compile and run the C++ code and compare its output to your MIPS code to verify its correctness. Feel free to add more test cases to both the C++ and `twisted_sum_array_main.s`.

Function Calls & Nested Loops [70 points]

In the second part of this lab, you will learn more about function calls and more complex loops.

Accumulate [35 points]

The first function will accumulate a value into a running total, based on some rules that will use two of the functions you wrote in part 1 of this lab. It takes two inputs: an integer `total` that represents a running total and an integer `value`. It returns an integer `total`.

The function below is an implemention of this accumulation process.

```
int
accumulate(int total, int value) {
    if (max_conts_bits_in_common(total, value) >= 2) {
        total = total | value;
    } else if (detect_parity(value) == 0) {
        total = total + value;
    } else {
        total = total * value;
    }
    return total;
}
```

If `value` and `total` have 2 or more continuous 1 bits in common, you will return the result of `value` and `total` `or`'ed together. Otherwise, if the parity of `value` is even, you will return the result of `value` and `total` `add`'ed together. Finally, if neither of the above conditions are true, you will return the result of `value` and `total` `multiply`'ed together.

Some examples:

- If `total = 8b'10111111, value = 8b'10101111`
  There are 4 continuous 1 bits in common, so we should `or`.
  `8b'10111111 | 8b'10101111 = 8b'10111111`
  Then `total = 8b'10111111`.
- If `total = 8b'00111111, value = 8b'00000000`
  There aren't any continuous 1 bits in common, so we shouldn't `or`.
  The parity of `value` is even, so we should `add`.
  `8b'00111111 + 8b'00000000 = 8b'00111111`.
  Then `total = 8b'00111111`.
- If `total = 8b'00111111, value = 8b'00000001`
  There is only 1 continuous 1 bit in common, so we shouldn't `or`.
  The parity of `value` is odd, so we shouldn't `add`.
  Therefore, we should `multiply`.
  `total = 8b'00111111 × 8b'00000001 = 8b'00111111`.
  Then `total = 8b'00111111`.

Your task is to translate this function to MIPS in `accumulate.s`. Some test cases are provided in

`accumulate_main.s`, and you can compile and run the C++ from `lab7.cpp` and compare its output to your MIPS code to verify its correctness. You're encouraged to add more test cases to both the C++ and `accumulate_main.s`.

Note that `accumulate.s` calls your `detect_parity` function from `detect_parity.s` and your `max_conts_bits_in_common` function from `max_conts_bits_in_common.s`. You should use `jal` to call these functions in your code. **Do NOT inline** `max_conts_bits_in_common` or `detect_parity`. Also, do not include the `max_conts_bits_in_common` or `detect_parity` functions in the `accumulate.s` file. Our tests will ensure that you're actually calling the function.

We will release our own versions of `detect_parity` and `max_conts_bits_in_common` after the first part of the lab is due. We recommend that you copy over our solutions into `detect_parity.s` and `max_conts_bits_in_common.s` files so that you can test `accumulate.s` in the same environment that we will run the autograder.

Calculating Identity [35 points]

The second function you will need to write for Part 2 of this lab will calculate the secret `identity` of a square array by accumulating the values in the array using the function `accumulate` that you wrote previously. It takes in `v` (a 1D array with `size` × `size` values), `size` (the side length of `v`), and returns the integer `identity`.

The function below is an implemention of calculating the secret identity of an array in C++.

```
int turns[4] = {1, 0, -1, 0}; // We take care of declaring this for you
int
calculate_identity(int *v, int size) {
    int dist = size;
    int total = 0;
    int idx = -1;
    turns[1] = size;
    turns[3] = -size;
    while (dist > 0) {
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < dist; j++) {
                idx = idx + turns[i];
                total = accumulate(total, v[idx]);
                v[idx] = total;
            }
            if (i % 2 == 0) {
                dist--;
            }
        }
    }
    return twisted_sum_array(v, size * size);
}
```

Your task is to translate this function to MIPS in `calculate_identity.s`. Some test cases are provided in `calculate_identity_main.s`, and you can compile and run the C++ from `lab7.cpp` and compare its output to your MIPS code to verify its correctness. You're encouraged to add more test cases to both the C++ and `calculate_identity_main.s`.

Note that `calculate_identity.s` calls your `accumulate` function from `accumulate.s` and your `twisted_sum_array` function from `twisted_sum_array.s`. You should use `jal` to call these functions in your code. **Do NOT inline** `accumulate` or `twisted_sum_array`. Also, do not include the `accumulate` or `detect_parity` functions in the `calculate_identity` file. Our tests will ensure that you're actually calling the function.

We will release our own version of `twisted_sum_array` after the first part of the lab is due. We recommend that you copy over our solution into your `twisted_sum_array.s` file so that you can test `calculate_identity.s` in the same environment that we will run the autograder.

Calling Conventions

Remember about calling conventions; treat the functions that you call (e.g. `detect_parity`, `twisted_sum_array`, etc.) as black boxes (*i.e.*, pretend that you don't know how they are implemented), and make sure you're correctly saving and restoring across the function call. We'll test your code against **our own versions** of the called functions to ensure you're following conventions.

Suggestions

Much of the course staff finds this code easier to write using the (callee-saved) `$s` registers! Using `$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. We've posted videos in Piazza about using `$s` registers.

There are also some helper functions for printing out values in `common.s` which you may find helpful for debugging purposes.