MPCS 52010 - Winter 2021
Project 2
Mersenne Primes in MIPS

Due Date: March 8, 2021 @ 6:00pm

Logistics

- In this project you will write a program in MIPS assembly using the QtSPIM emulator. QtSPIM can be downloaded at: `http://spimsimulator.sourceforge.net`. You are not allowed to use MARS or other MIPS emulators, as those applications have different system calls which may prevent the grader from running your code.
- You will need the `mersenne.c` file to begin this project, which is available in the project repository on GitHub Classroom.
- The academic integrity policy for this assignment is as described in the course syllabus.

1 Introduction

The overall goal of this project is to write MIPS code that can find very large prime numbers. Our approach focuses on a specific kind of prime number, known as a “Mersenne Prime”, which has a particular form.

1.1 Mersenne Primes

Mersenne primes are special prime numbers of the form:

$$M_p = 2^p - 1 \tag{1}$$

where M_p is the Mersenne prime, and p is a smaller prime number. Due to the exponential relationship between M_p and p, M_p values can rapidly become astronomically large even for relatively small p values. Not all prime values of p result in a prime M_p, but values of this form have a much higher chance of being prime than a regular number. An important additional aspect of Mersenne primes is that there exists a test for primality for an M_p value that runs in $\mathcal{O}(p)$ time rather than $\mathcal{O}(M_p)$. This means that not only do M_p values have a higher than normal chance of being prime, they can be checked for primality in logarithmic time rather than linear time as would be the case for a regular number.

Due to these useful properties of Mersenne primes, the largest prime numbers we've yet found are all Mersenne primes. The current record for largest prime number is constructed from $p = 82{,}589{,}933$, resulting in an M_p value that is 24,862,048 digits in length.

The algorithm for checking the value M_p for primacy given a prime number p is known as the “Lucas-Lehmer Test” (LLT), and is given as follows:

```c
// Determine if Mp = 2^p - 1 is prime for p > 2
int Lucas_Lehmer_Test(int p)
{
    int s = 4;
    int Mp = pow(2,p) - 1;
    for( int i = 0; i < p-2; i++ )
        s = ((s * s) - 2) % Mp;
    if(s == 0)
        return PRIME;
    else
        return NOT_PRIME;
}
```

1.2 Large Integers

While the LLT algorithm is only a few lines long, there is a great deal of hidden complexity required to implement it. Namely, native integer sizes on most CPUs are only 32 or 64 bits, meaning we can only represent about 2^{64} different integer values. Thus, the largest p value we can test on a modern computer natively is 64. To find larger primes requires doing integer manipulations in software.

Some languages (e.g., python) provide this software ability behind the scenes as part of the language specification. Most languages, (e.g., C) do not, and instead require use of a 3rd party library or user defined functions. In this project you will write the Mersenne prime checker in 32 bit MIPS assembly with no access to any outside libraries, so you will need to handle big integers explicitly.

1.2.1 Large Integer Representation

One method of representing integers is to break them up into an array of smaller chunks. An example of this is to simply represent the integer as a string that can be any length, with each base 10 digit in the integer being represented by a single character in the string. In C, we might store the integer value “10,000,004,300” in big endian string form as follows:

```c
char my_big_int[] = "10000004300";
```

Another more efficient method is to store the big integer as an array of higher base integer "chunks", with the base being as high as possible while still allowing for integer arithmetic to be performed on the chunks using 32 or 64 bit integer operations without overflowing. For instance, if we were using a 32 bit system, the maximum signed integer value is 2,147,483,647, so we might pick our chunk integers to be of base 10,000. This would allow for maximum chunk values of 9,999, which when squared would be $9999^2 = 99,980,001$, which is below the maximum value for a signed 32 bit integer so would not overflow. In C, we might store the integer value "10,000,004,300" in big endian chunk form as follows:

```c
int my_big_int[] = {100, 0, 4300};
```

For the purposes of this assignment, in order to maximize code simplicity we will store our values using an array of integer chunks as above, but use base 10 rather than base 10,000. This is not particularly efficient, but can help to minimize the complexity of the code we will be writing. Additionally, we will store our value in little endian format, where the least significant values are stored first, which will make doing many integer operations (subtraction, multiply, modulus, etc) a little simpler. In C, we would therefore store the integer value "10,000,004,300" in little endian base 10 chunk form as follows:

```c
int my_big_int[] = {0, 0, 3, 4, 0, 0, 0, 0, 0, 0, 1};
```

1.2.2 Large Integer Arithmetic

With the data format defined, the question remains how to perform arithmetic operations using these values. There are a large number of options, some more efficient than others, but for this project we will try to use the simplest possible "grade school" arithmetic methods while still allowing the code to be fast enough to handle numbers larger than what we could do with 32 or 64 bit integer arithmetic alone.

Such grade school arithmetic algorithms typically involve processing each digit, starting from low to high importance, and carrying over overflow values to the next digit. We can visualize this process for elementary digit-wise addition as below:

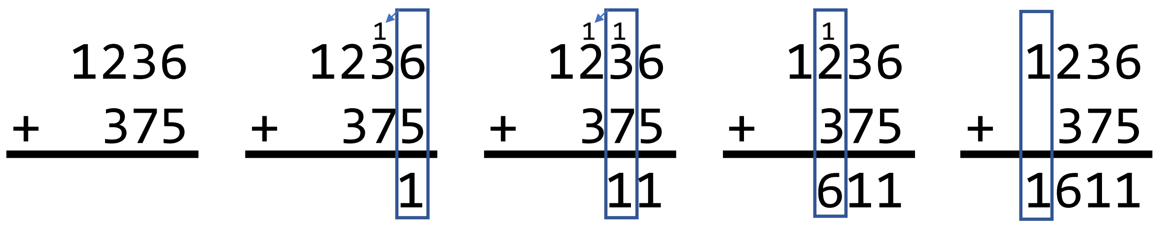

Figure 1: Example of an elementary base 10 addition.

For the purposes of finding large prime numbers, we don't actually need to be able to perform all different types of arithmetic, so we will limit ourselves to the following big integer arithmetic operations:

- Subtraction ($A - B$)
- Multiplication ($A \times B$)
- Powers (A^n)
- Modulus ($A\%B$)
- Division by 10 ($A/10$)
- Comparison ($A < B$, $A == B$, $A > B$)

We have provided you with (almost) all of the algorithms you'll need to accomplish the above arithmetic operations for big integers in a high level language. See the `mersenne.c` file at `https://github.com/jtramm/mersenne_starter` for more details.

1.3 MIPS

The "Microprocessor without Interlocked Pipelined Stages" (MIPS) architecture is an instruction set architecture that defines a set of registers and instructions that form a computer. By writing assembly code composed of these instructions, you can define a program. You will be running your code not on an actual MIPS processor, but rather on the QtSPIM MIPS emulator.

For more background on MIPs, here are some good resources:

- Chapter 2 of Patterson & Hennessy
- MIPS and SPIM Handbook: `http://pages.cs.wisc.edu/~larus/HP_AppA.pdf`
- Instruction set cheat sheet: `http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm`
- QtSPIM system calls: `https://www.doc.ic.ac.uk/lab/secondyear/spim/node8.html`

2 Project: Mersenne in MIPS

Your task for this project is to implement a program in MIPS that runs in QtSPIM that checks all prime numbers $3 <= p <= 128$ to see if they result in a Mersenne prime M_p or not. Your code should use the LLT algorithm and allow for big integers of arbitrary length.

2.1 Finishing the High Level Implementation (10 points)

Your first task will be to fill in the missing functions in `mersenne.c`. The missing functions are as follows:

- `sub_big`: This function subtracts two big integers, and returns the result. I.e., $c = a - b$. You should use the simplest possible "elementary school" algorithm to make this work. Assume $a >= b$, and that both a and b are positive.
- `shift_left`: This function divides a big integer value by 10. Since our numbers are stored in base 10, this means you will just need to shift your values to the left by one slot. Assume the input value a is $a >= 10$. Note, you simply drop any fractional part of the value that may result, e.g. it is like floored integer division in C and other languages `3 / 2 = 1`.

The goal for this portion of the assignment is to ensure that you understand the big integer data structure representation before attempting to implement anything in assembly.

We will also allow for an additional assumption (for convenience) that big integers in our program will have at most 350 digits, and will not be negative. This assumption holds for both the high level representation, as well as your low level MIPS implementation described in the next section. You could easily remove this assumption by using dynamic memory management to allocate more space for big integer values as they grow larger, but to keep things as simple as possible we will just assume a fixed maximum length of 350 digits.

Once you complete your functions, compile and run the code and ensure it is working correctly. Include your completed `mersenne.c` file in your submission.

2.2 Implementing in MIPS

Next, port your completed `mersenne.c` code to MIPS. You have freedom to change the interfaces or data representations to make things easier for you to implement in MIPS as you see fit. However, there are the following restrictions and guidelines:

- You must adhere to MIPS caller/callee conventions for handling "saved" register values inside subroutines.

- You should not have any large memory leaks in your code. For instance, you should not be allocating memory with the `sbrk` service inside of commonly called functions like `sub_big`, which may be called millions of times in your code. Instead, you should use the stack or allocate a limited number of big integer variables in your data segment. If using the stack, you will still need to make sure you do not have a memory leak by popping any allocated items off at the end of the subroutine.

- While verbose comments in high level languages are a good idea, they are absolutely required when writing assembly. Typical assembly programs will have a comment explaining every single instruction. For this project, you are required to write an accurate comment on most, if not all, of the lines in your assembly code. Your comments can feature pseudo-code and/or a simple explanation of which variables and operations are being used. Feel free to add in paragraph comments as well to explain what blocks of instructions do. **Failure to adequately comment your code may result in loss of credit.** Your code must be human readable, clean, and you must make it so that the grader can reasonably understand your code.

- You are free to alter the interfaces from the C code when implementing things in MIPS. For instance, the `sub_big` and `mult_big` functions in the C code are pass by value, and return the value of c in $c = a * b$ to the caller. In the LLT algorithm, we find that these functions are only ever used in the manner of $a = a * b$, so you may want to simplify the interface to pass by reference and overwrite the value of a.

- You are encouraged to use QtSPIM for debugging your code. Setting breakpoints and stepping through the execution will allow you to monitor the values of registers to more easily identify bugs.

- You are not allowed to use any automated C to MIPS conversion utilities or compilers − you must write MIPS by hand, and your code must run on QtSPIM. Your code must be written with a logical "human readable" structure and must be well commented, so use of such utilities will not be helpful in any case. MIPS code that appears to have not been hand written will receive zero credit.

- Your implementations of the big integer arithmetic functions do not necessarily need to handle all edge cases. Your subroutines only need to provide the functionality required by the LLT algorithm to compute primes.

- Unlike the C code which by default considers primes up to $p = 547$, your MIPS code only needs to run to $p = 127$.

2.3 Output Requirements and Grading

The overall goal of this project is to write a MIPS program to find large Mersenne prime numbers. To encourage you to test and debug your code along the way, and to allow partial credit to be given even if you are not able to fuly complete the project, a number of small "waypoint" tests are defined below. Table 1 shows what tests need to be run for each function, and how much credit each function is worth. The testing regime specified here is not exhaustive, so you are encouraged to develop additional tests when developing your code. However, limit the output of your final code submission to the following tests, so that the grader knows what to expect and which of your functions are working correctly. Note that the big integer initialization function which appears in `mersenne.c` is not required to be implemented in your MIPS program if you don't want, as initialization may be better done in an in-line manner in MIPS.

As some of the arithmetic functions are dependent on others (e.g., the power function is dependent on the multiplication function), you are advised to start implementing the lower level functions without dependencies first so that you can fully test them and get full credit for them before moving on. Functions that are implemented, but cannot pass their tests due to dependent functions not working may not receive very much partial credit. Therefore, thoroughly test your lower level functions first (e.g., subtraction, multiplication) before starting on higher level ones (e.g., modulus, LLT).

Points	Function Name	Tests
5	is_small_prime	- Test regular int 7 for primacy, output 1 if prime, 0 otherwise (expected value: 1) - Test regular int 81 for primacy, output 1 if prime, 0 otherwise (expected value: 0) - Test regular int 127 for primacy, output 1 if prime, 0 otherwise (expected value: 1)
5	print_big	- none
5	compress	- Initialize a big int value of 0003 (size 4), call compress, and print (expected output: 3)
5	shift_right	- Initialize a big int value of 3, shift right 3 times, and print (expected output: 3000)
5	shift_left	- Initialize a big int value of 7000, shift left 2 times, and print (expected output: 70)
5	compare_big	- Compare big ints 42 and 30, print return code (-1, 0, or 1) (expected output: 1) - Compare big ints 30 and 42, print return code (-1, 0, or 1) (expected output: -1) - Compare big ints 42 and 42, print return code (-1, 0, or 1) (expected output: 0)
10	mult_big	- Multiply big ints 3 and 7 (expected output: 21) - Multiply big ints 30 and 42 (expected output: 1260) - Multiply big ints 10,000,000 and 9,000,000 (expected output: 90000000000000)
5	pow_big	- Find 3^4 (expected output: 81) - Find 42^{42} (expected output: 150130937545296572356771972164254457814047970568738777235893533016064)
10	sub_big	- Subtract big ints 7 and 3 (expected output: 4) - Subtract big ints 42 and 12 (expected output: 30) - Subtract big ints 9000000000 and 7654321 (expected output: 8992345679)
10	mod_big	- Find 7 % 3 (expected output: 1) - Find 48 % 12 (expected output: 0) - Find 9000000000 % 7654321 (expected output: 6172825)
5	LLT	- Test p = 11 for M_p primacy, output 1 if prime, 0 otherwise (expected value: 0) - Test p = 61 for M_p primacy, output 1 if prime, 0 otherwise (expected value: 1)
5	mersenne_scan	- Test all prime values p from $3 <= p <= 128$ for M_p primacy, output text results

Table 1: Grading table showing what tests your code needs to perform.

The expected output for your MIPS assembly code when run with QtSPIM should look as in the example printout below. It's OK if there are small formatting changes, but please make sure the tests are of the right type and in the right order so as to make things easy on the grader.

```
Small Prime Tests
1
0
1
Compress Tests
3
Shift Right Test
3000
Shift Left Test
70
Comparison Tests
1
-1
0
Multiply Tests
21
1260
90000000000000
Power Tests
81
150130937545296572356771972164254457814047970568738777235893533016064
```

```
Subtraction Tests
4
30
8992345679
Modulus Tests
1
0
6172825
LLT Tests
0
1
Mersenne Scan
Testing p = 3 found prime Mp = 7
Testing p = 5 found prime Mp = 31
Testing p = 7 found prime Mp = 127
Testing p = 11 Mp not prime
Testing p = 13 found prime Mp = 8191
Testing p = 17 found prime Mp = 131071
Testing p = 19 found prime Mp = 524287
Testing p = 23 Mp not prime
Testing p = 29 Mp not prime
Testing p = 31 found prime Mp = 2147483647
Testing p = 37 Mp not prime
Testing p = 41 Mp not prime
Testing p = 43 Mp not prime
Testing p = 47 Mp not prime
Testing p = 53 Mp not prime
Testing p = 59 Mp not prime
Testing p = 61 found prime Mp = 2305843009213693951
Testing p = 67 Mp not prime
Testing p = 71 Mp not prime
Testing p = 73 Mp not prime
Testing p = 79 Mp not prime
Testing p = 83 Mp not prime
Testing p = 89 found prime Mp = 618970019642690137449562111
Testing p = 97 Mp not prime
Testing p = 101 Mp not prime
Testing p = 103 Mp not prime
Testing p = 107 found prime Mp = 162259276829213363391578010288127
Testing p = 109 Mp not prime
Testing p = 113 Mp not prime
Testing p = 127 found prime Mp = 170141183460469231731687303715884105727
```

2.4 Code Cleanliness, Commenting, Structure, & Coding Habits (15 points)

- Your submission should include a README that has your name in it, and that documents any deficiencies your code may have. I.e., in the grading table above, which functions are working and passing tests, and which are not? If there are functional dependencies preventing other functions from working, please make this clear. For example, if your multiply function doesn't work, then your power function won't work either. Explain if you think this is because there is an issue only in the multiply function, or if there is also likely an issue in the power function as well. The more documentation you have, the more partial credit is likely to be awarded.

- Your submission should also include an "output.txt" file that contains the terminal output from your final MIPS code when run through QtSPIM.

- Your MIPS code should be human readable, which means it should be very clean, well commented, well structured, and easy for the grader to understand. We are not going to deduct credit for small or occasional comment deficiencies, and are not using a rigid coding standard, but you need to make your code reasonably easy for the grader to follow. The majority of lines in your code should be commented. Messy and/or poorly commented code that is unreasonably difficult to follow may receive deductions even if it is logically correct.
- Your submission must include the following items:
  1. README
  2. Completed high level C implementation (`mersenne.c`)
  3. Makefile for your `mersenne.c` code
  4. MIPS implementation (mersenne.asm)
  5. QtSPIM terminal output (output.txt).