Computer Architecture Final Project

Kevin Blum

May 2, 2017

1 Introduction

The project that I would like to code is a mix of assembly and C++ linked together to create a compendium of mathematical formulas and algorithms such as the Collatz conjecture, the Sieve of Eratosthenes, and Montgomery Multiplication. Most of these algorithms are related to cryptography which is why I think it is important because they are integral to computing in general and security and because they are purely mathematical in nature I think assembly is the proper fit to write these programs as it should be theoretically more efficient than a high-level language. The implementations of each formula will be in assembly and C++ will be used for user input and data output. The main loop of the program will be a menu for each algorithm to choose from and program flow will be controlled by the assembly code.

2 Algorithms

2.1 Collatz Conjecture

The Collatz conjecture, also called the half or triple plus one conjecture, is an interesting formula because it is a deceptively simple formula but it is a yet unproven theorem. The implementation of this conjecture is not necessarily difficult but I think it would be a nice challenge to find the most efficient and fastest implementation. The efficiency would be interesting to test based on the increasing size of the input values and whether the speed of implementation is slowed.

2.2 Sieve of Eratosthenes

The Sieve of Eratosthenes is a unique algorithm that calculates all the prime numbers below the input value. The importance of knowing prime numbers can be valuable in prime factorization which is heavily used in modern cryptography as well as many other applications because prime numbers are such a special type of number in mathematics. The formula itself is not incredibly complex but could still provide a challenge as the number of potential primes is not known beforehand and thus must be dynamically stored somehow and this algorithm is similar to the Collatz conjecture that it may be simple in theory but I think a good challenge would be to find the most efficient and fastest implementation and test its ability to compute higher input values and tell whether the implementation is significantly slower because of those high input values.

2.3 Montgomery Multiplication

Montgomery Modular Multiplication is a method for quick modular multiplication that is used in cryptography quite heavily, especially in RSA. This specific algorithm is intended to be fast especially for large values but it is also much more complex than the other algorithms and therefore a lot of the challenge from this algorithm will be its complexity and the potentially high difficulty to implement in assembly as it difficult to gauge. The algorithm again is centered around speed and efficiency so it would definitely be interesting and challenging to compare the speed and efficiency of the Montgomery multiplication to tradition modular multiplication, granted that Montgomery is meant to be much faster but it would be interesting to find out by how much.

3 Implementations

3.1 Collatz Conjecture

There are two implementations to the collatz conjecture, one in Assembly MASM 32-bit and the other C++. The hope was to utilize 64-bit for a larger space for values and the 64-bit implementation was successful for Collatz but could not be used because 64-bit caused too much complication for the Sieve of Eratosthenes and Montgomery Reduction implementations. Below is the implementation of Collatz in both assembly and C++.

```
begin:
cmp collatz, 1
jz endCollatz
test collatz, 1
jnp oddCollatz

evenCollatz:
inc collatzCounter
mov eax, collatz
div ebx
mov collatz, eax
jmp begin

oddCollatz:
inc collatzCounter
imul ecx, collatz, 3
mov collatz, ecx
inc collatz
jmp begin

endInvalid:
lea eax, collatzInvalid
push eax
call _printString
add esp, 4
jmp endProgram
```

Figure 1: Main algorithm of Collatz in Assembly

```
while (collatz != 1) {

    if (collatz % 2 == 0) {
        collatz = collatz / 2;
        collatzCounter++;
    }
    else {
        collatz = 3 * collatz + 1;
        collatzCounter++;
    }
}
```

Figure 2: Main algorithm of Collatz in C++

It can be determined that the algorithm is quite simple and both look similar despite being in syntactically different languages. There are more commands in the assembly but does not necessarily imply the implementation is slower as often with most inputs complete in the same time frame, which may be a result of the simplicity of the algorithm itself. The algorithm itself though works by comparing the number with one and jumping to the end if it is zero otherwise

testing the collatz number with one. Which will either jump to the odd segment where it is tripled and incremented otherwise it just continues to the even segment where the number is halved and eventually in the begin segment the number will reach one where the algorithm will end. The C++ implementation accomplishes the same task in a similar fashion but instead just tests if the collatz number modulus two results in zero to test if it is even, otherwise it is odd.

3.2 Sieve of Eratosthenes

There are also two implementations of the Sieve of Eratosthenes, one in MASM 32-bit and the other in C++. This algorithm was giving a lot of trouble within 64-bit because it was difficult to implement a version of the algorithm that allowed for a user defined variable size array and even with a predefined array size it gave many errors for array manipulation with the stack that could not be resolved. The resulting implementation in assembly is 32-bit and a predefined array size, which is not the desired result, but it accomplishes the Sieve of Eratosthenes on the given number but it is not nearly as efficient as one would hope. The implementation could not be optimized so effectively which is disappointing because at large values the C++ implementation performs the algorithm quite fast while the assembly implementation is quite slow. This is most likely due to the number of array manipulations which slows down the algorithm but optimization attempts resulted in unwanted and unexpected errors that could not be resolved. The Assembly and C++ implementations of the main algorithm are below.

The Assembly algorithm works by populating a DWORD array of size 1001 where the prime candidate is array size - 1, which is 100. It begins at two because that is the first prime above one and so the Sieve begins there and then once it populates the array it runs through the three different loops. The first loop is the main loop where it compares the index of the array and if it is not even it will pass it to the second loop otherwise will increment and keep iterating. The second loop compares the value and passes it to the third loop if the value is not even where the third loop tests if the value is a prime and if it is not then it reduces the value to zero meaning that it will not be included in the final array of primes. The algorithm then just iterates through all these loops until the end of the array is reached and then just outputs the array of primes. The C++ is more elegant as it is uses more complex data structures than the assembly implementation and also functions much more efficiently than the assembly version. The

3.3 Montgomery Reduction

Unfortunately, an implementation for the Montgomery Reduction could not be accomplished. An implementation for C++ was completed with some errors in which not all proper cases of the algorithm worked and the assembly implementation for the actual algorithm itself proved to be too complex and required too much high level functions and assistance that it essentially did not accomplish the work of the algorithm in assembly, defeating the purpose of the goal. I did not include the implementation of my code because it was incomplete but will include reference to the resources of the Montgomery reduction code.

4 Reference

Referenced works and resources that were utilized in completing this project. http://www.geeksforgeeks.org/sieve-of-eratosthenes/ https://gist.github.com/johnnykv/960732/7b2d7fb6a6c404f92
https://rosettacode.org/wiki/Sieve_of_Eratosthenes http://stackoverflow.com/questions/40926318/sieve-of-eratosthenes-x86-assembly https://rosettacode.org/wiki/Montgomery_reduction http://stackoverflow.com/questions/20111827/various-questions-about-rsa-encryption/20114154#20114154

```
clearReg
populateArray:
    mov eax, ecx
    add eax, 2
    mov [primeArray+4*ecx], eax
    inc ecx
    cmp eax, arraySize
    jnz populateArray

finishArray:
clearReg
partOne:
    mov ebx, ecx
    inc ebx
    cmp [primeArray+4*ecx], 0
    jne partTwo
    resumeOne:
    inc ecx
    cmp ecx, arraySize
    jb partOne
    jmp outputPrime

partTwo:
    cmp [primeArray+4*ebx], 0
    jne partThree
    resumeTwo:
    inc ebx
    cmp ebx, arraySize
    jb partTwo
    jmp resumeOne

partThree:
    xor edx, edx
    xor eax, eax
    mov eax, [primeArray+4*ebx]
    div [primeArray+4*ecx]
    cmp edx, 0
    je removeValue
    jmp resumeTwo
```

Figure 3: Main algorithm of Sieve of Eratosthenes in Assembly

```
char *prime = new char[n + 1];
int p = 2;
std::fill_n(prime, n, true);

for (p; p*p <= n; p++)
{
    if (prime[p])
    {
        for (int i = p*p; i <= n; i += p)
            prime[i] = false;
    }
}
_clockEnd();

for (p = p + (1 - (p % 2)); p <= n; p += 2) {
    if (prime[p])
    std::cout << p << " ";
}
```

Figure 4: Main algorithm of Sieve of Eratosthenes in C++