ECE 375 FINAL PROJECT

Written Report

Marth 14th, 2021

Alex Young

COMPUTING SATELLITE DETAILS

My program has specific functions that will multiply, divide, and square root large inputs (with the maximum number of bits below). In order to compute satellite details, these functions are called in a specific order that correlates to the physics problem that needs to be solved.

For starters, my program will first read the GM value and radius value from program memory and store them. Then in order to first compute the satellite velocity the program will start by dividing the GM by the radius (This would be stored into the Quotient location in memory). Next, the program will square root the division result to compute the velocity, which will be stored in memory.

For computing the period of the satellite's revolution, the program starts by multiplying 40 and the radius. Then the program will multiply the result of the multiplication with the radius twice (This result would be stored in the Product location in memory). Next, the program will divide the multiplication result by the GM. Finally, the program will square root the result to compute the period, which will be stored in memory.

To make the implementation process clearer, when dividing x by y, y has to first be stored in DIV_OP1 (the first operand for the division function), and x has to first be stored in DIV_OP2 (the second operand for the division function), before calling the DIV subroutine, and the result of the division will appear in DIV_RESULT in memory.

Similarly, for multiplying values x and y together, x has to first be stored in MUL_OP1 (the first operand for the multiply function), and y has to first be stored in MUL_OP2 (the second operand for the multiply function), before calling the MULT subroutine, and the result of the multiplication will appear in MUL_RESULT in memory.

Finally, for square rooting an input x, x first must be stored in SQRT_OP, before calling the SQRT subroutine, and the result of the square root will appear in SQRT_RESULT in memory.

MAXIMUM NUMBER OF BITS EACH ARITHMETIC FUNCTION HANDLES

ADD – Inputs two 48-bit values, outputs a 54-bit result where the high byte contains the carry out bit

SUBTRACT – Inputs two 64-bit values, outputs a 72-bit result where the high byte contains the carry bit

MULTIPLY – Inputs two 48-bit values, outputs a 96-bit result

DIVIDE – Inputs two 64-bit values, outputs a 48-bit result

SQUARE ROOT – Inputs a 48-bit value, outputs a 48-bit result

To make this clear, the maximum number of bits that each arithmetic function can handle is the same as their input values listed above.

SQUARE ROOT ALGORITHM

I calculated the square root by starting from 0, squaring it, comparing it to the input, and then incrementing and repeating. To start, I would increment by 0x0100 after every compare. Once the square passed the input, I would subtract 0x0100 from my incremented value and repeat again but with 0x0001 as the new amount I increment by. Once the square of my incremented value reaches the input, I subtract one from the incremented value (because we always are rounding down) and that final value is the square root result.

PRIMARY CHALLENGES

One of the primary challenges of this assignment was the difficulty it took to debug my code and make sure my results were correct. I had to constantly be checking what values were being stored in program memory and it could get confusing sometimes because I had so many operands and results being stored. For this, setting breakpoints and stepping through code helped a lot to find the rogue bugs like a forgetting to increment registers or using lpm instead of ld. In a similar subject, it took a long time to check my results because it actually took a bit of time to solve for the correct answers with a hexadecimal calculator. To be honest, this made the project more challenging because it was just so hard to debug and run through test cases. When implementing a similar project in the future it would be very useful to code a python or c program to print out test results.

Another primary challenge was implementing variable division and square root functions that would increment larger numbers at a time. I used these functions to improve the runtime of the division and square root algorithms. This was a larger challenge for me because in order to implement these new functions I had to add a lot of other functions and variables, as well as store and access a lot more information in memory. It was challenging to keep track of the large amount of extra information.

LONGEST FEATURE TO IMPLEMENT

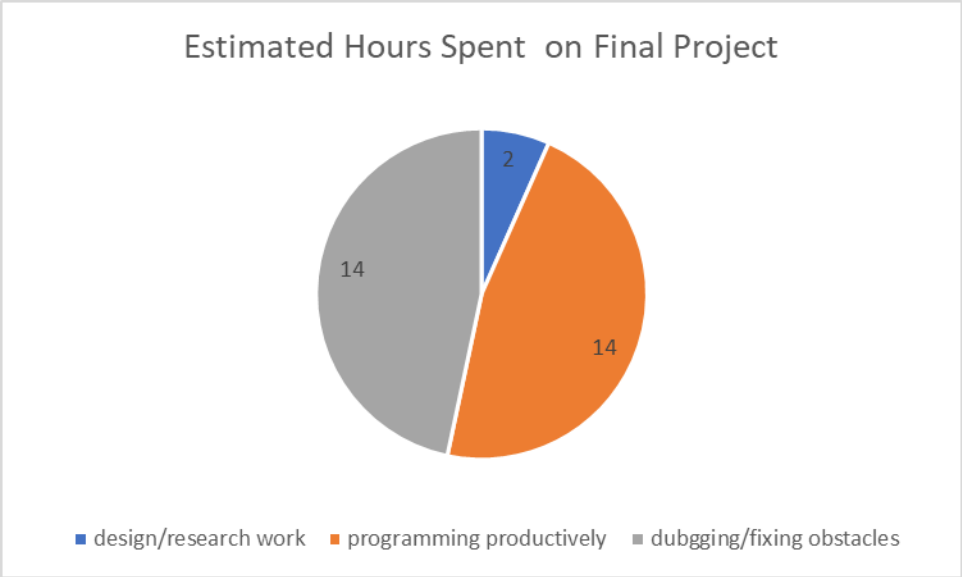
The feature that took the longest to implement was reducing the cycle counter that the DIV function would take. Due to my lack of foresight and poor planning, I originally coded my DIV function in the exact manner that the professor gave as an example. I had code working for DIV that would subtract the divisor and increment the quotient one at a time, until I reached 0 or a negative number. I found through testing that this would take so long to solve large inputs that it was effectively freezing my laptop. Instead of doing any intensive research or extra planning, I thought this would be a simple fix if I multiplied the divisor by a constant and incrementing the quotient by the same constant, and then did the same with smaller constants until I was incrementing by 1 and got the correct answer. While in theory this seemed like a logical and simple plan to me, it took a very long time to implement due to various bugs. Also, doing this required me to change the way I was storing my quotient as well as include an ADD function to increment by a variable amount.

POTENTIAL DESIGN CHANGES

If I were to re-implement this project, I would probably design my code differently to read in a 9.8696 as pi. This would be an interesting challenge because I am actually not that sure where to start to implement this quickly. I am thinking that maybe I could brute force it by taking the multiplication result, multiplying it by 1000, and essentially getting rid of the extra 0.1404 and dividing back by 1000 but it still sounds tricky.

Interestingly enough, when I was writing this, I originally noted that I would like to add design changes that would allow for a larger radius to be inputted (the other extra credit), when I realized that my program already can handle a larger radius, so I instead quickly added some code to check the size of OrbitalRadius before storing the radius into memory – either as a 16-bit input or 24-bit input.

Time Spent Estimation



This pie chart is an estimation of how much time I spent working on this project. However, I feel like the hours I listed here are not entirely accurately depicting how I spent my time because I wasn't sure what might count as "fixing unexpected obstacles" and "programming productively". For example, I spent a large amount of time changing my code to take less clock cycles after I realized my division and squareroot were taking too many cycles. This was both an obstacle that I had to fix because of poor planning and productive programming because I was improving my code. In this process I was constantly switching between testing my code, debugging, and writing new code. Overall, I do think it is pretty accurate that I spent about 2 hours planning, and a spent around total of 30 hours working on the entire project (although I was losing focus and taking breaks at some points so it is hard to say for sure).

Extra Credit Work

I implemented the extra credit for allowing a maximum orbital radius of 120,000 km. This means that the orbital radius can have a maximum little-endian hex value of C0 D4 01. Because my original code already worked for division and multiplication of over 24-bits, the only thing I had to change for this extra credit work was my STORE_RADIUS subroutine. Originally, this subroutine stored the 16-bit radius from program data to data memory. For the extra credit I had the subroutine compare the address of (OrbitalRadius + 2) to the address of (SelectedPlanet). If these addresses were equal, that means that the input is only 16-bits and I would store the 16-bit radius into memory, and if it isn't equal, then I would read in 24-bits instead to memory as the radius.