ECE 375 Final Project

By Alexander Uong

Introduction

In this final project, we were tasked with implementing an AVR assembly program that computes two equations and stores the results into data memory. The first equation we are tasked with calculating is the velocity of an artificial satellite orbiting an object in space given by the equation $v = \sqrt{\frac{GM}{r}}$. GM is a 32-bit value that is the standard gravitational parameter of one of the nine planets provided. R is a 16-bit value that is the orbital radius of the satellite. The second problem we are tasked with solving is calculating the period of revolution for the satellite given the equation $T = \sqrt{\frac{4\pi^2 r^3}{GM}}$. Each of the variables will be retrieved out of program memory, with the results being stored in data memory.

Project Overview

Regarding my implementation of this final project, I made subroutines for each of the steps it would take for me to calculate the velocity and period of the given satellite.

Prior to calling any subroutines, I obtain the values provided in program memory and store them into registers in order to begin my calculations. In order to find the velocity, GM values are provided in program memory, but there are nine different planets. In order to obtain the correct planet's GM value, I read from the value "SelectedPlanet" and treat that value as the index of PlanetInfo, which is an array of planets. I then read from the array of planets provided in PlanetInfo, fetching the correct planet's GM value. Along with this, the 16 bit orbital radius value is provided and fetched out of program memory. The stored program data can be seen here:

```
;***********************************************************
;*  Stored program data that you cannot change
;***********************************************************

; Contents of program memory will be changed during testing
; The label names (OrbitalRadius, SelectedPlanet, PlanetInfo, MercuryGM, etc) are not changed
; NOTE: All values are provided using the little-endian convention.
OrbitalRadius:  .DB  0xe9, 0x03 ; the radius that should be used during computations (in kilometers)    0x64, 0x19 0xe9, 0x03
                                ; in this example, the value is 6,500 kilometers
                                ; the radius will be provided as a 16 bit unsigned value (unless you are
                                ; completing the extra credit, in which case the radius is an unsigned 24 bit value)

SelectedPlanet: .DB 0x02, 0x00  ; This is how your program knows which GM value should be used.
                                ; SelectedPlanet is an unsigned 8 bit value that provides you with the
                                ; index of the planet (and hence, tells you which GM value to use).
                                ; Note: only the first byte is used. The second byte is just for padding.
                                ; In this example, the value is 2. If we check the planet at index 2, (from the data below)
                                ; that corresponds to Earth.
                                ; if the value was 7, that would correspond to the planet Neptune

PlanetInfo:                     ; Note that these values will be changed during testing!
MercuryGM:      .DB 0x0E, 0x56, 0x00, 0x00  ; Gravitational parameters will be provided as unsigned 32 bit integers (little-endian)
VenusGM:        .DB 0x24, 0xF5, 0x04, 0x00  ; the units are in: (km * km * km)/(sec * sec)
EarthGM:        .DB 0x08, 0x15, 0x06, 0x00  ; <-- note that this is 398,600          0x08, 0x15, 0x06, 0x00
MarsGM:         .DB 0x4E, 0xA7, 0x00, 0x00
JupiterGM:      .DB 0x30, 0x13, 0x8D, 0x07  ; A word of advice... treat these like an array, where each element
SaturnGM:       .DB 0xF8, 0xC7, 0x42, 0x02  ; occupies 4 bytes of memory.
UranusGM:       .DB 0xD0, 0x68, 0x58, 0x00  ; Mercury is at index 0, Venus is at index 1, ...and the final planet is at index 8.
NeptuneGM:      .DB 0x38, 0x4B, 0x68, 0x00
FinalGM:        .DB 0xFF, 0xFF, 0xFF, 0xFF
```

It is also important to note and be aware of the four computations that will be stored in data memory. The quotient, which is $\frac{GM}{R}$ in the velocity equation, will be stored. The final velocity value, which is seen by $v = \sqrt{\frac{GM}{r}}$, will be stored. The product, which is $4\pi^2 r^3$ in the period of revolution equation, will be stored. Lastly, the final period value which is seen by $T = \sqrt{\frac{4\pi^2 r^3}{GM}}$, will be stored. Below is a picture of the allocation of the computations in data memory, where these following results will be stored.

```
;***********************************************************
;*  Data Memory Allocation for Results
;*  Your answers need to be stored into these locations (using little-endian representation)
;*  These exact variable names will be used when testing your code!
;***********************************************************
.dseg
.org    $0E00                   ; data memory allocation for results - Your grader only checks $0E00 - $0E14
Quotient:       .byte 3         ; This is the intermediate value that is generated while you are computing the satellite's velocity.
                                ; It is a 24 bit unsigned value.
Velocity:       .byte 2         ; This is where you will store the computed velocity. It is a 16 bit signed number.
                                ; The velocity value is normally positive, but it can also be -1 or -2 in case of error
                                ; (see "Special Cases" in the assignment documentation).
Product:        .byte 7         ; This is the intermediate product that is generated while you are computing the orbital period.
Period:         .byte 3         ; This is where the orbital period of the satellite will be placed.
                                ; It is a 24 bit signed value.
                                ; The period value is normally positive, but it can also be -1 or -2 in case of error
                                ; (see "Special Cases" in the assignment documentation).

;***********************************************************
;*  Additional Program Includes
;***********************************************************
; There are no additional file includes for this program
```

Error Cases

There are also four error cases that should be discussed. This includes:

- The orbital radius (r) being ≤1000
- The computed velocity of the satellite (v) equaling 0
- The standard gravitational parameter (GM) being ≤1000
- The period of revolution (T) being < 25 seconds

Handling the Error Cases

In the case of the orbital radius (r) being ≤1000, 0xFF 0xFF, the 16 bit signed value of -1 in little endian notation, will be stored into “Velocity” in data memory.

In the case of the computed velocity of the satellite (v) equaling 0, 0xFE 0xFF, the 16 bit signed value of -2 in little endian notation, will be stored into “Velocity” in data memory.

In the case of the standard gravitational parameter (GM) being ≤1000, 0xFF 0xFF 0xFF, the 24 bit signed value of -1 in little endian notation, will be stored into “Period” in data memory.

In the case of the period of revolution (T) being < 25 seconds, 0xFE 0xFF 0xFF, the 24 bit signed value of -2 in little endian notation, will be stored into “Period” in data memory.

Subroutines and Steps

In terms of subroutines and steps taken to complete this project, I made subroutines based on the sequential steps I felt were needed in order to efficiently obtain the values and calculate the results that are stored in data memory. As mentioned above, this is the quotient, velocity, product, and period.The following subroutines described below are executed in this exact order in my program.

Subroutine #1: CheckRadius

After obtaining the orbital radius from program memory, the first step is to check if the orbital radius is ≤1000. This was done by assigning the value 0x03E9 to two registers, which is equivalent to 1001, and comparing these two registers to the two registers that are storing the value of the orbital radius. If the value of the orbital radius stored in the two registers is less than 0x03E9, 0xFF 0xFF is stored into “Velocity” in data memory.

Subroutine #2: getGM

This routine focused on obtaining the GM value of the correct planet stored in program memory. The value of selectedPlanet is stored into a register and used to correctly fetch the correct planet out of the planetInfo list, which I treat as an array. The Z register is used to point at the address of the planetInfo array in program memory. If the value of selectedPlanet is > 0, 4 bytes are added to the z register to move to the address of the next planet in planetInfo; we know adding 4 bytes moves to the next planet in program memory, as each planet's GM value is 4 bytes. Each time this is done, the value of selectedPlanet is decremented by one. Once the Z register is pointing to the correct planet, the GM value is loaded into four separate registers.

Subroutine #3: checkGM

The second error handling case is handled in this function, where if the standard gravitational parameter (GM) is ≤1000. This is very similar to CheckRadius, where the value 0x03E9 (1001 in decimal) is stored into two registers. If the GM Value of the planet stored in the four registers is less than 0x03E9, 0xFF 0xFF 0xFF, the 24 bit signed value of -1 in little endian notation, is stored into "Period" in data memory.

Subroutine #4: calculateQuotient

This routine calculates the quotient, which is is $\frac{GM}{r}$in the velocity equation $v = \sqrt{\frac{GM}{r}}$. This follows the approach shown on the assignment page, where r is subtracted from GM until it no longer can be. Each time r is able to be subtracted from GM (i.e. GM is still greater than r), a register serving as a counter is incremented. This essentially serves as division, as division is the amount of times the divisor can go into the dividend.

However, in the case of division in this project, we're asked to round up or down based on the remainder. To approach this, the leftover value of GM is added to itself or multiplied by 2. If the value is greater than the orbital radius r, we know GM is over half of the orbital radius, meaning the quotient should be rounded up (adding one to the counter). If it's not, the quotient is not rounded up. The counter is then stored into "Quotient" in data memory. The screenshot below explains the logic behind the division implemented in this subroutine:

- In order to divide two numbers, you can simply keep subtracting the divisor, until the number is too small to subtract further. This is similar to the division approach that is sometimes taught to children. For example, "How many times does 5 go into 14?"
  - Subtract 5 from 14
    - 14-5 = 9
  - Subtract 5 again
    - 9-5 = 4
  - Subtracting a third time would result in a negative number. Therefore, we are done. 14/5 generates a quotient of 2, and a remainder of 4. The remainder is useful because it tells us whether we should round the quotient up or down.

Subroutine #5: calculateSquareRoot

This subroutine calculates the square root of $\frac{GM}{R}$, which in this case gives us our final velocity value $v = \sqrt{\frac{GM}{r}}$. This subroutine follows the same approach shown on the assignment page, where multiplication is done starting at the value zero. The two operands are incremented each time multiplication is done until the result of the given multiplication surpasses the value of the quotient. 16 bit multiplication is implemented in the subroutine for this purpose, as the quotient can be up to a 24 bit value. If multiplication of the two operands surpass the value of the quotient, the square root of the number, or final velocity value, has been found. The value of the operand, which serves as the velocity value, is then stored into “Velocity” in data memory. Here is a good example displaying the logic.

- 0 * 0 = 0 (too small)
- 1 * 1 = 1 (too small)
- 2 * 2 = 4 (too small)
- 3 * 3 = 9 (too small)
- ...
- 70 * 70 = 4900 (too small)
- 71 * 71 = 5041 (too small)
- 72 * 72 = 5184 (too large)

Since the instructions tell you to round downward, we conclude that $\sqrt{5123} = 71$

Subroutine #6: checkVelocity

Now that the velocity has been computed, the third error handling case is handled in this subroutine, where if the velocity of the satellite (v) equals 0, 0xFE 0xFF, the 16 bit signed value of -2 in little endian notation, is stored into “Velocity” in data memory. In this subroutine, the value of the velocity is fetched out of data memory and loaded into two registers. If the value of the two registers are equivalent to 0x00 0x00, the velocity equals 0, meaning 0xFE 0xFF is stored into “Velocity” in data memory

Subroutine #7: obtainPeriodOperands:

This routine implements 16 bit multiplication, obtaining the two 32 bit operands that will be passed into a 32 bit multiplication subroutine to obtain the product. Given the equation $T = \sqrt{\frac{4\pi^2 r^3}{GM}}$, the product, or $4\pi^2 r^3$, must be determined. In order to simplify this, I used 16 bit multiplication to multiply $4\pi^2$ * r and r * r. In the case of this assignment, $4\pi^2$ is equivalent to the decimal value 40. Using 16 bit multiplication, the values of both $4\pi^2$ * r and r * r are obtained in this subroutine and stored into memory.

Subroutine #8: calculateProduct

This routine multiplies the two 32 bit operands that were acquired in the obtainPeriodOperands subroutine. Implementing 32 bit multiplication, the result of the multiplication is at largest a 56 bit value, which is the product of the period. Once the 32 bit multiplication has completed, the result is stored into “Product” in data memory.

Subroutine #9: calculateQuotientPeriod:

This subroutine is extremely similar to calculateQuotient, except it calculates the quotient of $\frac{4\pi^2 r^3}{GM}$ instead of $\frac{GM}{r}$. It uses the same approach of subtracting until no longer possible, keeping track of the amount of times GM has been subtracted from $4\pi^2 r^3$. There are two significant differences between this subroutine and calculateQuotient; this subroutine involves subtracting the 32 bit value of GM from $4\pi^2 r^3$, which is up to a 56 bit value, whereas calculateQuotient only involved subtracting r from GM, which is subtracting a 16 bit value from a 32 bit value. This subroutine is also able to produce a

quotient of up to 48 bits, where calculateQuotient only produced a quotient of up to 24 bits.

Subroutine #10: calculateSquareRootPeriod

Similar to calculateSquareRoot, except this involves 32-bit multiplication, as the value of the quotient is much larger compared to the quotient computed in the velocity equation $v = \sqrt{\frac{GM}{r}}$. Same logic as before, where multiplication is done starting at the value zero. The two operands are incremented each time multiplication is done until the result of the given multiplication surpasses the value of the quotient. Once the result of the multiplication is greater than the quotient, the value of the operand serves as a counter of the number of times multiplication has been done. This result gives us the period of the revolution$T = \sqrt{\frac{4\pi^2 r^3}{GM}}$, which is up to a 24 bit value that is stored into “Period” in data memory.

Subroutine #11: checkPeriod

Now that the period has been calculated, the last error handling case is handled in this function, where if the period of revolution (T) is < 25 seconds. The value of the period is stored into three registers, whereas the value 0x000019 is stored into another three registers. 0x000019 is equivalent to 25 in decimal. If the value of the period stored in the three registers is < the value 0x000019, 0xFE 0xFF 0xFF, the 24 bit signed value of -2 in little endian notation, will be stored into “Period” in data memory.

Questions

In your code, what is the maximum number of bits that each stage of your program can handle? For example, what is the largest number (in bits) that your code can divide? What is the largest number (in bits) that you can multiply? When you determine the square root, what is the largest number (in bits) that your code can handle?

Within my program division, multiplication, and determining the square root are done twice, once for determining the velocity of the satellite, and once for determining the period of revolution.

For the velocity of the satellite $v = \sqrt{\frac{GM}{r}}$, 32 bit/16 bit "division" is done, where the value of GM is 32 bits and the value of the orbital radius is 16 bits. Technically, this process is simply subtracting a 16 bit number from a 32 bit number and incrementing a counter that is able to store up to 24 bits.

When calculating the square root of $v = \sqrt{\frac{GM}{r}}$, 16 bit multiplication is used, however technically only 12 bit multiplication is needed. This is due to the fact that the quotient is at largest, a 24 bit value. In terms of the actual square root answer, or velocity, the value can be at most a 16 bit value.

For the period of revolution $T = \sqrt{\frac{4\pi^2 r^3}{GM}}$, 56 bit/32 bit "division is done, where the product, or $4\pi^2 r^3$, can be at most a 56 bit value. GM is at most a 32 bit value.

When calculating the product, 32 bit multiplication is used. 32 bit multiplication is also used when calculating the square root to obtain the period of revolution, however only 24 bit multiplication is technically needed considering the quotient is at most a 48 bit value. In terms of the actual square root answer, or period of revolution, the value can be at most a 24 bit value.

Describe the algorithm that you used to determine the square root of a number.

The algorithm I used to determine the square root of a number is based on the algorithm shown on the assignment page. Given two operands starting at zero, the operands are multiplied. If the result of the multiplication is less than the quotient, in this case $\frac{GM}{R}$ or $\frac{4\pi^2 r^3}{GM}$, increment the operands by 1 and multiply again. Once the result of the multiplication is greater than the quotient, decrement the value of the operand by 1. This provides the rounded down value of the square root of the quotient. This screenshot represents the process well.

- 0 * 0 = 0 (too small)
- 1 * 1 = 1 (too small)
- 2 * 2 = 4 (too small)
- 3 * 3 = 9 (too small)
- ...
- 70 * 70 = 4900 (too small)
- 71 * 71 = 5041 (too small)
- 72 * 72 = 5184 (too large)

Since the instructions tell you to round downward, we conclude that $\sqrt{5123} = 71$

What were the primary challenges that you encountered while working on the project?

This project involved a lot of the previous concepts taught and learned in lab. I struggled with implementing many of the functionalities required, including multi-byte multiplication, subtraction, and addition. It was also difficult to come up with the logic on how to properly divide a number and take the square root of a number, as these functionalities aren't built into AVR assembly. I also found it difficult to keep track and manage personal use of registers. Although 32 registers seems like a lot, I quickly realized working with large multi-byte values requires a lot of registers.

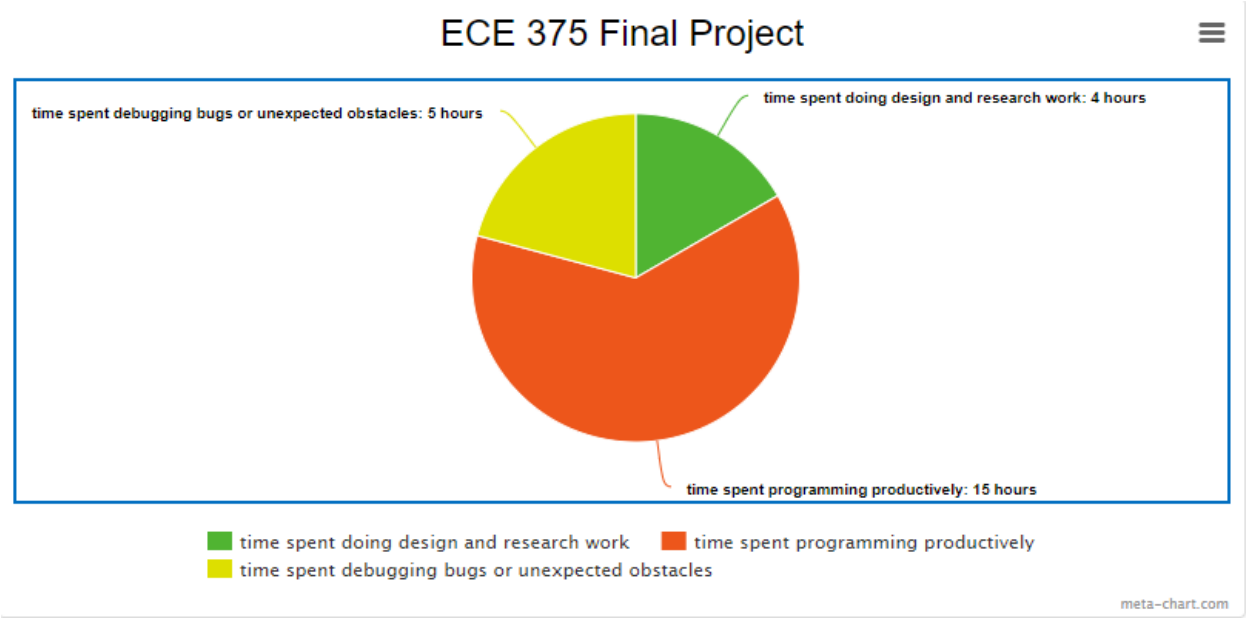
What features of the program took longest to implement?

The feature that took the longest to implement was the 32 bit multiplication. This involved a lot of mul and adc instructions. It was difficult to envision how to approach this even having learned 16 bit multiplication. Tracking each byte of the value and multiplying one byte from the upper operand with another byte from the lower operand took a lot of tracking to correctly compute the product. I also found implementing division difficult in this final project, as it involves multi-byte addition, subtraction, and comparing.

Is there anything you would design differently if you were to re-implement this project?

I would definitely make a lot of my functions more modular. In this final project I made a lot of subroutines, most likely more than I needed to. I implemented two multiplication, square root, and division functions. I definitely wasn't as efficient as I could have been in terms of my programming on this final project. I would definitely try to save time and effort next time by implementing a multiplication, square root, and division function that I could use for both the velocity and period equations.

Draw a pie chart and give estimates of the time that you invested into this final project. In particular, be sure to label these three categories:



This is roughly the amount of time spent on this final project displayed in a pie chart. 4 hours was spent doing design and research work, 15 hours was spent programming this final project, and 5 hours was spent debugging unexpected bugs and obstacles.

Conclusion

In conclusion, this final project felt like the ultimate lab project. We were tasked with implementing nearly all of the concepts we were taught in lab. This included accessing program memory, moving data from program memory to data memory, multi-byte arithmetic, and more. After completing this final project, I feel I have become much more comfortable programming in assembly language and have learned a lot as a result. I feel much more confident programming in assembly and have a much better understanding of AVR assembly instructions, logic, and memory management. Although difficult, this final project felt very rewarding.