ECE 375 Project Report

By Mitchell Radford

3/12/2021

Routines and Macros

This section will cover all routines, subroutines, and macros. It will also state the maximum operand size of each function.

Initialization Routine

This is the routine where all the calculations and functions are called from. The first step in the process is to initialize the stack pointer. Once this is done, the PlanetChoice routine is called to fetch the desired GM. Once a planet GM has been stored into memory at $0100-$0103, the orbital radius is fetched from memory and saved into memory at $0104-0105 for easy reference. Since orbital radius is a two-byte number (16 bit) at most, only 2 bytes are ever written to or read from that location. Once this is fetched, the program calls CheckCases1And2 to check two special cases. Next, Registers A, B, and XL are cleared as they were used earlier.

This next part of the initialization routine covers problem A: orbital velocity. GM is fetched from memory again and stored into registers A-D. The orbital radius is store into registers H and I. Now that the registers are set up, BigDiv is called to divide GM by radius. Once the division is finished, the MoveResult macro is called to move the division result from A-G to O-T and XL. This is the quotient and will at most be 3 bytes. As such, three bytes are written to memory at the location of the QUOTIENT label. Once saved, the program calls Sqrt to root the quotient and yield the orbital velocity. CheckCase3 is then called to verify that the root is not less than or equal to 0, and the velocity is written to memory at the location of the VELOCITY label.

The last part of the initialization routine covers problem B: orbital period. Before performing any calculations, all registers are cleared. The orbital radius is then loaded from memory into registers A and B. The radius is then stored into memory at the locations of the mul48 operands, and the memory at the result location is cleared to ensure the correct result. Mul48 is then called to square the radius. This result is loaded into registers A-D and stored back as Mul48_Op1. Next, the result is cleared again before Mul48 is called to cube the radius. Once again, the result is moved to Mul48_Op1 and the result memory is reset. Operand 2 is then cleared and replaced with 0x28, which is the value of 4 * pi^2. This final multiplication result is loaded into registers A-G and stored into memory at the product label. The GM is then loaded from memory and stored into registers H-K before BigDiv is called to perform the division. Once the division is complete, the MoveResult macro is used to move the contents of registers O-XL to registers A-G. Next, the memory holding the multiplication operands and result are cleared, and the Sqrt routine is called.

After Sqrt completes, CheckCase4 is called to check whether the orbital radius is less than 25. If it is not, then the program continues. Finally, the contents of registers O, P and Q are stored into memory at the period label and the program jumps to Grading to be checked.

PlanetChoice Routine

The PlanetChoice routine checks the SelectedPlanet value and retrieves the corresponding GM from memory. This value is re-written to memory from $0100-$0103 for ease of access later. To check which GM to read, this routine checks the SelectedPlanet value against all possible values, where possible values are zero to eight.

CheckCases1And2 Routine

The CheckCases1And2 routine starts by loading 1000 into the Y register. Comp64 is then called to compare the contents of registers A and B to that of Y. At this time, A and B hold the orbital radius value. If this value is less than or equal to 1000, the program loads -1 into XL and stores this into memory at the velocity label. The program then jumps to Grading to finish.

If the orbital radius is greater than 1000, GM is loaded from memory into registers A B C D. Comp64 is used again to compare the values. If the value in R16 is not 0x02, the program stores the value of -2 into memory at the period label via the X register. If both tests pass, the subroutine returns.

CheckCase3 Routine

The CheckCase3 routine compares the values of the O and P registers to zero using Comp64. If these are both zero, then -2 is loaded into XL and written into the two bytes of memory at the velocity label and the program jumps to Grading. Otherwise, the subroutine exits.

CheckCase4 Routine

The CheckCase4 routine compares the values of the O, P and Q registers to 0x0019 using Comp64. If O P Q is not greater than 0x0019, -2 is loaded into XL and written into the three bytes of memory at the period label and the program jumps to Grading. Otherwise, the subroutine exits.

BigDiv Routine and Algorithm

The BigDiv routine is a division routine, made to support division of two numbers up to 72 bits (7 bytes). The numerator uses registers A-G, while the denominator uses F-N. The result is stored in O-T and XL. The division algorithm used is the simple check-and-subtract. If the numerator (A-G) is greater than the denominator (H-N), then the program subtracts the denominator from the numerator and adds to a tally (O-XL). The Comp64 macro is called to compare the numerator and denominator. Once the denominator is greater than the numerator, the program enters step 2.

In step two, the denominator is halved. If the numerator is greater than the new denominator, one is added to the result in O-XL as rounding up. Otherwise, the result remains the same and the routine is returns.

Mul48 Routine and Algorithm

The Mul48 routine supports the multiplication of at most two 6 byte (48 bits) numbers, providing a result of 12 bytes (96 bits). The operands are provided in the memory locations of Mul48_Op1 and Mul48_Op2. This routine uses an extended version of the 24-bit multiplication created in Lab 5. The operand locations are loaded into X and Y, while the result is loaded into Z. For six inner loops (N), the Nth operand1 bit is multiplied the the Ith operand2

bit. Any overflow is stored into registers Q-T and added to the current values at their respective memory locations in the result. After six loops, the result memory location is set back by 5 bytes and the process is repeated for a total of six outer loops (I). The final result is stored in memory at the location of Mul48_Res.

Sqrt Routine and Algorithm

The Sqrt routine calculates the root of a value of up to 6 bytes (48 bits). The operand is stored in registers A-F and the intermittent multiplication results are stored in H-M. The final result is stored in O-XL. The algorithm used is a multiply-and-check method. The intermittent result is kept at the location of the multiplication result in memory and loaded into O-XL to be updated each loop. One is added to this value with Add64 before it is stored into memory at the location of both multiplication operands and the result is cleared out. Mul48 is then called to multiply the values. The result is then grabbed from memory and placed into registers H-M so it can be compared to the expected result (the Sqrt operand) using Comp64. If the multiplication result is less than the Sqrt operand, the program loops again.

If the values are equal, the last multiplication operand is the result, and loaded from memory into O-XL. Otherwise, the result is greater, so the last multiplication operand is loaded from memory into O-XL and Sub64 is called to subtract one from it before returning.

Add64 Macro

This macro performs a simple 56-bit addition on 7 input registers. The result is kept in the same registers as the first operand.

Sub64 Macro

This macro performs a simple 56-bit subtraction on 7 input registers. The result is kept in the same registers as the first operand.

Comp64 Macro

The Comp64 macro performs a comparison of two sets of 7 registers values. As such, it supports numbers up to 56 bits. If the first comparison results in Operand1 < Operand2, a value of 0x03 is loaded into R16 and the macro finishes. Otherwise, the bits are compared again. If Operand 1 > Operand 2, 0x02 is loaded into R16. Otherwise, 0x01 is loaded into R16. R16 can then be compared in the main code to check for specific cases after this macro has run.

DivBy2 Macro

The DivBy2 macro performs a right-shift on 7 registers from the highest bit to the lowest bit. This effectively halves the contents of the registers. DivBy2 supports numbers up to 7 bytes (56 bits) and does not shift the highest byte through carry.

MulBy2 Macro

The MulBy2 macro performs a left-shift on 7 registers from the lowest bit to the highest bit. This effectively doubles the contents of the registers. MulBy2 supports numbers up to 7 bytes (56 bits) and does not shift the first byte through carry.

MoveResult Macro

The MoveResult macro is a simple macro that moves a 56-bit number from one set of registers into another via the MOV instruction.

Challenges

The primary challenges faced in this project included expanding multiplication and creating the square root function. I had not had 24-bit multiplication working yet due to the lack of carrying the result over two registers, so it took some time to troubleshoot and work out where exactly to carry. However, once I figured out which part of the code should be changed to carry more bits, it was an easy fix.

Creating a square root function was slightly problematic due to the use of many registers. Mul48 also uses many registers for carrying, so eventually I figured I should use memory loading and storing, despite it being a bit slower than I would have liked.

The feature that took the longest to implement was also the square root function. It uses the most loading and storing of any function that isn't Init and was dependent on how multiplication was run. I couldn't use the same registers in both because Mul48 overwrites several. As such, I had to re-design my square root to load and store into memory. Although not difficult or very time consuming, it meant changing the form of my function, which took a little bit longer than the rest of my functions.

If I were to re-implement this project, I might try to optimize it more by operating smaller numbers more times so that there aren't huge chunks of division in the larger cases. I might also try to manage my registers better by not relying on memory loading and storing, possibly by implementing an alternative multiplication method.

Time Allocation

The pie chart to the right shows the amount of time that was spent on each segment of this project. The numbers are in units of hours.