

Final Project

Submit Assignment

Due Mar 18 by 11:59pm

Points 156

Submitting a file upload

File Types pdf and asm

ECE375 Final Project

Submit your work by **11:59pm on Thursday, March 18th, 2021.**

Late submissions will be deducted 20 points per day.

Your submission should consist of two items:

1. 100 points - a single .asm file containing your AVR assembly source code
2. 56 points - a PDF document containing your typed report

Your grade will be reduced if you do not use the [assembly code template file](#)  that is provided.

Please take the time to read the entire documentation.

Introduction

In July of 1969, the Apollo 11 mission departed earth and headed to space for an eight day visit. The spacecraft consisted of three distinct components: the command module, the service module, and the lunar module. The command module and lunar modules each contained an AGC (Apollo Guidance Computer).

The specifications of the AGC were as follows:

- 16-bit word length (divided into 15 data bits and 1 parity bit)
- 2048 words of RAM
- 36,864 words of ROM (program memory)
- 2.048 MHz clock oscillator

The AGC was designed in a similar manner to the CPUs which we have discussed during lecture. The instruction format of the computer utilized 3 bits for the opcode and 12 bits for the address. An accumulator register was available for mathematical operations. The AGC provided five vectored interrupts which could be used for timer support and remote communications.

AGC Assembly Code

The Apollo Guidance Computer was developed specifically to control the spacecraft on its way to the moon and to manage the many details that were required to make the mission successful. Suffice to say, this was not an easy job. Perhaps most fascinating, the Apollo 11 assembly code has been transcribed and uploaded to GitHub for public review ([see here](https://github.com/chrislgarry/Apollo-11) [_\(https://github.com/chrislgarry/Apollo-11\)_](https://github.com/chrislgarry/Apollo-11)).

The programmers included various annotations throughout the source code, some of which would not be out of place in ECE375 assembly code. The assembly commands are different, but it's easy to recognize the basic structure of comparisons and conditional branches.

For example, consider this snippet of AGC assembly code (pay special attention to the comments):

```
KILLTSK2      LXCH   ITEMP2      # SAVE CALLER'S BBANK
[...]
```

```
ADRSCAN      ZL
INDEX        L
CS           LST2
AD           ITEMP4      # COMPARE GENADRS
EXTEND
BZF          TSTFBANK     # IF THEY MATCH, COMPARE FBANKS
LETITLIV     CS          LSTLIM
AD           L
EXTEND       # ARE WE DONE?
BZF          DEAD        # YES -- DONE, SO RETURN
INCR         L
INCR         L
TCF          ADRSCAN     # CONTINUE LOOP.
```

Other areas of the code are more specific to the world of rocket engines...

```
IGNYET?      CAF      ASTNBIT     # CHECK ASTNFLAG: HAS ASTRONAUT RESPONDED
MASK         FLAGWRD7  # TO OUR ENGINE ENABLE REQUEST?
EXTEND
INDEX        WHICH
BZF          12        # BRANCH IF HE HAS NOT RESPONDED YET

IGNITION     CS        FLAGWRD5    # INSURE ENGONFLG IS SET.
MASK         ENGONBIT
ADS          FLAGWRD5
CS           PRI030     # TURN ON THE ENGINE.
```

My personal favorite section of the code is located [in this file](https://github.com/chrislgarry/Apollo-11/blob/master/Luminary099/THE_LUNAR_LANDING.agc#L245) [_\(https://github.com/chrislgarry/Apollo-11/blob/master/Luminary099/THE_LUNAR_LANDING.agc#L245\)_](https://github.com/chrislgarry/Apollo-11/blob/master/Luminary099/THE_LUNAR_LANDING.agc#L245) which apparently involved proper alignment of the landing radar antenna:

```
BZF          P63SP0T4    # BRANCH IF ANTENNA ALREADY IN POSITION 1

CAF          CODE500     # ASTRONAUT: PLEASE CRANK THE
```

	TC	BANKCALL	#	SILLY THING AROUND
	CADR	GOPERF1		
	TCF	GOTOP00H	# TERMINATE	
	TCF	P63SP0T3	# PROCEED	SEE IF HE'S LYING
P63SP0T4	TC	BANKCALL	# ENTER	INITIALIZE LANDING RADAR
	CADR	SETPOS1		
	TC	POSTJUMP	# OFF TO SEE THE WIZARD...	

Your Assignment

Navigating in space requires a lot of physics and math. In order to get to the moon, you need trigonometric equations, a bit of calculus, lots of arithmetic, and a reliable way to propel and steer your spacecraft. Of course, you also need to design and build a sturdy spaceship.

The AVR microcontroller on your lab board is arguably more powerful than the CPU that drove the Apollo 11 mission. In honor of this early computer, we will use our ATmega128 board to solve some simple mathematical equations that are related to spaceflight.

What do you have to do?

In this final project you will write AVR assembly code to solve some equations as described below. You will store the computed results into data memory. **A template file is provided to you with details about the various operands.** You must write your source code within the template file. Please be sure to read this entire document in order to understand how the values will be provided to your code. Each time your program runs, you will solve the equations shown below.

All math will be performed using integers. No floating point operations are required!

Satellite Velocity (problem A)

Calculate the velocity of an artificial satellite orbiting an object in space. Your answer must be in units of kilometers per second.

$$v = \sqrt{\frac{GM}{r}}$$

You will be provided the radius of the orbit (r) and the standard gravitational parameter (GM) corresponding to a particular planet. Round your answer at all stages of the equation. For example, $\frac{GM}{r}$ should be rounded to the nearest integer when performing division. If the square root of $\frac{GM}{r}$ is not already an integer, round it downward.

Math Example

Please look at the template file as you read through this example. Suppose that the source code file indicates that we have an OrbitalRadius of **0x64, 0x19**. Assume that it also specifies SelectedPlanet as **0x02**.

Since the values are stored in little-endian format, the OrbitalRadius has a decimal value of **6500**. With a SelectedPlanet value of 2, this is indicating that we need to use the GM value that is located at index 2 of the array (explained within the template file). Based on the example file, this corresponds to the hexadecimal data **0x08, 0x15, 0x06, 0x00**. Interpreting this as an unsigned 32 bit value in little-endian format, it's equivalent to the decimal value **398,600**.

We now compute $\frac{GM}{r} = \frac{398,600}{6500} = 61.3230 = 61$

When you perform division, you will round to the nearest integer. That's why we rounded to 61 even though 61.3230 is a closer approximation. You need to store this value into memory at address "Quotient". This allows us to evaluate your code for partial credit.

When the grader inspects the Quotient in data memory (using Atmel Studio) they will see the 24 bit representation in little-endian format:

```
0x3d, 0x00, 0x00
```

Next, we need to compute the square root: $\sqrt{61} = 7.8102... = 7$ (note that we round down for square roots)

Our final answer is 7 km/s. The value 7 should be stored into the "Velocity" location as a 24 bit representation:

```
0x07, 0x00, 0x00
```

Period of Revolution (problem B)

What is the period of revolution for the satellite described in problem A? Provide your answer in units of seconds.

As before, you will be provided the radius of the orbit (r) and the standard gravitational parameter (GM). Round to the nearest integer when performing division, and round downward when computing square roots.

For the sake of simplicity, your program should approximate π^2 as 10.

$$T = \sqrt{\frac{4\pi^2 r^3}{GM}}$$

Math Example

For this example let's use the same parameters as we used for part A.

We first need to compute $4\pi^2 r^3$

Plugging in the orbital radius value of 6500, we get

$$4 \cdot 10 \cdot 6500^3 = 10,985,000,000,000 = 0x9FDA505DA00(\text{big-endian})$$

(note that we used the approximation of $\pi^2 = 10$)

We need to store this intermediate value into data memory at the "Product" location. As before, this is used for the grader to consider partial credit. The number will be stored as a 7 byte little-endian value:

0x00, 0xDA, 0x05, 0xA5, 0xFD, 0x09, 0x00

Our next step is to compute the quotient:

$$\frac{10,985,000,000,000}{398,600} = 27,558,956.347 = 27,558,956 \text{ (after rounding to the nearest integer)}$$

Finally, we need the square root:

$$\sqrt{27,558,956} = 5249.662 = 5249 \text{ (after rounding downward)}$$

The square root will be stored into data memory at the "Period" location:

0x81, 0x14, 0x00

Special Cases

Your code will also be capable of indicating four special error cases:

- If $r \leq 1000$ then you must store a value of -1 into "Velocity" to indicate the error.
- If your code detects a computed velocity value of 0 km/s (after rounding), store a value of -2 into "Velocity" to indicate the error.
- If $GM \leq 1000$ then you must store a value of -1 into "Period" to indicate the error.
- If your code detects a period of revolution that is less than 25 seconds (after rounding), store a value of -2 into "Period" to indicate the error.

After handling any of these special cases, your code should halt (you do not need to continue computing the math equations).

Dealing with Parameters

In order to compute the correct values, your program needs to know the orbital radius and standard gravitational parameter (which depends on the satellite's height and the particular planet you are orbiting). These values will be provided to you within specific program memory locations. **All values are provided in little-endian format.**

Inside the template code, there is a location named "OrbitalRadius". That value contains the radius (in kilometers). The orbital radius is an unsigned 16-bit value (implying that the maximum value is 65,535).


The template file provides you with a list of gravitational parameter values. The units are $(\text{km} * \text{km} * \text{km})/(\text{sec} * \text{sec})$. You must retrieve the correct GM value by utilizing the "SelectedPlanet" information that is specified at runtime. The template file provides you with 9 different GM values, each of which corresponds to a particular planet. As an example, if SelectedPlanet has a value of 2, then your code must use the GM value that is stored at index 2 within the array. From looking at the template file, we can see that this is the GM value corresponding to Earth. Please see the template file for more context. Since each GM is an unsigned 32 bit value, the maximum value is 4,294,967,295.

Your solution **MUST** read the provided 32-bit GM values at runtime. You are not allowed to hardcode the GM values into your code! These values will be changed during testing!

Some Ideas to Consider

The AVR does not have built-in instructions to divide numbers or compute the square root. As a result, you will need to implement your own approach to compute those values. There are lots of ways to do this (feel free to implement any algorithm of your choosing). Since you are specifically working with integers (and not floating point numbers) both of these mathematical tasks are simplified.

- 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.
 - A small code snippet is [available here](#) . It demonstrates the concept using 8 bit values.
- The simplest way to determine the square root of a number is to start from 0 and work your way up. You can compute the square of a number, check to see if the result is correct, and continue trying.
 - For example, suppose that you want to determine the square root of 5,123.
 - We try all options beginning from zero. Eventually, we will work up to the correct value.
 - $0 * 0 = 0$ (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 $\sqrt{5123} = 71$
 - For our purposes, the correct value is 71 (even though a calculator says 71.575135347)
- We are only working with integers, hence the rounding.

Additional Details

- When you are rounding quotients, use the convention that a value of 0.5 rounds up (and anything lower rounds down). For example, a value of 153.4999 would round to 153. A value of 536.501 would round to 537.
- When you are rounding square roots, you must round down (unless the value happens to be a perfect square). For example, $\sqrt{25} = 5$ and $\sqrt{24} = 4$
I understand that this isn't the most accurate approach, but it allows us to simplify the assignment and make it faster to implement.
- Your submission will be tested with a variety of parameters. Do not hardcode the answers.
- All numerical values that are provided to your program will be arranged in little-endian format. Furthermore, all of your answers must be stored in little-endian format.
- This is an individual assignment. You are not allowed to work in groups (and submissions will be compared for signs of plagiarism).
- You are encouraged to re-use your work from earlier assignments. It is perfectly fine to incorporate code that you wrote previously in this course (including any lab code that was written in collaboration with a partner).
- Please, design your project on paper and try to envision any obstacles before you begin programming. It is much easier to fix bugs at design time (rather than realizing the problem after you've written the code). Design twice, code once.
- My number one suggestion... design your code in a modular fashion. For example, write your square root code inside a procedure that you can call from anywhere. On a similar note, if you implement a multiplication procedure, write it in such a way that you can reuse it for other parts of your code.
- You will not receive credit during testing if your code fails to reach the "Grading" label (or consumes more than 10 seconds of CPU time).

Materials

Required template file - [available here](#) 

Outside Resources

While working on the exam, students are welcome to use code from the ECE375 slides, previous homework, or previous labs. You may recycle code that you have written previously.

It is acceptable to adapt an algorithm from an outside source but you must write the assembly code yourself. Do not copy assembly code from others!

Extra Credit

There are two options for which you can earn extra credit. In total, you can earn up to 10 additional points by implementing these features.

- 4 pts - Design your code in such a way that π^2 is approximated as 9.8696 (rather than 10). This will make your answers more accurate. As with the standard implementation, you will still round your final answers. There are multiple ways to tackle this challenge.
- 6 pts - Write your code so that the maximum allowable orbital radius can be as large as 120,000km. This means that your code will need to accept a 24 bit unsigned value in the "OrbitalRadius" field. See the template assembly file for more context. The original project implementation can only handle a maximum radius of 65,535km but your extra credit implementation will be tested with values as large as 120,000km. As with all parameters, the orbital radius will still be provided in little-endian format.

Note: even though the radius will be provided as a 24 bit number, you can assume it will never be larger than 120,000 (in decimal).

Written Report

Document your work in a well-written report. Be sure that your report includes all of the content that is listed in this section. Present your work to the reader in a professional manner. I suggest that you use images, charts, diagrams or other visual techniques to help convey your information to the reader. Make your report something that is intriguing and interesting.

- Explain how you implemented your source code to compute the satellite details. You should provide enough information that a knowledgeable programmer would be able to draw a reasonably accurate block diagram of your program.
- This project includes division, multiplication, and square root operations. 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?
- Describe the algorithm that you used to determine the square root of a number.
- What were the primary challenges that you encountered while working on the project?
- What features of the program took longest to implement?
- Is there anything you would design differently if you were to re-implement this project?
- 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:
 - time spent doing design and research work

- time spent programming productively
- time spent debugging bugs or unexpected obstacles
- Proofread your report! Better yet, ask another person to proofread your work. You will be marked down for spelling or grammatical mistakes.
- If you are implementing extra credit work, you MUST include an explanation of your additional work within the report.

There is no arbitrary minimum length requirement on the report. For example, if you are able to document your work and answer all of the questions in a two page document, that is perfectly fine. If you need five pages, that's perfectly fine too.

I will review your assembly code separately, so there is no need to include the source code within your final project report (unless you are describing certain snippets of the code).

Where to get help?

If you are working on the design of your final project, feel free to stop by office hours and ask questions. Since this is the final project, I expect students to demonstrate initiative while testing and debugging their code. We are happy to provide feedback and answer questions, but if you encounter a bug in your code, you will need to use your debugging skills to resolve it.

If you have a question regarding the project requirements, please let me know (either during lecture or send me an email). If the instructions are unclear, I can update the documentation and add an entry in the errata. justin.goins@oregonstate.edu (<mailto:justin.goins@oregonstate.edu>).

Grading Rubric

The grading rubric will be based on the requirements that have been included in this document. A more specific breakdown of the point distribution will be provided at the end of week 9.

Your code will be tested with a wide variety of configuration settings. Keep this in mind while designing your algorithms and test your code so that it works for all valid combinations of input data.

Errata

This section will be updated when changes or clarifications are made. Each entry here should have a date and brief description of the change so that you can look over the errata and easily see if any updates have been made since your last review.

March 1st - Posted the original documentation