

|  |  |
| --- | --- |
| **Intake: APD1F2409CS(CYB)** | **Module Code: CT046-3-1-ICS** |

|  |  |  |
| --- | --- | --- |
| **HAND OUT DATE: Thursday, 24 April 2025 Week 6** |  |  |
| **SUBMISSION DATE: Monday, 30 June 2025 Week 14** |  |  |

**TITLE:**

***SECTION A: ASSEMBLY PROGRAMMING***

**Module Lecturer: Ms. VIJAYALAXMI AMUTHAN**

**GROUP NAME 9**

|  |  |
| --- | --- |
| **AHMED MIRAHUSAIN ALVI** | **TP084807** |
| **ABDALLAH MOHAMED MAHMOUD MOHAMED** | **TP085097** |
| **SULTAN ABDULLA OMAR TAKRORI** | **TP085327** |
| **MOHAMMED YOUSEF MOHAMMED MOHAMMED** | **TP085042** |
| **MAHRUS SHAMSUL AHSAN** | **TP085562** |
| **ABDUL AZIZ VAYANI** | **TP082376** |

TABLE OF CONTENTS

[CHAPTER 1: INTRODUCTION 1](#_Toc201947979)

[CHAPTER 2: ASSEMBLY PROGRAM AND ANALYSIS 2](#_Toc201947980)

[**2.1** Main Menu 2](#_Toc201947981)

[**2.2** Ahmed Mirahusain Alvi 3](#_Toc201947982)

[**2.2.1** Hexagon 3](#_Toc201947983)

[**2.2.2** Parallelogram 4](#_Toc201947984)

[**2.3** Mahrus Shamsul Ahsan 5](#_Toc201947985)

[**2.3.1** Square 5](#_Toc201947986)

[**2.3.2** Rectangle 6](#_Toc201947987)

[**2.4** Mohammed Yousef Mohammed Mohammed 7](#_Toc201947988)

[**2.4.1** Alphabet X 7](#_Toc201947989)

[**2.4.2** Trapezium 8](#_Toc201947990)

[**2.5** Sultan Omar Abdulla Takrori 9](#_Toc201947991)

[**2.5.1** Pentagon 9](#_Toc201947992)

[**2.5.2** Diamond 10](#_Toc201947993)

[**2.6** Abdul Aziz Vayani 11](#_Toc201947994)

[**2.6.1** Triangle 11](#_Toc201947995)

[**2.6.2** Right Arrow 12](#_Toc201947996)

[**2.7** Abdallah Mohamed Mahmoud Mohamed Mahmoud 13](#_Toc201947997)

[**2.7.1** Heart 13](#_Toc201947998)

[**2.7.2** Star 14](#_Toc201947999)

[CONCLUSION 15](#_Toc201948000)

[REFERENCES 16](#_Toc201948001)

[WORKLOAD MATRIX 17](#_Toc201948002)

CHAPTER 1: INTRODUCTION

This application enables users to create shapes in a terminal. Our Shape Generation System accepts twelve different shapes, and dynamically generated development processes, as opposed to predefined ones. The program taken as a whole is written for the Netwide Assembler (NASM) in 64bit and uses raw manipulation of registers and memory to spit highly accurate and highly efficient output. Once loaded, the user is presented with an option to select one of two display colors giving an easy way to personalize the color of the shape. Furthermore, each form is represented by a different ASCII character, which is an example of different characters representing. The package automatically handles invalid inputs and unexpected runtime situations to make sure everything is reliable. Combining dynamic generation, user customization, and novel symbols, this system represents a concise and robust incumbent of an assembly-language application. This tiny tool is a powerful lesson about low-level programming.

CHAPTER 2: ASSEMBLY PROGRAM AND ANALYSIS

## **2.1** Main Menu

A black background with white text

Description automatically generated

Global \_start indicates the entry point. \_start is where the executions starts.

A screen shot of a computer program

Description automatically generated

Here we display a welcome message in the terminal to the user.

A screen shot of a computer program

Description automatically generated

Next, we print the shape menu in the terminal to show the user all the different choices of shapes.

A screen shot of a computer program

Description automatically generated

Next we prompt the user to choose a shape

A screen shot of a computer code

Description automatically generated

Once a choice has been made, it is read.

A black background with white text

Description automatically generated

After its read, the choice must be converted from ASCII to real number.

A screenshot of a computer program

Description automatically generated

now we enter a loop to convert each character to a number. If any character is not a digit, then jump to error.

A screenshot of a computer program

Description automatically generated

Here we validate and make sure whether the number entered by the user is within the range of 1- 12. If not, an error message is shown, and the user is prompted again.

A screenshot of a computer program

Description automatically generated

Here we compare the user input, then jump and redirect the user to the menu of the shape they have chosen.

A screen shot of a computer program

Description automatically generated

Here we have error handling. If input is invalid, then an error message is printed, and it loops back to the start.

**2.2** Ahmed Mirahusain Alvi

**2.2.1** Polygon

A screenshot of a computer program

Description automatically generated

This assembly code, labeled `draw\_polygon`, appears to be an attempt to draw a shape (likely a triangle or part of a parallelogram's slanting side) by printing a decreasing number of "+" symbols per row. It initializes `r12` and `r14` with the `shape\_size`, intending `r12` for the total number of rows and `r14` for the initial number of "+" symbols per row. The `print\_row` section then enters a loop (`print\_pluses`) that prints a "+" character repeatedly until the count in `rbx` equals `r14` (the desired number of stars for the current row). After printing the "+" symbols, a newline character is printed. Then, `r14` is decremented (reducing the number of stars for the next row) and `r15` (the row counter) is incremented. The process loops back to `print\_row` as long as the current row count (`r15`) is less than or equal to `r12` (the total number of rows to print).

A screenshot of a computer program

Description automatically generated

This assembly code snippet defines several utility routines. The set\_green\_color routine uses a sys\_write call to output an ANSI escape code to stdout, which changes the terminal's text color to green. Similarly, the reset\_to\_default\_color routine outputs another ANSI escape code to stdout to revert the terminal's text color to its default. The print\_newline routine simply prints a newline character to the console. Finally, the exit\_program routine uses the sys\_exit system call (rax set to 60) with an exit code of 0 (rdi set to 0) to gracefully terminate the program. These routines encapsulate common terminal operations, making the main program logic cleaner.

A screenshot of a computer program

Description automatically generated

This assembly code section, ask\_continue, prompts the user to decide whether to continue running the program. It first calls a print\_newline routine for formatting, then displays the "Do you wish to continue? (y/n):" prompt using a sys\_write call. After the prompt, it uses a sys\_read call to read the user's input (up to 10 bytes) into continue\_buffer. Finally, it processes the first character of the user's input: if it's 'y', the program jumps to the \_start label (presumably to restart the main program logic); if it's 'n', it jumps to the exit\_program routine; for any other input, it loops back to ask\_continue to re-prompt the user.

**2.2.2** Parallelogram

A screenshot of a computer program

Description automatically generated

This .data section in an assembly language program defines various text messages and single characters used for user interaction and formatting. It includes prompts for user input like "Enter the size of parallelogram (3-20):" and "Do you wish to continue? (y/n):", along with an error message for invalid input. Additionally, it stores individual characters such as "+" and a space, and a newline character for formatting output. In short, this section serves as a storage area for all the static text and basic display elements the program needs.

A screen shot of a computer

Description automatically generated

The size\_prompt contains assembly code that first **prompts the user** to enter a shape size and then **reads their input** from the keyboard. It uses system calls to write the prompt message "Enter the size of parallelogram (3-20):" to the terminal and then to read up to 10 characters of the user's response, storing it in a memory location called shape\_size.

A screenshot of a computer program

Description automatically generated

This assembly code section, input\_validate, is responsible for **processing and validating user input** for the shape size. It iterates through each character of the input, checking for newline or null terminators to end the input processing. For each character, it verifies if it's a digit between '0' and '9'. If not, or if the input is too long (more than 3 characters assumed based on cmp rcx, 4), it jumps to a size\_error. Valid digits are converted from their ASCII representation to a numerical value and accumulated to form the final integer. After processing all characters, the post\_validation section checks if the resulting numerical shape size falls within the valid range of 3 to 20. If it's outside this range, it also jumps to size\_error. Finally, if all checks pass, it stores the validated size and prints a newline character to the terminal for formatting.

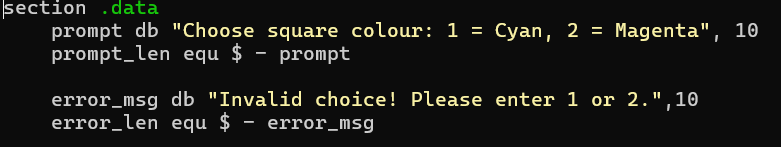
A screenshot of a computer program

Description automatically generated

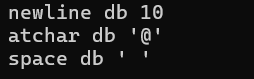
This assembly code segment, draw\_parallelogram, is responsible for drawing a single line of the parallelogram, specifically the top or bottom horizontal line. It begins by loading the validated shape\_size into register r8. The fixed\_plus\_outer and fixed\_plus\_spaces sections then print a calculated number of leading spaces based on the shape size, decrementing a counter (r9) with each space printed until the count reaches zero. Following the spaces, the fixed\_plus\_print and fixed\_plus\_loop sections print a fixed number of "+" symbols equal to the shape\_size, again using a loop and a counter (r10). Finally, a newline character is printed to move to the next line, preparing for the drawing of the next part of the parallelogram.

**2.3** Mahrus Shamsul Ahsan

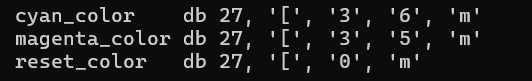
**2.3.1** Square



Here we have a named section where we store constant data. Variable prompt is used to store the string and variable prompt\_len is used to store the length of the string. Variable error\_msg stores the string for error message while error\_len stores the length of that string.



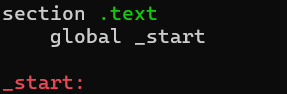
The variable newline is used to store a newline, atchar is used to store the special character ‘@’ and space is used to store a space.



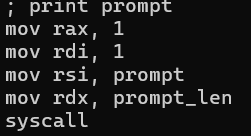
here we initialize the ANSI escape sequences to colour the text to variables.



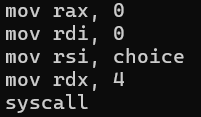
this is the section for uninitialized memory. choice is a label for the input buffer, and it reserves 4 bytes.



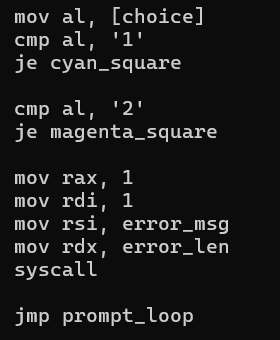
Here we enter the main code. The global \_start indicates the entry point. \_start indicates where the program should begin execution.



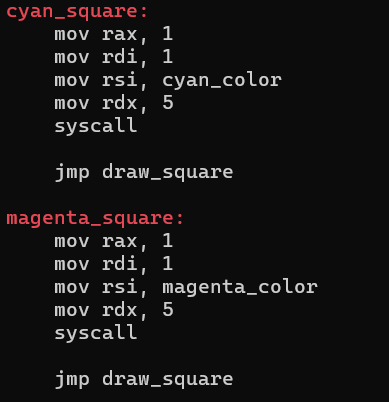
Here we display the prompt message to the user using the previously initialized variables.



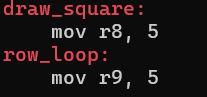
Here we read the user input



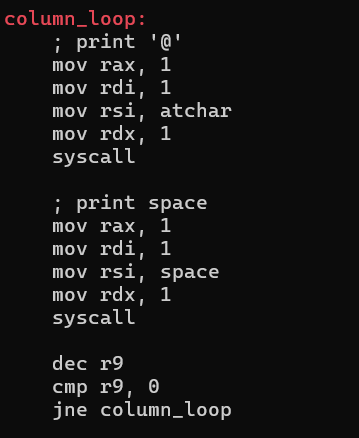
Here we check and compare the user input. If the user input is 1 the program jumps to the cyan\_square loop, if the user input is 2, it jumps to the magenta\_square part. If invalid number was input, an error message is displayed, and the user is asked to enter a correct value.



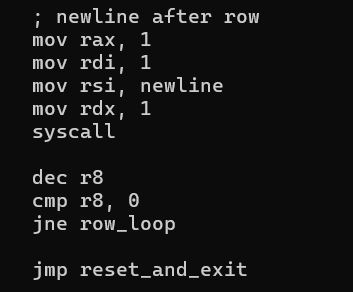
Here we set the colour of the text in the terminal based on the user input. The variable previously initialized with the ANSI colour are being called here. And once the colour of the text has been set, the program jumps to the drawing the shape



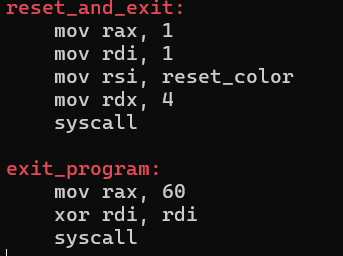
Here we assign the variables r8 and r9 the dimensions of the square. The square is made up of 5 rows and 5 columns.



here we use looping to print the shape. First a ‘@’ character is printed followed by space and this is repeated until there are five ‘@’s and 5 spaces. This looping is for a row. It loops until all columns in that row are printed.

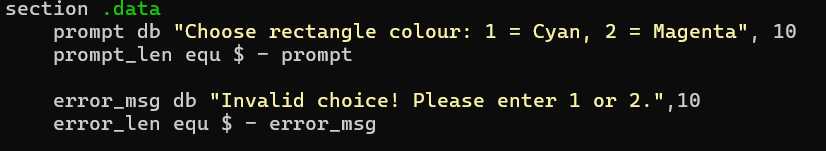


Here we print newline character. Then once the shape is completed the program jumps to reset and exit part.

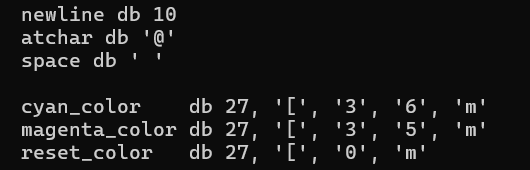


Once the shape has been completed, the colour is reset back to the default colour. Then we exit the program.

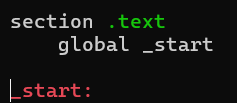
**2.3.2** Rectangle

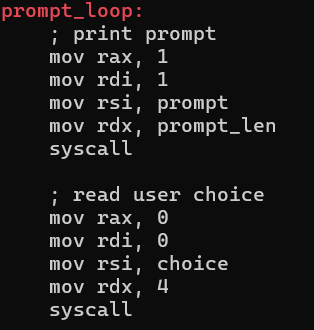


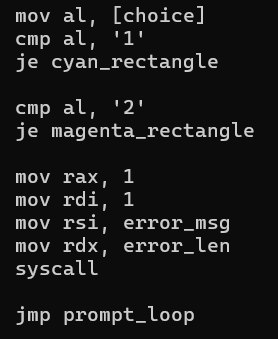
Here we have a named section where we store constant data. Variable prompt is used to store the string and variable prompt\_len is used to store the length of the string. Variable error\_msg stores the string for error message while error\_len stores the length of that string.

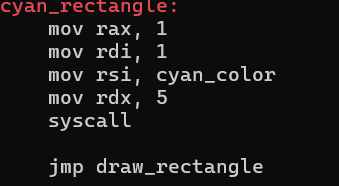


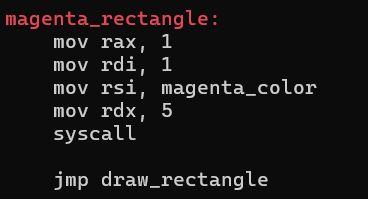


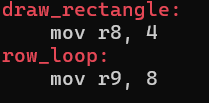


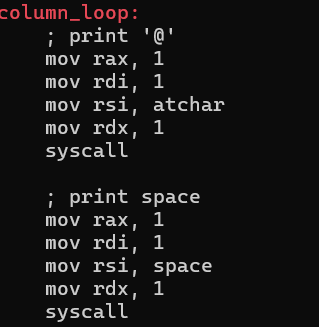


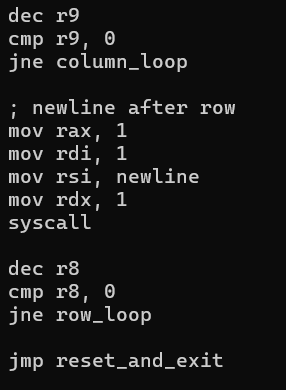


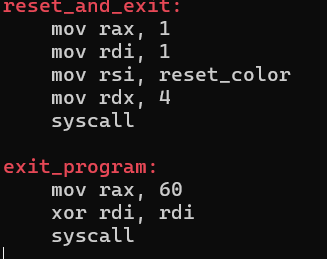






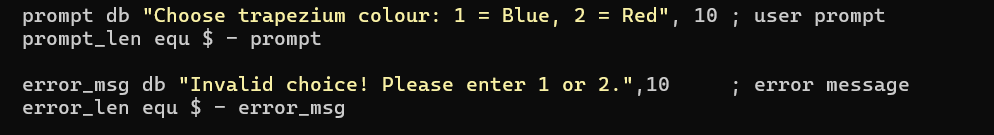




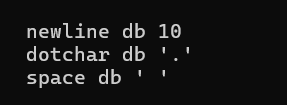


**2.4** Mohammed Yousef Mohammed Mohammed

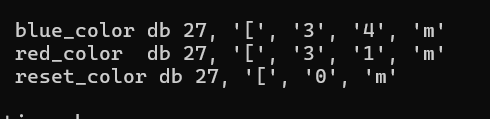
**2.4.1** Trapezium



These are the messages the program shows to the user. The first message asks the user to choose a colour. The second one warns the user if they type an invalid option.



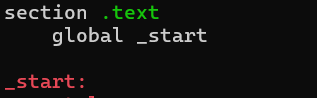
**newline** adds a line break. **dotchar** is the dot used to draw the “**trapezium**”. **space** is used for spacing and shaping the “**trapezium**”.



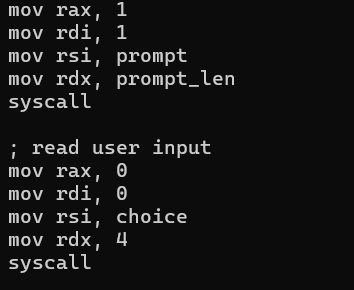
**blue\_color** makes the text blue. **red\_color** makes the text red. **reset\_color** changes the colour back to normal afterward.



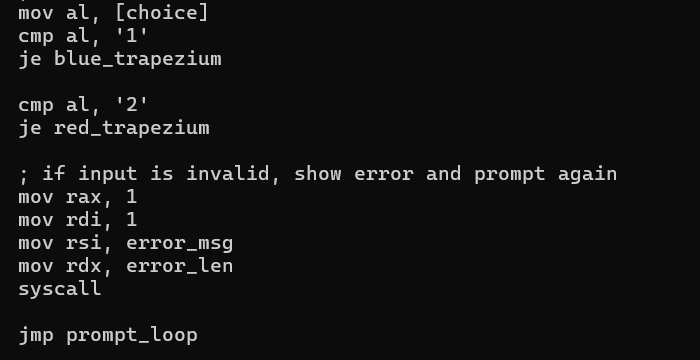
This reserves 4 bytes to store the user’s color choice (from the keyboard input).



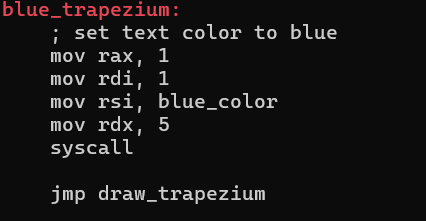
This is where the program begins running. **\_start** is the label that marks the entry point for the operating system to start executing the code.



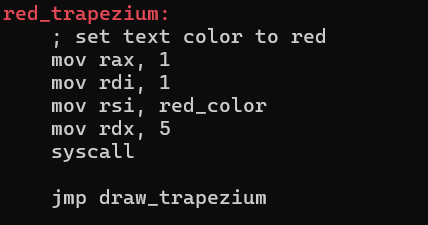
This shows the colour selection message. And waits for the user to type something.



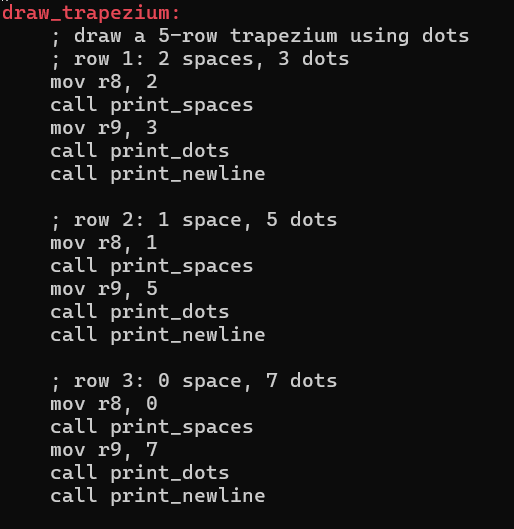
Checks the user's input from the choice variable. If the input is '1', it jumps to **blue\_trapezium**. If it's '2', it jumps to **red\_trapezium**. If the input is neither '1' nor '2', it prints an **error\_msg** to the terminal and then jumps back to **prompt\_loop** to ask for input again. This creates a loop that continues until valid input is received.



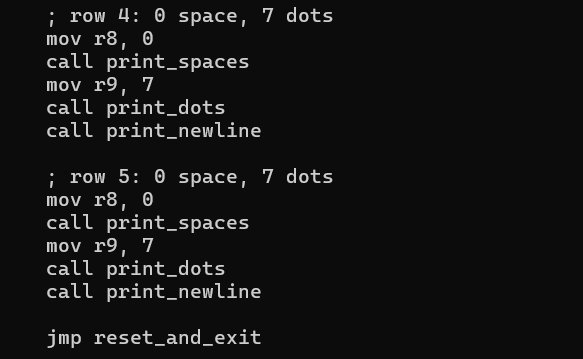
labeled **blue\_trapezium**, first sets the text colour to blue by writing a blue\_color code 5 bytes long to standard output using the sys\_write system call. Afterwards, it unconditionally jumps to the **draw\_trapezium** routine, presumably to render a **blue-colored** trapezium.



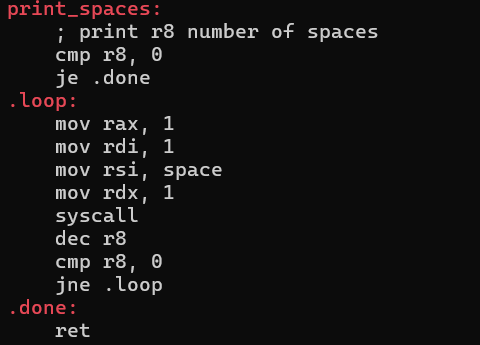
labeled **red\_trapezium**, first sets the text colour to red by writing a **red\_color** code 5 bytes long to standard output using the sys\_write system call. Afterwards, it unconditionally jumps to the **draw\_trapezium** routine, presumably to render a **red-colored** trapezium.



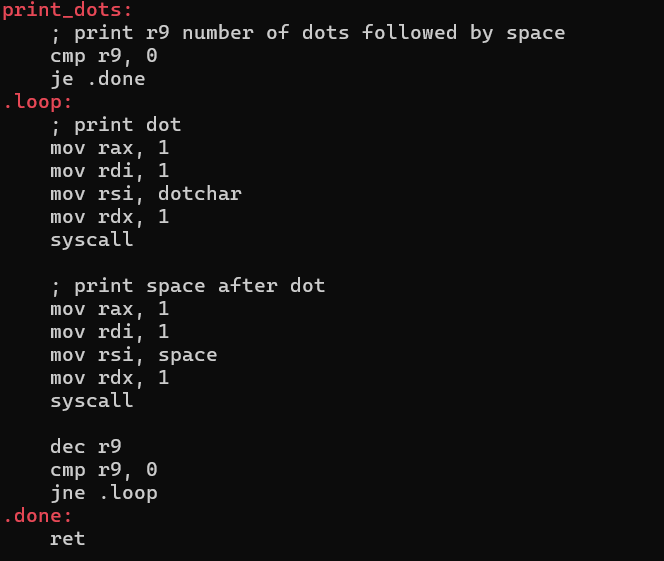
Within the **draw\_trapezium** routine, draws the first three rows of a 5-row trapezium using dots. It sequentially sets the number of spaces in r8 and dots in r9 for each row, then calls **print\_spaces**, **print\_dots**, and **print\_newline** to render them on the console.



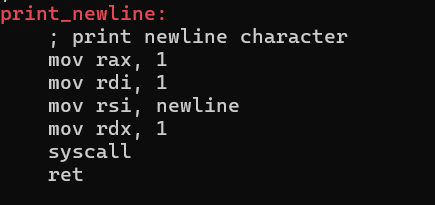
Draws rows 4 and 5 of the trapezium. For both rows, it prints zero spaces followed by seven dots, each on a new line. After completing the drawing, it jumps to the reset\_and\_exit routine.



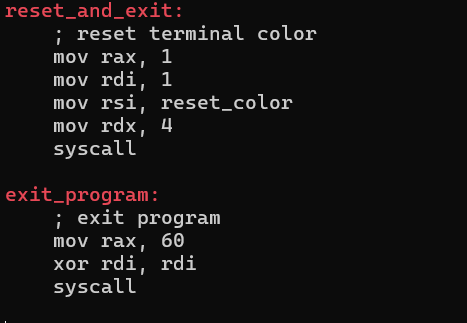
**Print\_spaces** prints a number of spaces equal to the value in the r8 register. It repeatedly calls the **sys\_write** system call to output a single space character until r8 becomes zero, then it returns.



**Print\_dots** prints a number of dots, each followed by a space, as specified by the value in register r9. It repeatedly uses the **sys\_write** system call to output a dot and then a space until r9 counts down to zero, at which point it returns.

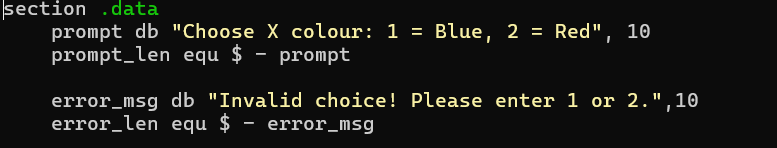


**Print\_newline**, prints a single newline character to the terminal. It achieves this by using the **sys\_write** system call the memory address of the newline character, and a length of 1 byte. After printing, it returns.

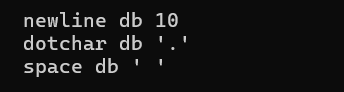


The code first resets the terminal color by writing a 4-byte reset\_color code to standard output. Immediately after, it exits the program by invoking the sys\_exit system call with an exit status of zero, indicating successful termination.

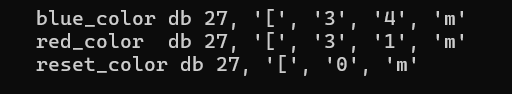
### **2.4.2** Alphabet X



These are the messages the program shows to the user. The first message asks the user to choose a colour. The second one warns the user if they type an invalid option.



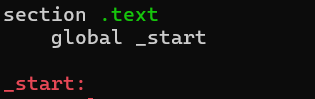
**newline** adds a line break. **dotchar** is the dot used to draw the “**X**”. **space** is used for spacing and shaping the “**X**”.



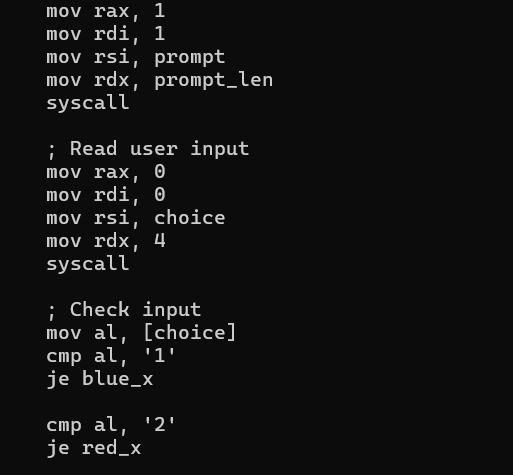
**blue\_color** makes the text blue. **red\_color** makes the text red. **reset\_color** changes the colour back to normal afterward.

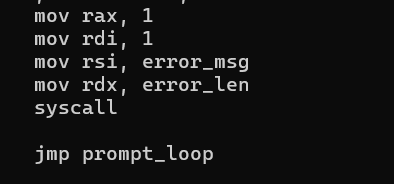


This reserves 4 bytes to store the user’s color choice (from the keyboard input).

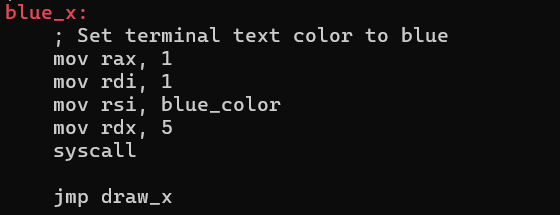


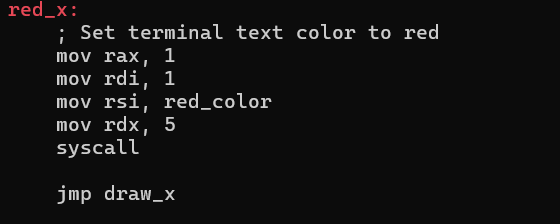
This is where the program begins running. **\_start** is the label that marks the entry point for the operating system to start executing the code.



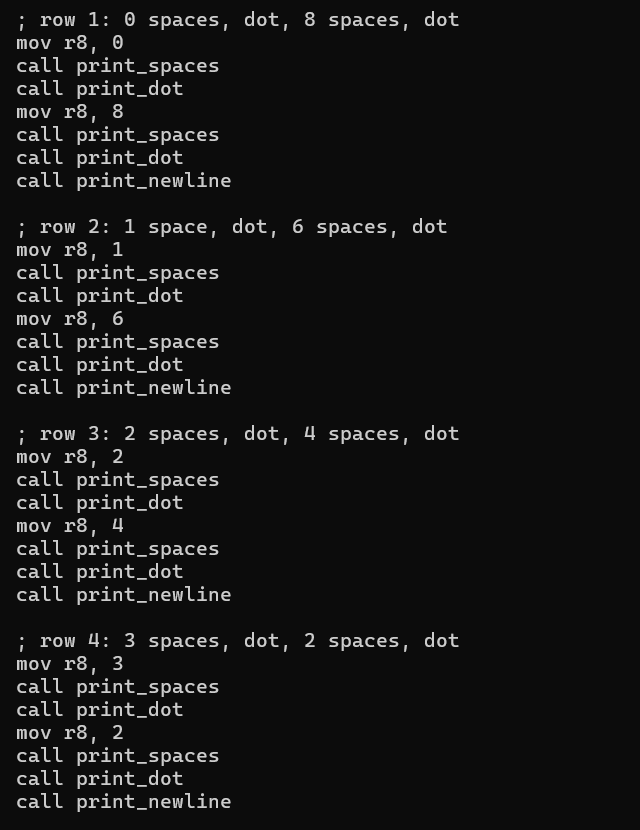


**Conditional execution** based on user input. After the user provides their choice, the code checks the first character they entered. If that character is '1', the program will **unconditionally jump** to a code block labeled blue\_x. If, instead, the character is '2', it will then jump to red\_x. This allows the program to branch to different parts of the code depending on the user's selection.

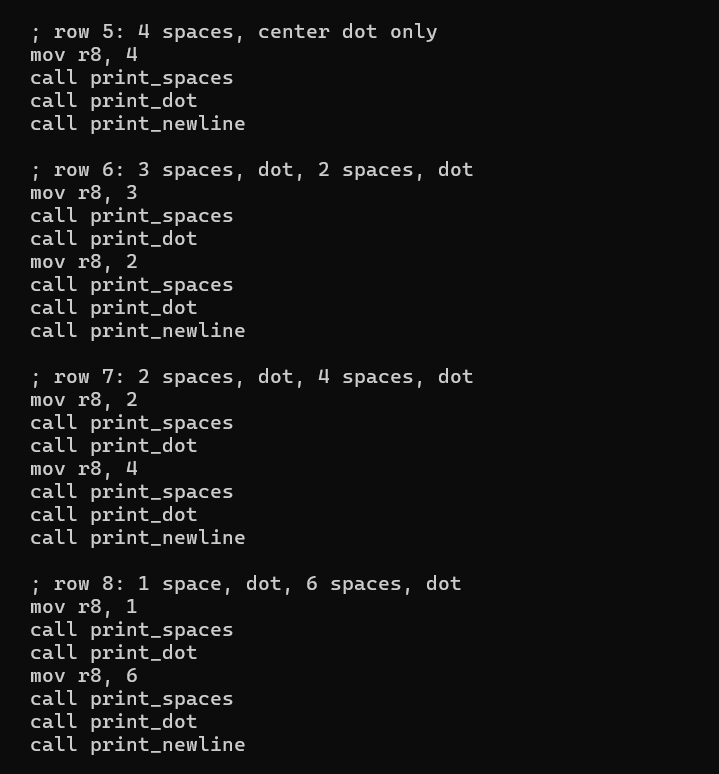




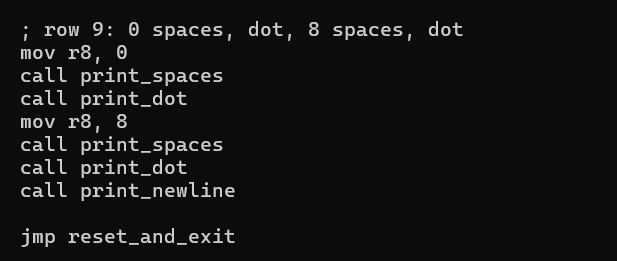
This will prompts the user for a choice, then validates their input to determine whether to draw a blue or red “X”. It achieves this by calling specialized routines to set the color, print varying numbers of spaces and dots line by line to form the shape, and finally resets the terminal color before exiting.



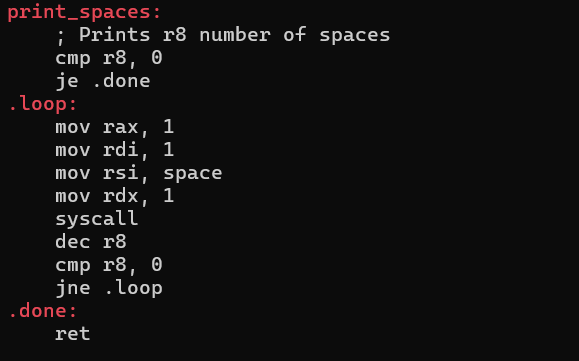
This code draws four rows of a pattern. Each row starts with a number of spaces, followed by a dot, then more spaces, and finally another dot, before moving to a new line. The number of leading spaces increases by one in each subsequent row, while the number of spaces between the dots decreases by two.



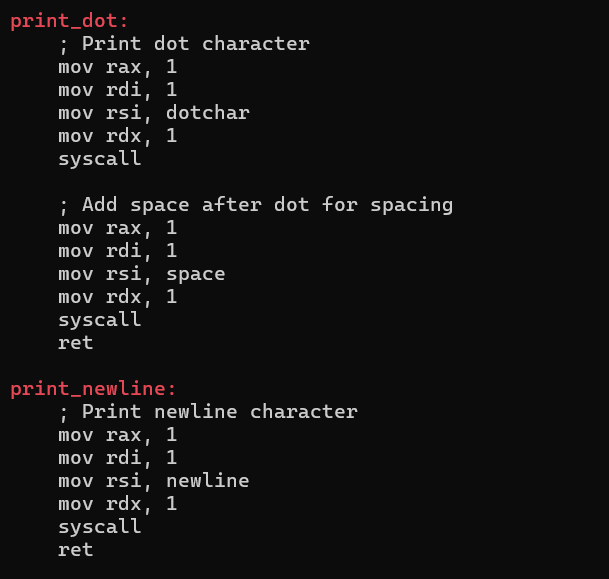
This code continues drawing the pattern. Row 5 has a single centered dot. Rows 6, 7, and 8 mirror the earlier rows in terms of dot and space placement.



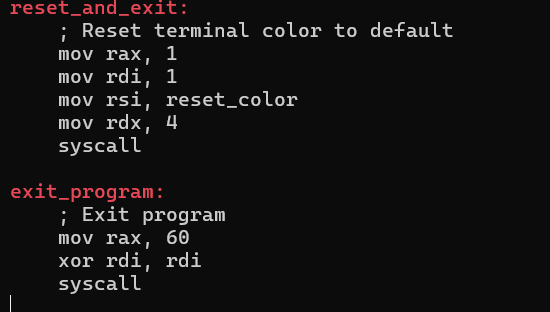
code concludes the pattern. Row 9 mirrors the very first row, starting with no spaces, followed by a dot, eight spaces, and another dot.



This code defines a print\_spaces function. It checks if the r8 register (which holds the number of spaces to print) is zero; if so, it exits. Otherwise, it enters a loop that prints a single space character using a system call, decrements r8.



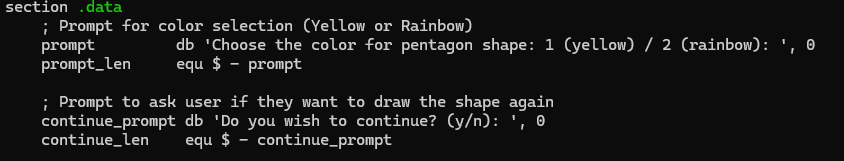
This code defines two functions: print\_dot and print\_newline. print\_dot prints a dot character using a system call, and then immediately prints a single space character for spacing purposes. print\_newline simply prints a newline character, also using a system call.



reset\_and\_exit attempts to reset the terminal color to its default using a system call, likely for cleanup. Following this, exit\_program performs a standard program exit using system call 60, with an exit code of 0.

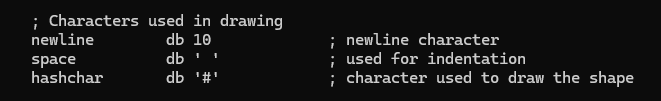
**2.5** Sultan Omar Abdulla Takrori

**2.5.1** Pentagon

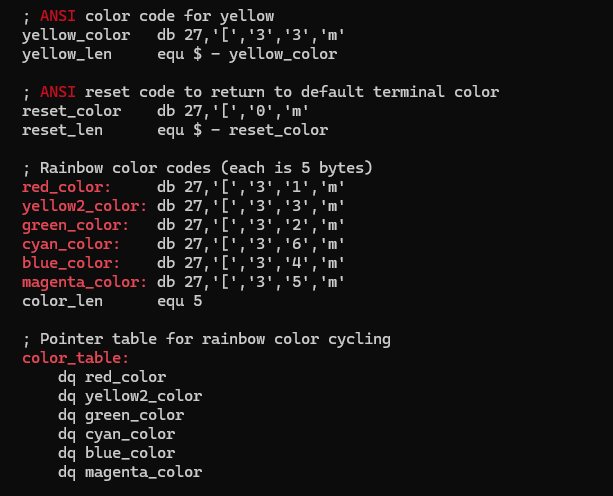


This part saves two messages that is shown to the user.

The first message asks the user to choose a color: 1 for yellow or 2 for rainbow. The second message asks if the user wants to draw again or exit. The program also saves the length of each message so it knows how much to print.

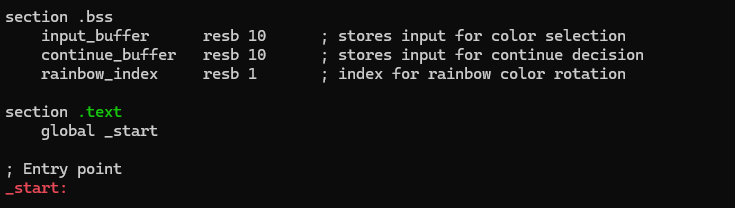


Newline is used to move to the next line. Space is used to add gaps so the shape is centered. Hashchar is the # symbol that is used to draw the actual shape.



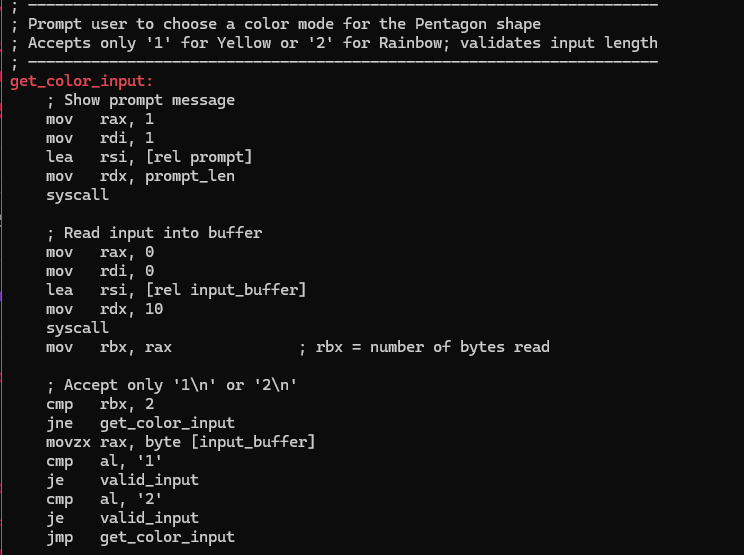
It starts with the yellow color, if the user chooses yellow mode. Then it sets a reset color to return the text back to normal after printing.

Next, it lists six rainbow colors: (red, yellow, green, cyan, blue, and magenta). These will be used one by one when rainbow mode is active. At the end, there’s a list called color\_table that stores the rainbow colors in order so the program can use them while drawing.

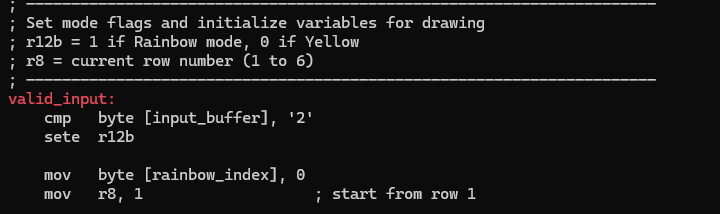


In the section.bss, input\_buffer is used to store what the user types when choosing a color. continue\_buffer stores the user answer when asked if they want to continue or exit. rainbow\_index keeps track of which rainbow color to use next.

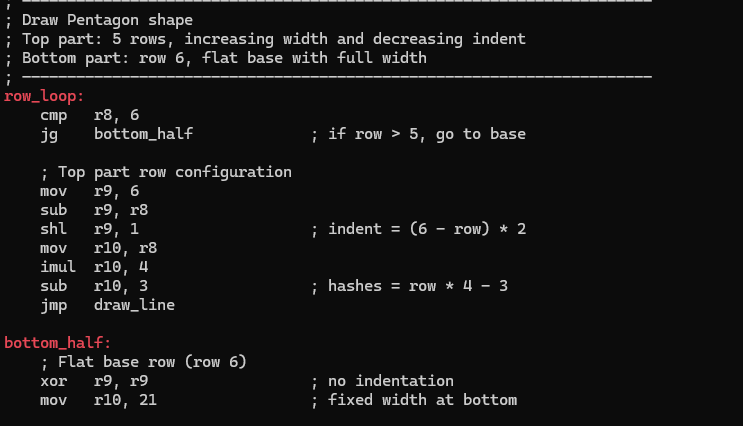
The .text section is where the code starts running. It marks \_start as the entry point, which is the first place the program begins.



User to choose a color for the pentagon shape and checks if the input is valid. First, it shows a message asking the user to pick a color: 1 for yellow or 2 for rainbow. Saves the choice in the input\_buffer. If it's '1' or '2', the input is valid and the program moves on. If it's anything else, it loops back and asks the question again.

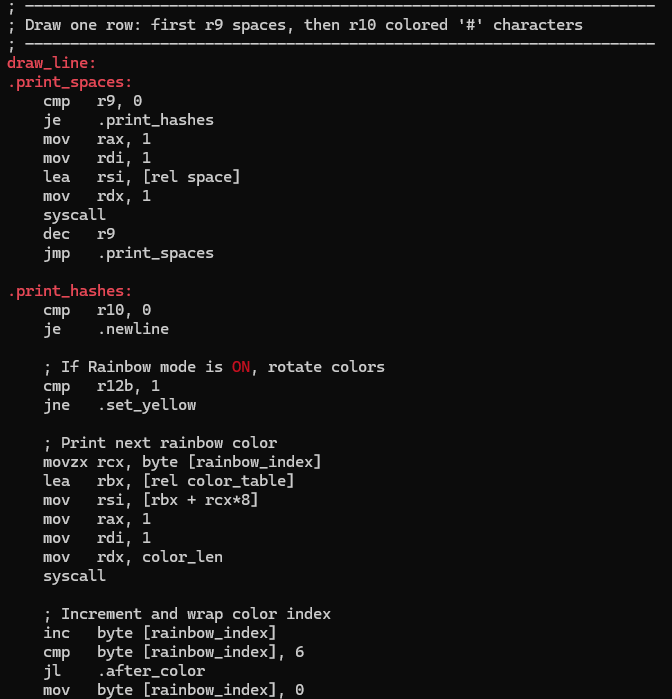


It checks if the user chose rainbow mode by looking for '2' in the input. If so, it sets a flag (r12b = 1). If the user chose '1', it keeps the flag at 0 for yellow mode. Then it resets the rainbow color index to 0, so the colors will start from the beginning. Finally, it sets the row number r8 to 1 to begin drawing from the first row.



If the current row number is 1 to 5, the program makes the top part of the pentagon. It calculates how many spaces to add before the # symbols and how many # symbols to print. The shape gets wider with each row.

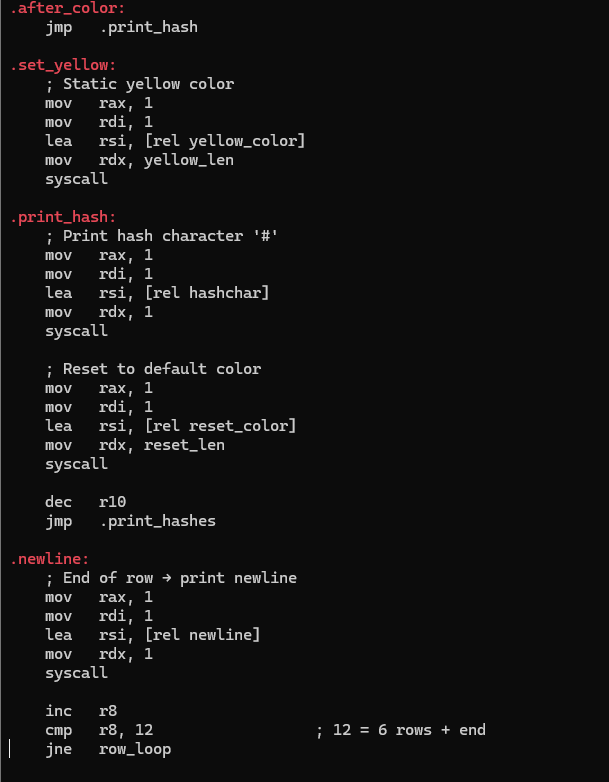
If the row is 6, it draws the bottom part. This row has no spaces and always prints 21 ‘#’ symbols to make a flat base.



Well first, .print\_spaces print the spaces using r9. It prints one space at a time until all spaces are done. This moves the shape to the right.

Then, .print\_hashes prints the ‘#’ symbols using r10. If rainbow mode is on, it picks a color from the list using rainbow\_index, prints the colored #, then moves to the next color. If it reaches the end of the color list, it loops from the beginning.

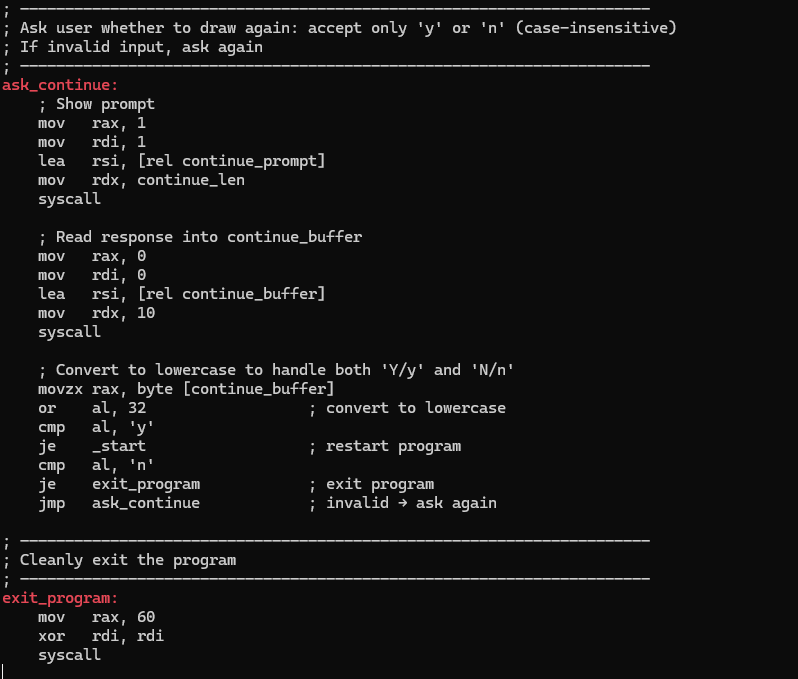
This prints each row with the right number of spaces and colorful # symbols.



Finishes drawing one row and moves to the next.

If the user choses yellow mode, it sets the color to yellow. Then it prints one ‘#’ symbol and resets the color back to normal. It keeps printing ‘#’ symbols until all are done for that row.

After that, it prints a new line to move to the next row. Then it increases the row number and checks if all 6 rows are finished. If not, it loops back to draw the next row.



Program asks the user if they want to draw another Pentagon or exit.

It prints the message asking ‘Do you wish to continue? (y/n):’. The response is saved in continue\_buffer.

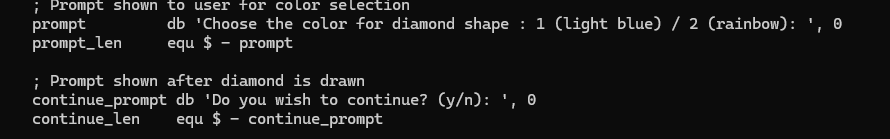
It reads the input and checks the first letter If they type y or Y, the program restarts. If they type n or N, the program exits. If they type anything else, it shows the question again until the input is valid.

At the end, exit\_program cleanly ends the program.

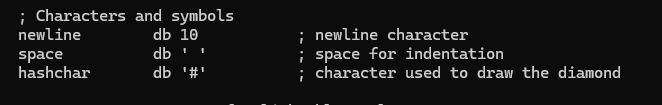
**2.5.2** Diamond



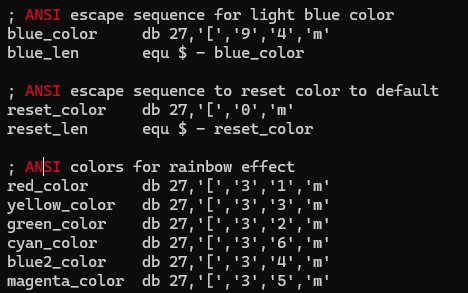
\_start is a command to start running the program.



This part shows messages that will appear on the screen for the user. The first message asks the user to choose the color. The second message asks if the user wants to draw another diamond after the first one is done.



Here, it sets up the basic characters needed to draw the diamond. newline is used to go to a new line, space is used to add empty space for shaping the diamond, and the character used for drawing the diamond is ‘#’.



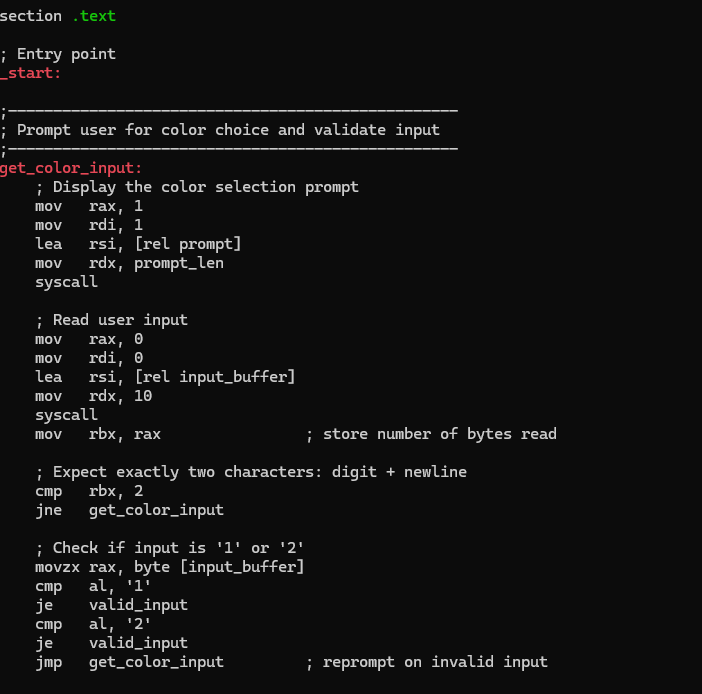
This part makes the diamond colorful. It uses a code for light blue, some codes for rainbow colors, and one code to reset the color back to normal after drawing.



The code sets up the rainbow colors and saves space for user input.

First, color\_table is a list of all the rainbow colors.It will go through the list one by one to make the shape colorful.

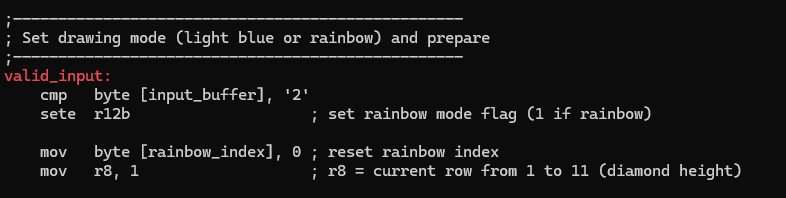
In the section.bss, the program reserves space in memory. input\_buffer is used to store what the user types when picking a color. continue\_buffer stores the user’s answer when asked if they want to draw again. rainbow\_index keeps track of which color to use next from the rainbow list. This helps the program cycle through the colors step by step.



This part of the program asks the user to choose a color and checks if the input is valid.

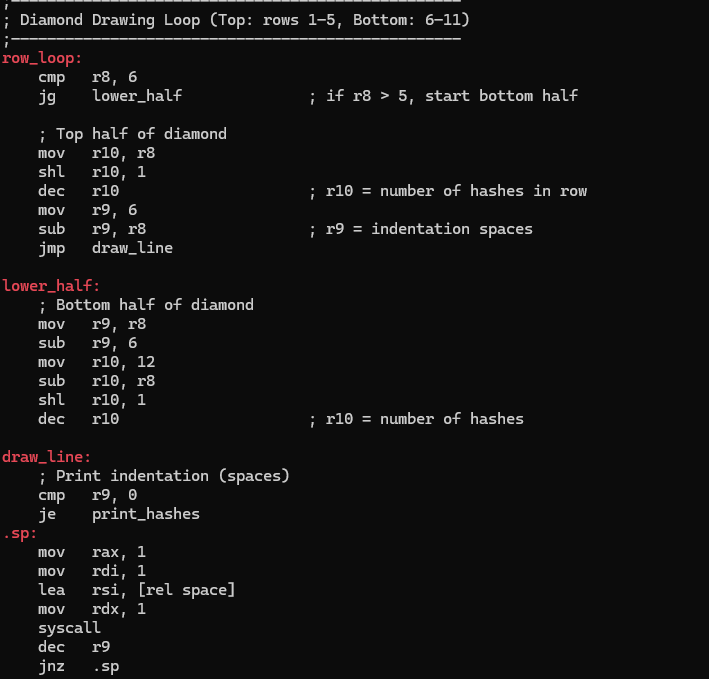
It first shows the message asking the user to choose between light blue or rainbow. Waits for the user to type something. The program only accepts 1 or 2 and a newline (Enter key). If the user doesn't type exactly two characters, it asks again.

Next, it checks if the user typed '1' or '2'. If the input is valid, it jumps to valid\_input. If not, **it loops** back and asks the question again. This loop makes sure the user can only continue by giving a proper answer.

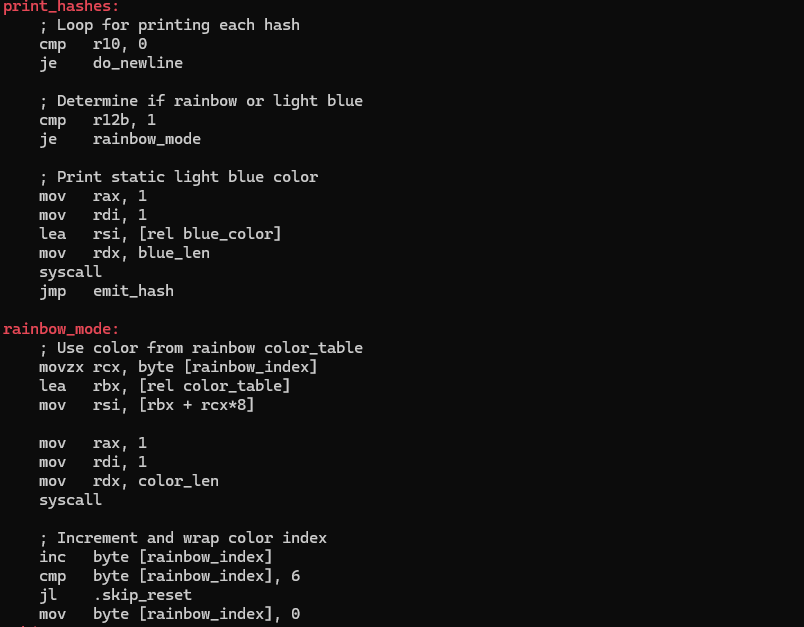


Gets ready to draw the diamond.

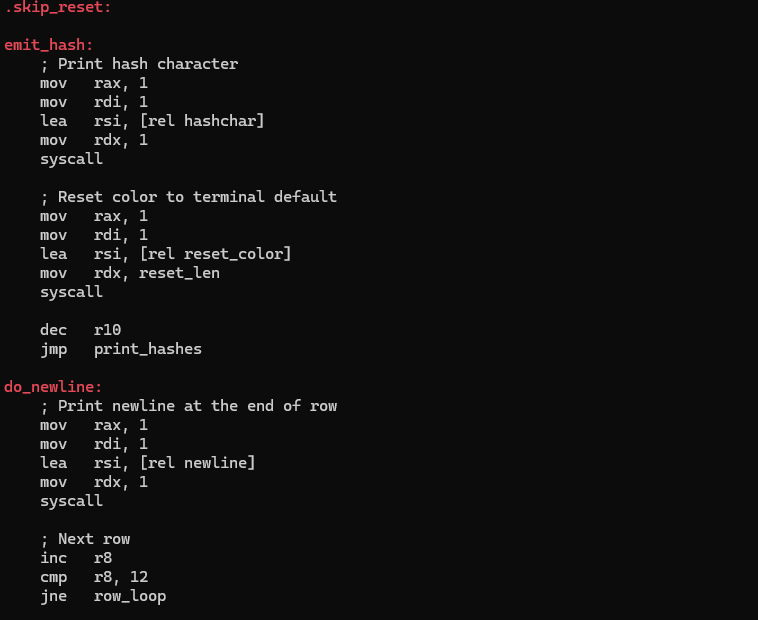
If the user picked rainbow mode (by typing 2), the program turns on the rainbow flag. It then resets the color counter to start from the first color. It also sets the row number to 1 so it can start drawing the diamond from the top**.**

****

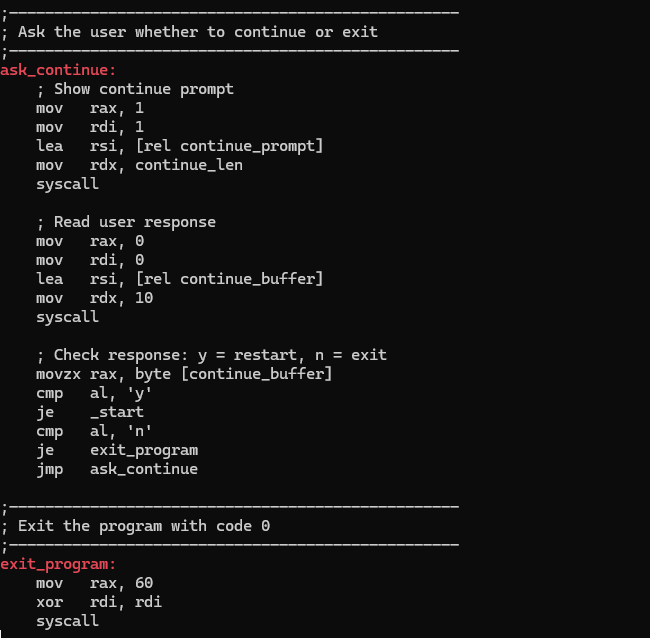
This shows the main loop of the diamond drawing, row\_loop. This part starts by checking whether the current row number r8 is in the top or bottom half of the diamond. If it’s row 1 to 5, it calculates how many spaces and hashes are needed for that row to make the top half of diamond shape. If the row number is 6 to 11, it jumps to the lower\_half section to calculate the spaces and hashes for the bottom diamond. After calculating, it jumps to draw\_line, which handles the actual printing.

****

As you can see it starts printing each row. It starts printing the spaces to make the diamond centred. It uses a small loop that prints one space at a time until all spaces are done. Then it starts printing the **#** symbols. Before printing each one, it checks if rainbow mode is on. If no, it sets the color to light blue. If rainbow mode is on, it picks a color from the rainbow list using the current color index. After printing the color, it moves to the next color. If it reaches the end of the rainbow list, it starts from the beginning.

****

This part prints the actual **#** symbol and then resets the color to normal so the next text won’t stay colored. It repeats this until all hashes for the row are printed. then, it prints a new line to move to the next row. Then it increases the row number and repeats the whole process for the next row until the diamond is finished.



Program asks the user if they want to draw another diamond or exit.

It shows the message:’Do you wish to continue? (y/n): It reads the input and checks the first letter. If the user types y, the program jumps back to the start and restarts. If the user types n, it goes to exit\_program and ends the program. If the user types anything else, it shows the question again until the input is valid.

The exit\_program part at the end simply ends the program using a system call, with exit code 0.

**2.6** Abdul Aziz Vayani

**2.6.1** Triangle

A screenshot of a computer screen

AI-generated content may be incorrect.

This section of the triangle program is tasked with establishing and choosing the color mode prior to any drawing activity. It begins in the .data section by specifying the newline, star (\*), and space characters utilized for illustrating the triangle. It also contains a prompt message that will appear to the user, requesting them to select a color mode — either magenta (choice 1) or rainbow (choice 2). ANSI color codes for red, yellow, green, cyan, blue, magenta, and the reset code are specified, each concluding with a null terminator to allow their use as strings. These color codes are subsequently used in a rainbow\_table, serving as a reference list for transitioning through colors in rainbow mode.

Within the .bss section, space is allocated for the user's input (input), a flag (rainbow\_flag) to determine the selected color mode, and an index (color\_index) to monitor the current position in the rainbow color cycle.

Execution of the program commences at the \_start label. Initially, it shows the prompt message requesting a color selection by utilizing the write\_string function. It then awaits the user's reply through the sys\_read system call. The input's first character is examined — if it is 2, the program turns on rainbow mode by updating rainbow\_flag to 1 and setting color\_index to 0, then proceeds to draw\_triangle to start the drawing. If the user inputs something different (such as 1), the program switches to magenta mode by assigning 0 to rainbow\_flag and promptly displaying the magenta color escape code with write\_string. This establishes the correct color prior to drawing the triangle.

A screenshot of a computer screen

AI-generated content may be incorrect.

This part of the program is tasked with printing the triangle shape, one line at a time. The triangle consists of 11 rows, and the loop starts with the row counter r8 initialized to 1. The label .row\_loop indicates the beginning of the loop, and the program verifies whether r8 has attained 12. If it has, it indicates that all 11 rows have been displayed, and the program will proceed to reset\_color to tidy up and terminate. For every row, it initially determines the number of leading spaces required to center the triangle by subtracting the row number from 11 and saving the outcome in r10. The .print\_space loop subsequently outputs that number of spaces, individually.

After the padding is printed, the program proceeds to the .print\_hash section. In this section, it determines the count of stars (\*) to display using the equation 2 \* r8 - 1, guaranteeing that every row contains an odd total of stars and that the triangle grows symmetrically. The .loop\_hash loop displays a single star at each iteration. Prior to printing each star, the set\_color function is invoked by the program to assign the correct color — either magenta or one of the rainbow hues based on the user's previous input. Following the printing of each star, it promptly invokes reset\_color\_code to guarantee that solely the star is colored and the subsequent output does not take on that color.

Once all the stars in a sequence are displayed, the .newline part generates a line break to proceed to the next row. The counter r8 for rows is then increased, and the loop continues until the triangle is finished. This reasoning creates a vivid, centered triangle that expands uniformly from row to row.

A screen shot of a computer

AI-generated content may be incorrect.

This section of the program guarantees a tidy and correct termination after the triangle is rendered. The reset\_color label is invoked after the drawing loop finishes. It initially invokes the reset\_color\_code function to restore the terminal text color to its default, ensuring that subsequent terminal output remains unaffected by the triangle’s color. It then makes a system call with sys\_exit (system call number 60) and an exit status of 0, indicating that the program terminated successfully.   
  
The write\_string function is a versatile tool that outputs any null-terminated string stored in the rsi register. It functions by initially tallying the characters in the string until it finds a null byte (0). It saves this count in rdx and then employs the sys\_write system call to display the complete string on the terminal. This function is utilized across the program to show prompts, implement color codes, and restore text formatting.   
  
The set\_color function handles the application of the correct color before outputting each star in the triangle. It initially verifies whether rainbow\_flag is equal to 1, signifying rainbow mode. If that’s the case, it retrieves the current color pointer from the rainbow\_table with the color\_index, then calls write\_string to use that particular color. Once a color is printed, it increases the index to proceed to the subsequent color for the following star. When the color list reaches its end (after 6 colors), the index is reset to 0, allowing the rainbow cycle to repeat. If rainbow mode is disabled, the program proceeds to .use\_magenta, where it just applies the solid magenta color using the identical write\_string routine.

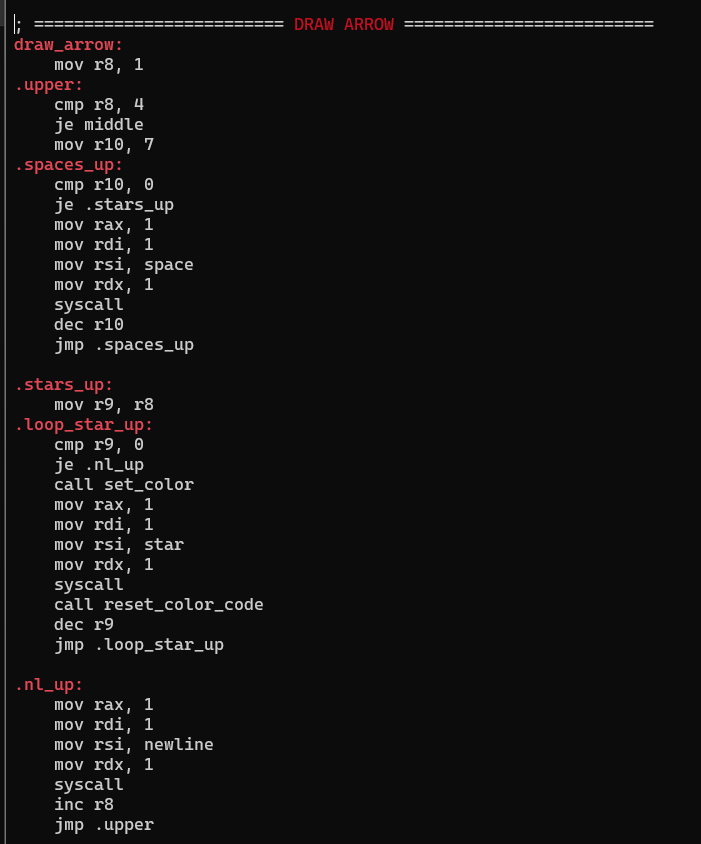
Finally, the reset\_color\_code function is a simple utility that restores the terminal's text formatting back to the default color. It loads the reset ANSI string into rsi and calls write\_string. This is called after every star is printed, so that only the stars are colored — ensuring no unintended coloring spills into spaces, newlines, or other characters.

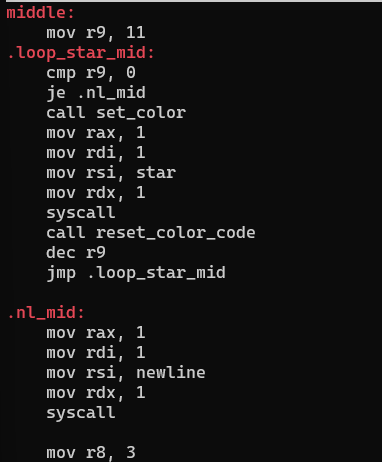
**2.6.2** Right Arrow

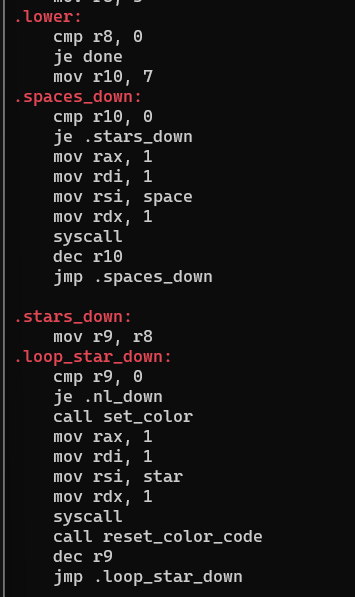
A screenshot of a computer program

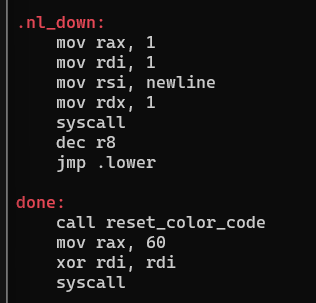
AI-generated content may be incorrect.

This part of the code begins by outlining all the information required for user engagement and color display. It begins in the .data section by initializing characters like the newline (\n), the star (\*) for shape rendering, and the space character for padding. It features a prompt message requesting the user to select between two color choices: 1 for magenta or 2 for rainbow mode. Subsequently, the program specifies ANSI escape sequences for different colors — red, yellow, green, cyan, blue, and magenta — each ending with a null byte allowing them to be displayed as strings. Additionally, there is a reset color sequence that brings the terminal back to its original color after printing, along with a distinct magenta string utilized for solid magenta mode. These color addresses are subsequently saved in a rainbow\_table, serving as a collection that the program can iterate over to implement rainbow coloring.   
  
The .bss section allocates memory for three elements: input, which holds the user's color selection; rainbow\_flag, which indicates if the user activated rainbow mode; and color\_index, which iterates through the rainbow color array as required.   
  
The program subsequently reaches the \_start label, marking the point where execution initiates. It shows the prompt requesting the user to select a color by invoking the write\_string function. It subsequently pauses for user input by employing the syscall instruction with sys\_read. The initial character of the input is compared to '2'. When the user inputs 2, the program activates rainbow mode by assigning rainbow\_flag a value of 1 and setting color\_index to 0 to commence with the first rainbow color. It subsequently proceeds to draw\_arrow, where the actual illustration commences. If the input differs from 2, the program resorts to magenta mode, assigns 0 to rainbow\_flag, and promptly outputs the magenta color with write\_string.









This segment of the code handles the creation of the complete right arrow shape, which includes three components: the upper point, the central bar, and the lower tail. The procedure starts at the label draw\_arrow, initializing the row counter r8 to 1. The .upper loop manages the initial three rows, with each row featuring a constant 7 spaces succeeded by a rising quantity of stars — 1, 2, and then 3. The loop initially outputs spaces with .spaces\_up, followed by stars using .stars\_up, invoking the set\_color function before each star to implement the chosen color (either magenta or a rainbow hue), and calling reset\_color\_code afterward to prevent color leakage. Once the stars are printed, a newline follows to proceed to the next row, and the row count is increased.

After the tip is completed, focus shifts to the center section, where a single line of 11 stars is displayed in a straight line. This creates the broadest section of the arrow. The loop .loop\_star\_mid outputs each star in the correct color, and afterwards shifts to the next line.

Following the center bar, the software gets ready to render the bottom tail using .lower. This section reflects the top part but in reverse order — the row counter begins at 3 and goes down to 1. Similar to before, it outputs 7 spaces followed by a decreasing sequence of stars (3, 2, 1), each colored using the identical dynamic technique. The loop concludes by outputting a newline and reducing the row counter until the tail is complete. Ultimately, the program arrives at the done label, where it resets the color for the final time and exits gracefully using the sys\_exit system call.

A screenshot of a computer program

AI-generated content may be incorrect.

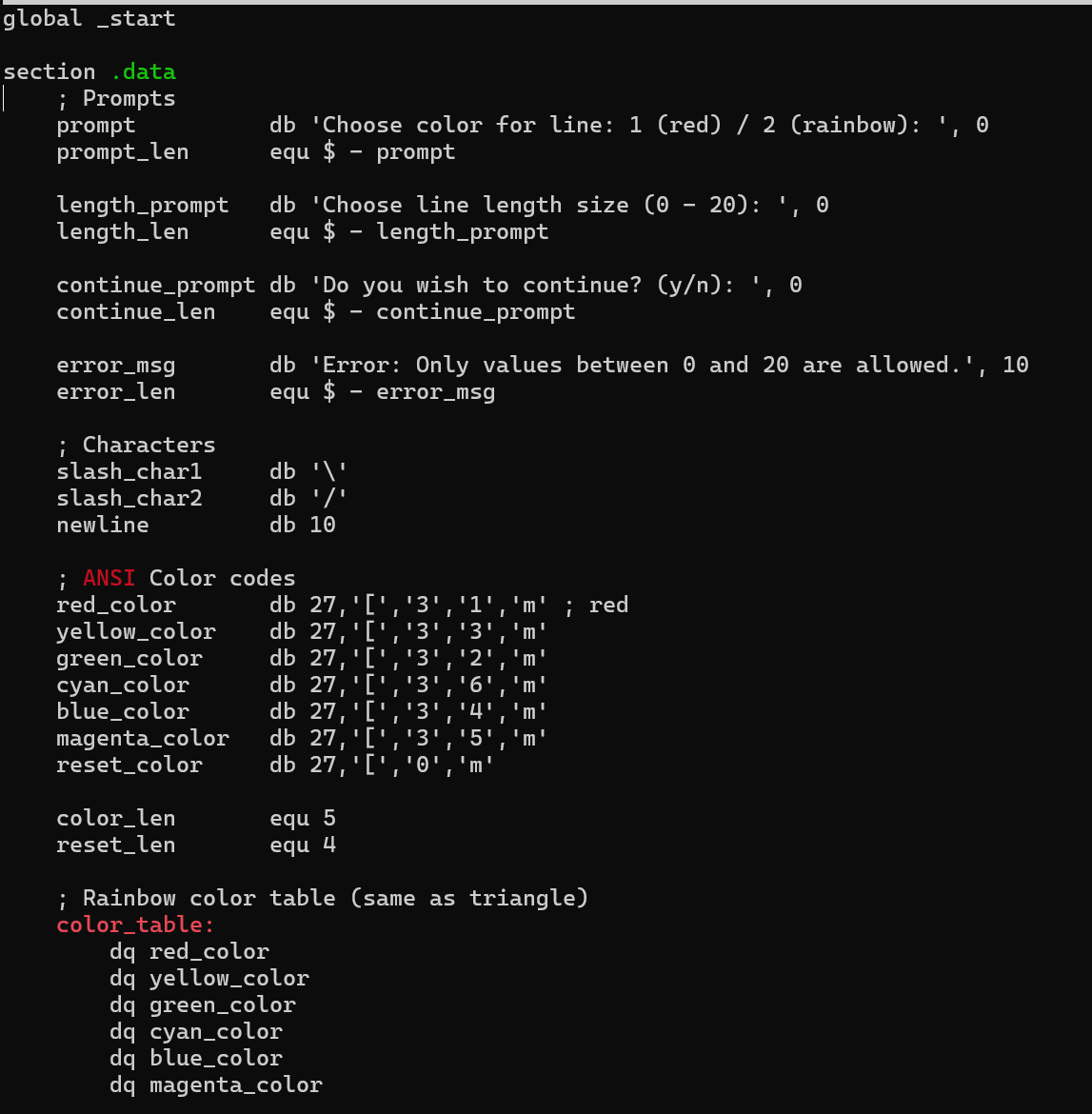
This section of the program establishes utility functions that are utilized during the arrow drawing procedure. The initial function, write\_string, receives a null-terminated string located in the rsi register and displays it on the screen through the sys\_write system call. It starts by measuring the string's length, byte for byte, until it encounters the null terminator. It subsequently reinstates the initial pointer, establishes the proper length in rdx, and executes the write operation.

The set\_color function comes next, applying the chosen color prior to printing each star. It initially verifies the value held in rainbow\_flag. When the flag is equal to 1, rainbow mode is enabled. The function subsequently retrieves the current color\_index, applies it to choose the matching color string from the rainbow\_table, and displays that color via write\_string. Following every use, the index advances, resulting in the next star displaying a distinct color. After all six colors are utilized, the index returns to zero to repeat the cycle. If rainbow mode isn't enabled, the program goes to .use\_magenta, where it merely outputs the magenta color string.

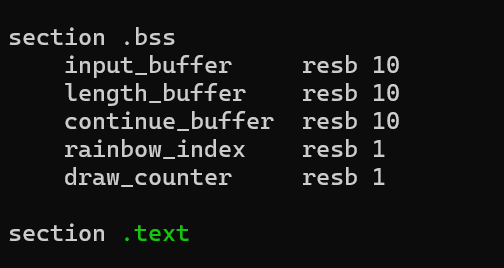
The ultimate utility function is reset\_color\_code, which returns the terminal text color to its default after displaying a star. It places the reset string into rsi and invokes write\_string to display it. This guarantees that each colored star is printed separately without influencing the subsequent character or line.

**2.7** Abdallah Mohamed Mahmoud Mohamed Mahmoud

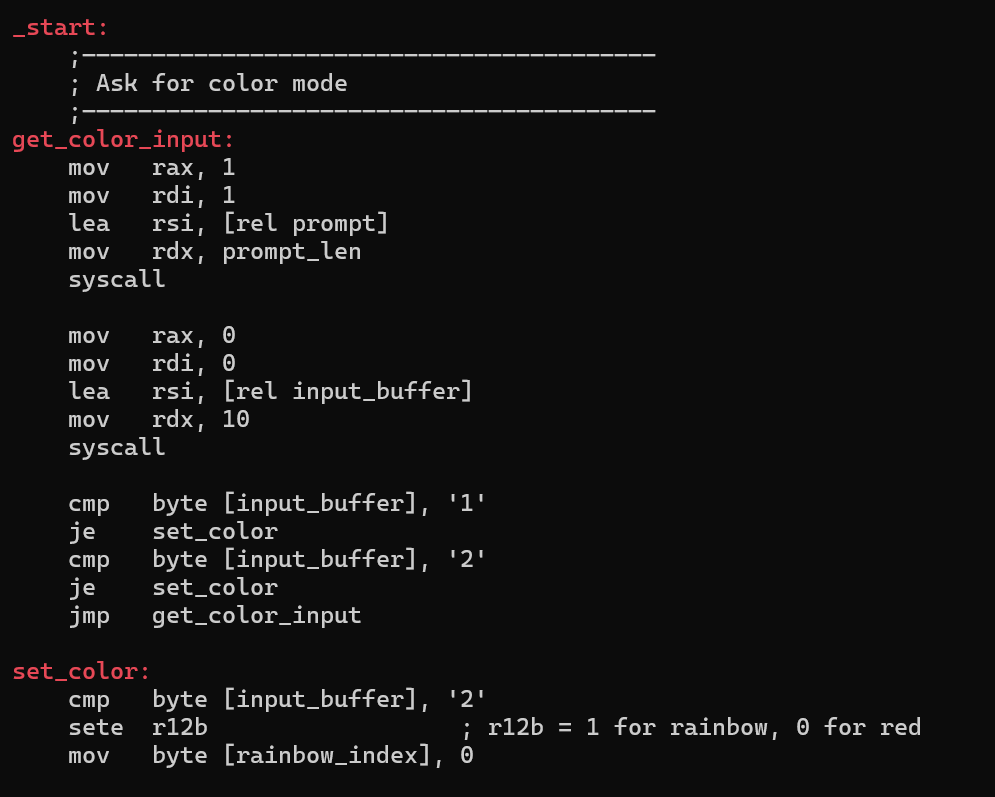
### **2.7.1** Line



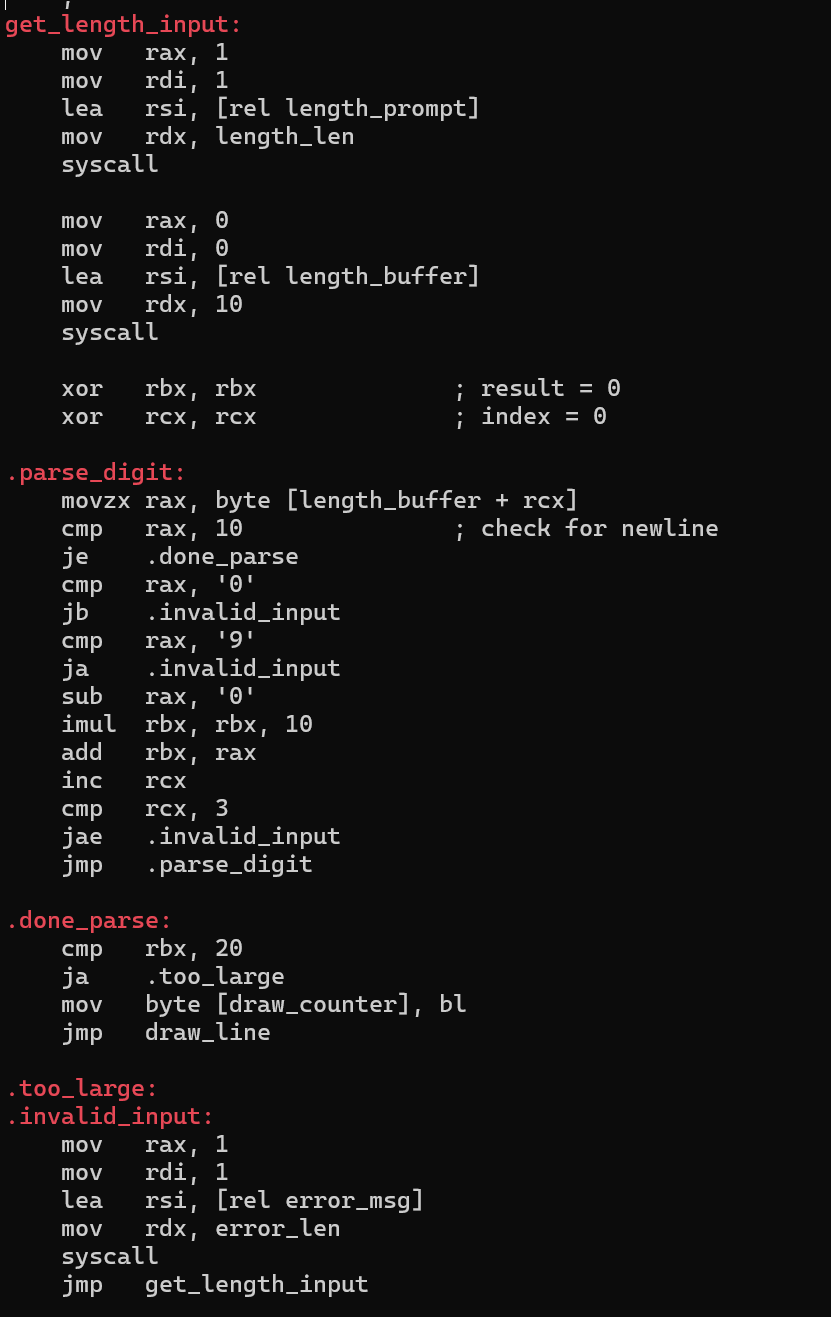
This is a command-line interactive program that asks the user to define and draw a line again and again. It directs the user to select a color of a line (red or rainbow) and a length (0-20 characters). The line is then given by the program with replacement of slash characters (\ and /). Once drawing has been done it gives an option of continuing to the user to play around again, and this takes them back to the start of the game should the user choose to answer with y, or end the game should the user choose n, or an incorrect input is made. Two functions of utilities take care of displaying text, keyboard reading, conversion of numbers, and work with terminal colors.



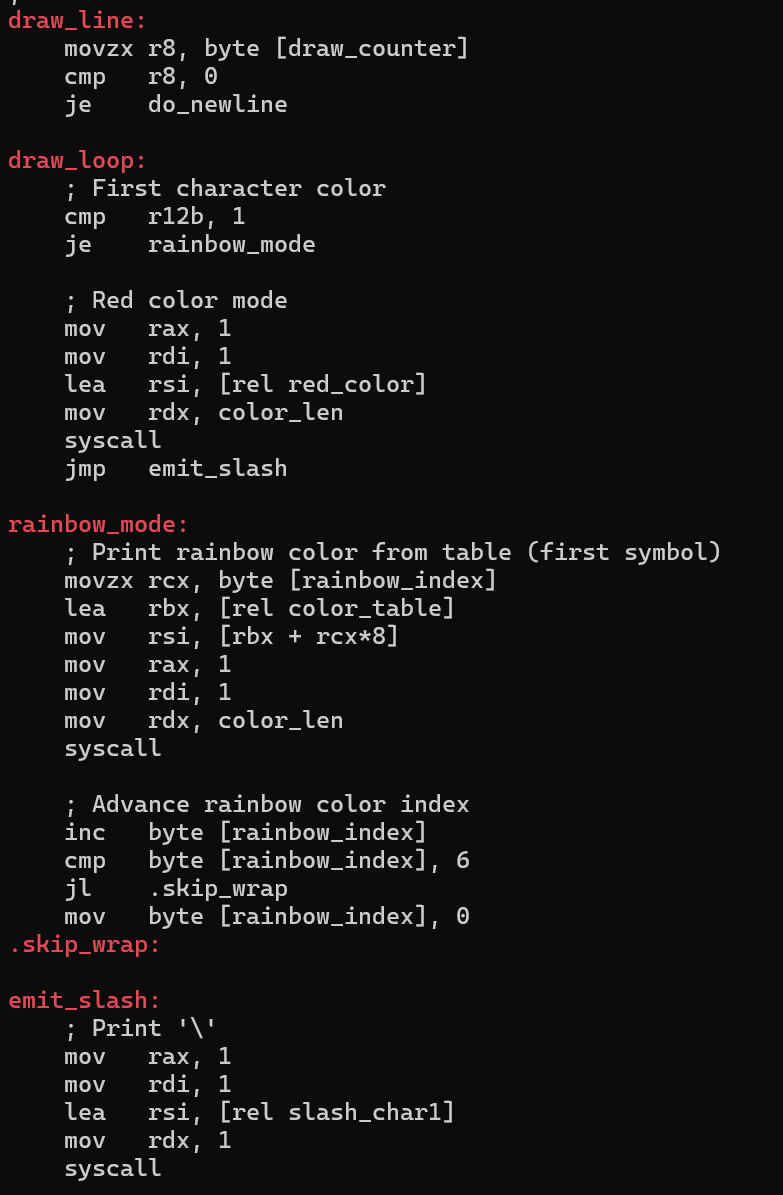
This part will save memory that will be used in running a program. The variables that will be saved here will be input\_buffer, length\_buffer and continue\_buffer which will be used to save user input. The variables rainbow\_index and draw\_counter will also be saved to use as a general counter during drawing process.

  
get\_color\_input and set\_color

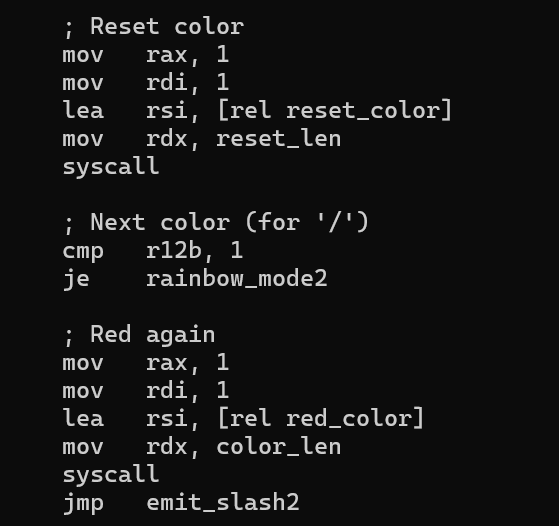
This block has an instruction to the user to choose a color mode (1 red, 2 rainbow). It reads the input along with the user, and checks its validity and repeats asking again till the user does provide something valid. When there is a valid decision registered, it resets r12b to 1 when colored rainbow is chosen, 0 when colored red, and rainbow\_index = 0 so that it loops the colors.



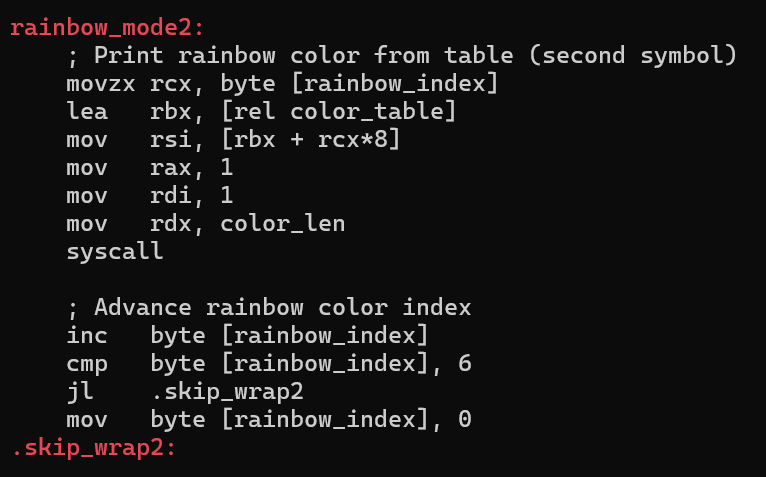
This program interatively asks the user to define and draw a line. It then requests a color option (red or rainbow), with proper validation of the input, then requests a line length which is between 0 and 20, with a conversion of ASCII to integer and fierce validation to exclude nonnumeric characters and the required range; an error message is shown and the prompt would restart in the event of giving an invalid input. There are memory locations set aside in the.bss section to store these user inputs and loop counters (accepting the rainbow colors, in rainbow indexing and drawing) After the successful validation of the input and depending on the color and length of the line to be drawn, the program is now ready to draw the line and lastly prompts the user asking whether he or she wants to continue in which case loop happens or exit occurs depending on the answer given by the user.



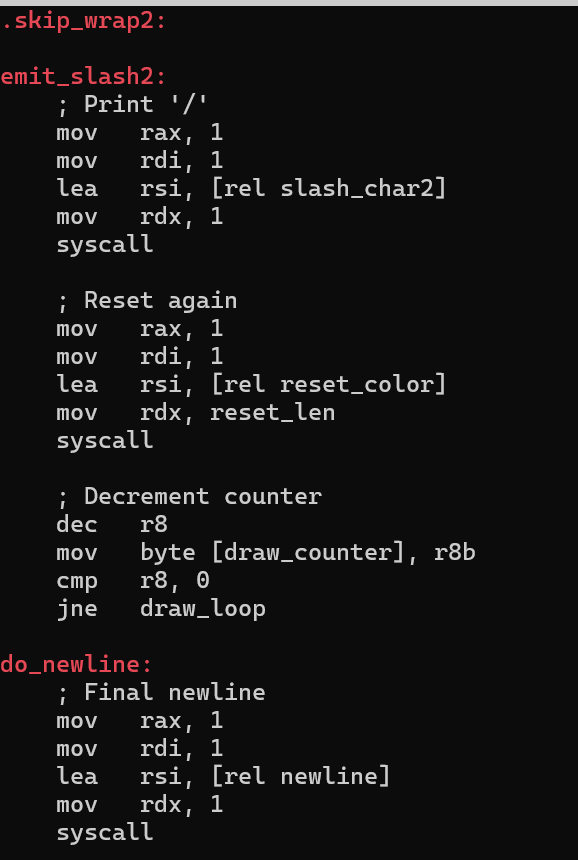
This procedure is in charge of printing a colored slash character (\) as a piece of a line drawing. It checks the value of draw\_counter denoting the number of remaining slashes to be drawn, then on reaching 0 the program goes to the newline routine. In case it is decided that drawing should proceed it tests the color mode selected in r12b. When in red mode it is used to place the ANSI escape code for red on standard out. Alternatively, in case rainbow mode is chosen, it reads the current color index at a given state in rainbow\_index, gets the color code as per a set color\_table and displays it on the screen. Once the color is printed it adds one to the index, sets it back to zero though after the last entry on the color list (6) is passed so there is a loop through the rainbow colors. Then, after printing the right color code in either mode, the program prints one character slash (character \) to print a part of the drawn line. This is done until the whole line is drawn.



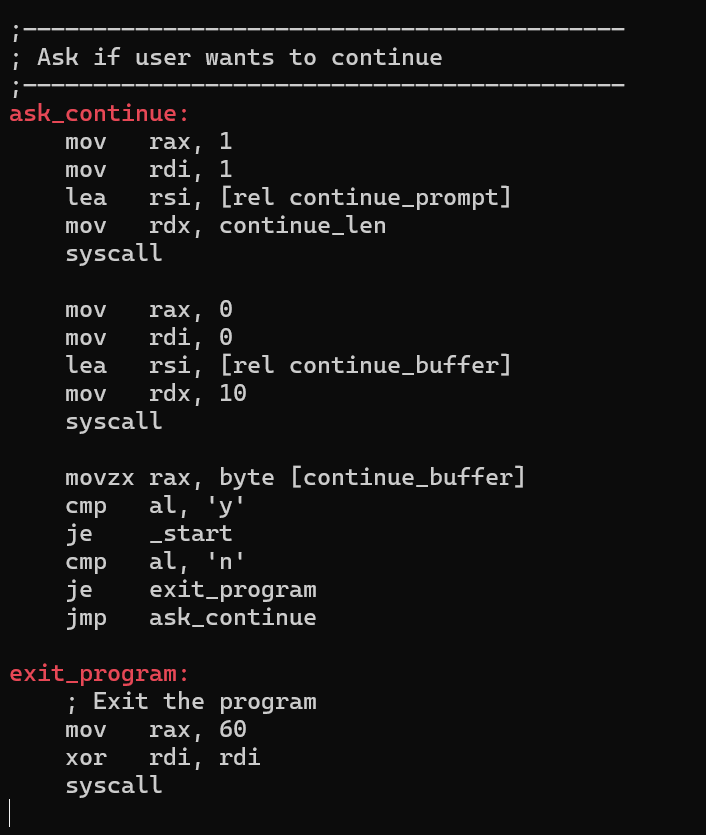
This part of the program is preparing to print the second character of the slash (/) symbol in the line and before that there is the need to reset the color of the text being displayed in the terminal. It does this by printing the ANSI reset color code to standard output so that any subsequent character will begin with a pristine color. Once the reset is done, the program sees the color mode once again through r12b. In the event of rainbow mode, it breaks to rainbow\_mode2 label to show the next color on the rainbow. Without this, it stays in red mode and writes the red-color-escape sequence of bytes again to standard output, but this time it will print the second character in red. After the proper color code has been written, control goes to emit\_slash2 to deal with the actual printing of slash character.



This part of the code is concerned with printing the color of second character when rainbow mode is enabled. It starts by retrieving the current color index present in rainbow\_index and using the index to fetch the color escape code present in the color\_table. A syscall is used to write this color code to the standard output with the next slash formatted using the colors we wanted. Once printed the program increases the value of rainbow\_index to store the next color of the sequence. When the index arrived at the final possible value of 6 (the number of rainbow colors), the value rolls over to 0 so that the rainbow colors will again cycle through unhitch whenever needed in the future.

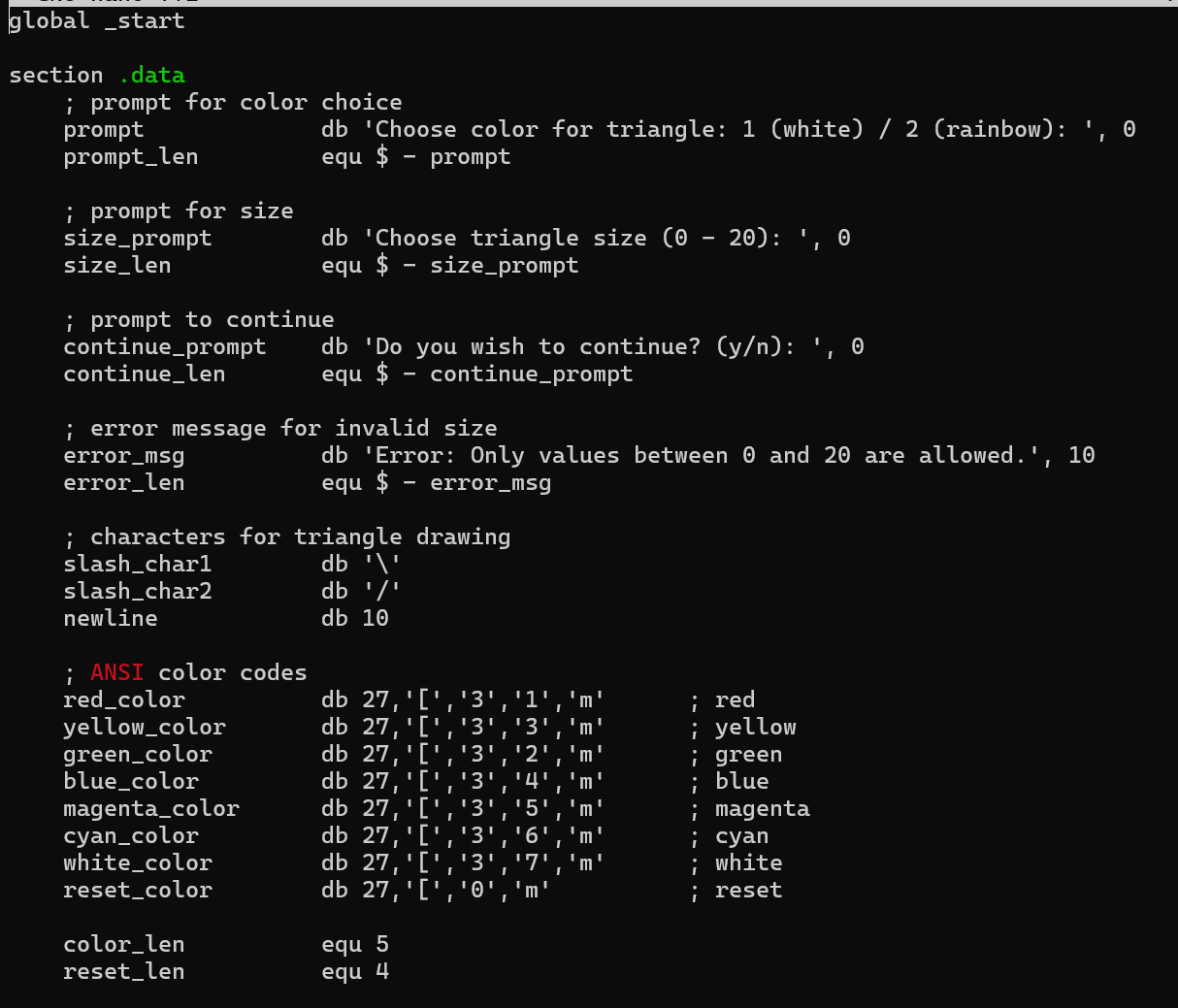


The first line of this section is .skip\_wrap2, where the skipping of the reset of the rainbow index is redirected should it have been unnecessary. It is then seen to emit\_slash2, in which the second slash character (/) is output on the screen via a standard output syscall. Printing Complete, the program restores the text default text color by writing the ANSI reset escape code, so no color formatting leaks into later output. It then uses the drawing counter r8, decrements it, changes the draw\_counter variable in memory, and also checks whether there are still some characters to draw. In that case it goes back into drawloopt, to start again. Otherwise it continues to do\_newline, writing a newline character to the screen causing that the cursor moves to the next line, which marks that the current line is currently rendered.

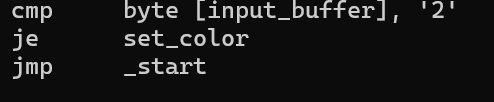


This part of the program handles the user prompt that asks whether they want to continue drawing another line. It begins by printing the continue prompt message to standard output using a syscall. Then, it reads up to 10 bytes of user input into continue\_buffer. After capturing the response, it checks the first character the user entered: if it's 'y', the program jumps back to \_start, restarting the entire drawing process; if it's 'n', the program jumps to exit\_program to terminate. If the input is anything else, the prompt is displayed again by looping back to ask\_continue, ensuring only valid inputs (y or n) are accepted. In the exit\_program section, the program cleanly exits by invoking the exit syscall (60) with return code 0, ending execution.

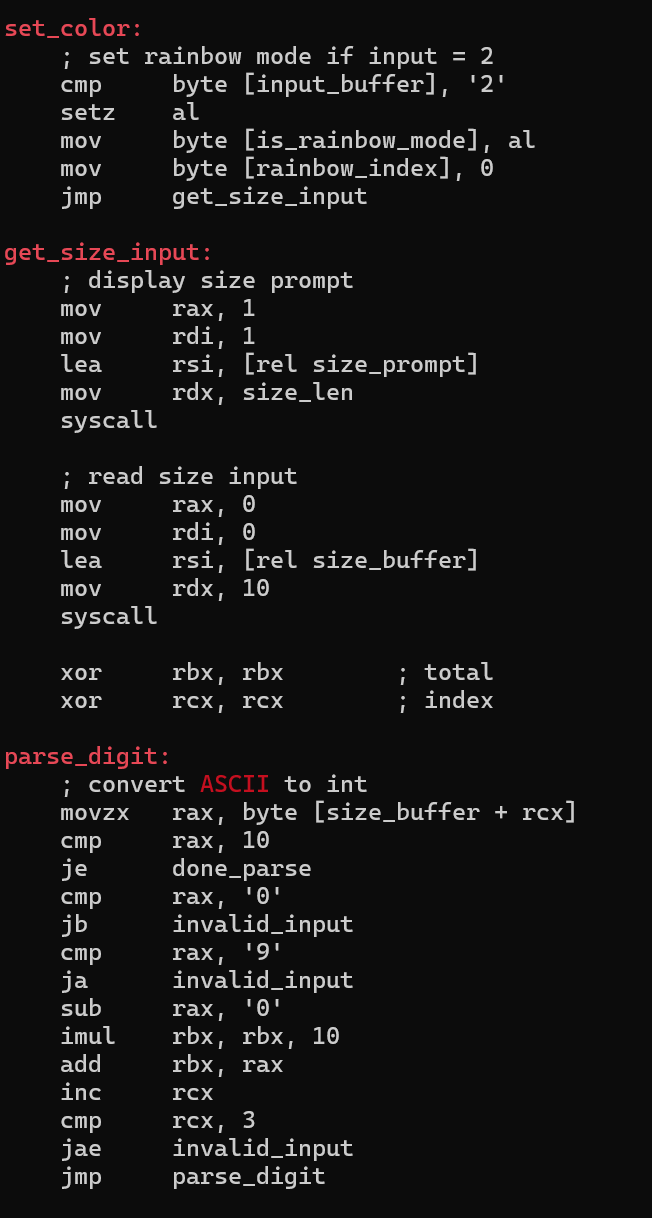
### **2.7.2** Right-Angled Triangle

All the data that will not change during the program (such as prompts, color codes, and the characters of a triangle draw) are mentioned in this section. It begins with a prompt that puts a request to the user to specify a color scheme of the triangle (white and rainbow), and it uses equ to compute the length of the prompt. The second prompt requests entry of the triangle size which has a range of 0 to 20. It also has a prompt to inquire the user on whether he or she wants to continue and a string providing an error message that is displayed when the user inserts an invalid size of a triangle. The slash characters backslash and forward slash used in drawing the triangle are stored in the slash\_char1 and slash\_char2 variables respectively and newline variable stores a line break character. A terminal color code set of ANSI escape codes specifies the terminal color used in the output and these are red, yellow, green, blue, magenta, cyan and white; and ANSI escape code sequence to restore the text to its default formatting. The constants color\_len and reset\_len give the lengths in bytes of the color and reset codes to be used in write system calls. All the writings and formats required to interact and visually present data to the user can be obtained through this data section.

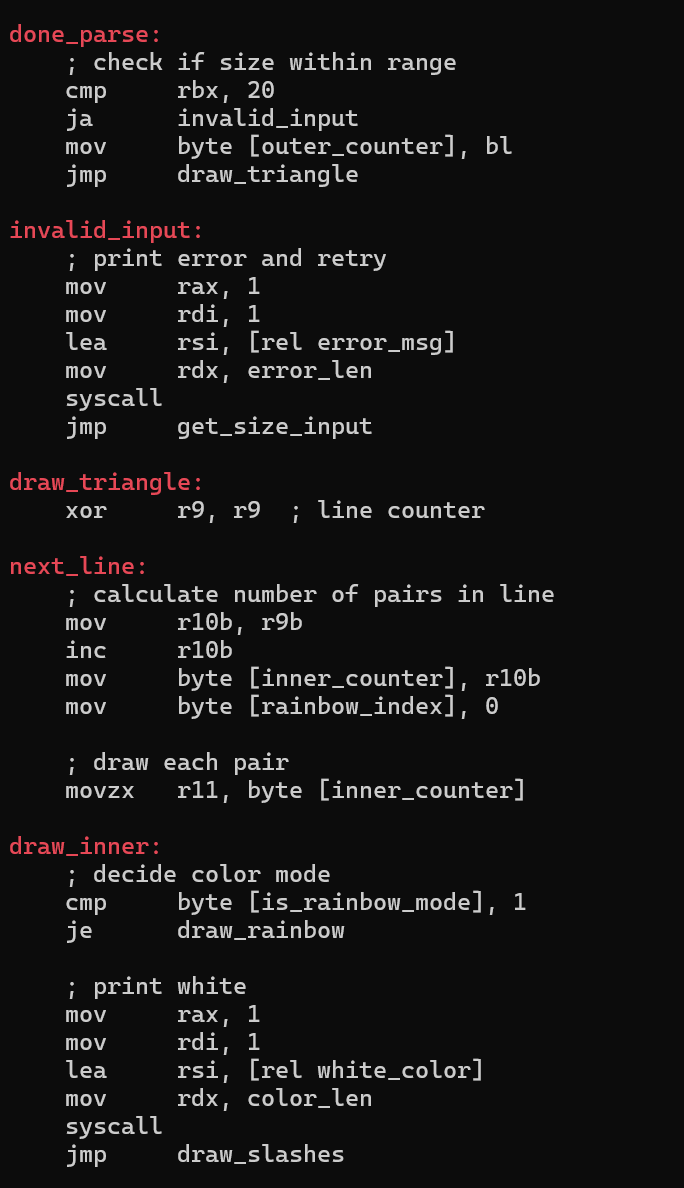




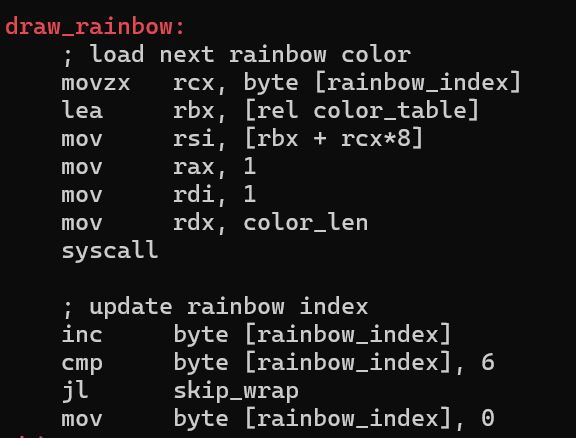
The current section determines data that the program utilizes. It has prompts on selection of triangle color, size, as well as whether to continue. It also has an error message that gives invalid input. The drawing of the triangle is done using characters \ and / and a line break is inserted with the usage of a newline character. ANSI color definitions exist on red, yellow, green, blue, magenta, cyan, white and reset. The length of each color byte is saved so that it could be printed. These values are applied in course of user interaction, coloring output and in drawing the triangle in the screen.



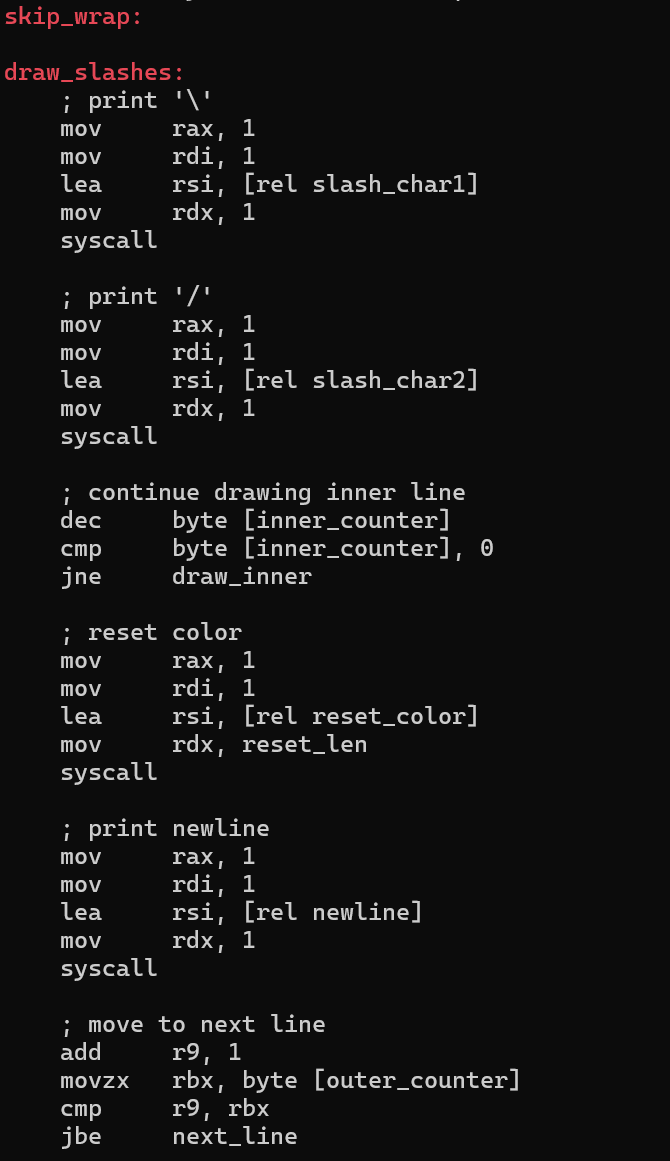
It establishes the mode of Colour and inputs the size of the triangle in this section. In case the input is 2, the rainbow mode is activated, and the rainbow index will revert to 0. It then requests the user to key in the size of the triangle. It reads the data into a buffer and the program begins converting the data to an integer, ASCII. It goes through each character to ensure that it is a digit (0 9) and creates the final number bit by bit. On encountering a non-digit or more than 3 characters, it goes jump to invalid input handling.



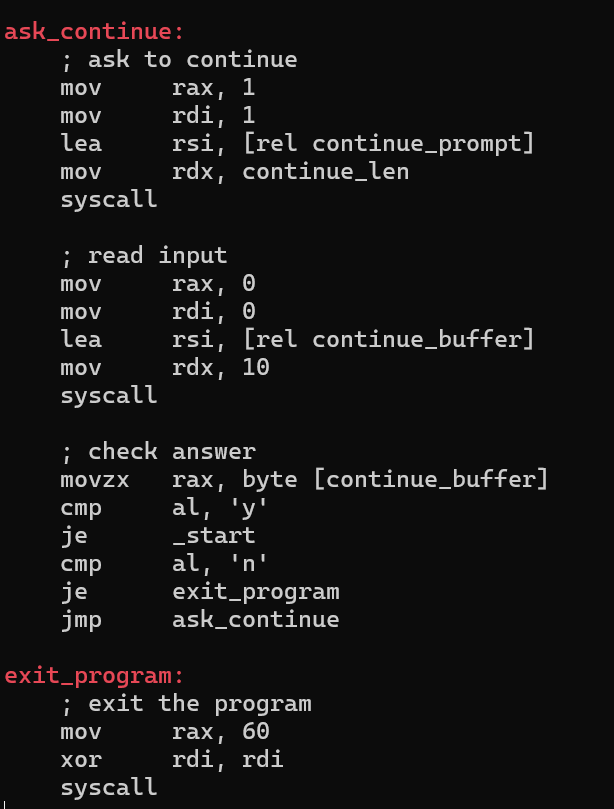
In this section, it is verified that the size of a triangle entered by a user will be valid (in range 0-20). When it is too large, it will print an error and request again. In case it is correct, it records the value and skips to initiate the drawing of the triangle. In draw\_triangle an outer loop is first established and follows r9 as the current line. It computes the number of pairs of slashes to draw, counting the value of the line variable in each line, after which it is decremented to invoke another rainbow color index as soon as a second rainbow color index is finished. Within the inner loop it verifies whether rainbow mode is enabled or not. Otherwise, it prints the escape code of white color and proceeds with the slash characters drawing.



This section does rainbow printing. The current rainbow\_index is used to obtain the next color in the color table and the next color is printed out on the screen. Once that is done it raises the index to the next color. After index 6 (last line of the color list) it starts clockwise where index 0 is and the rainbow effect continuously repeats.



This part deals with the aspect of drawing a single line of the triangle. It begins by printing a backslash ( \ ) and a forward slash ( / ) to give one V, shape. It reduces the inner loop counter after every two numbers are drawn. In case there are additional pairs left, then it revisits the draw process by attracting those. After the line is finished, it resets the color of its text font, outputs a newline character and increments its outer line counter to go to the next line. When the triangle is not done yet it goes back to draw the next line.



This component queries the user whether he/she wishes to continue or not. It prints a prompt, then reads to a buffer what a user has typed. In case y is entered, it re-starts the program and continues at \_start. when it is n, it quits the program through the exit syscall. In case the input is not the particular thing, it displays the prompt again until the correct answer is entered.

CONCLUSION

This project in essence entails investigation and application of numerous imaginative bodies and patterning through considering low-level and advanced concepts of assembly language. Each member individually contributed some programs that did not only demonstrate a good level of comprehension of low-level instructions but also paid a significant attention to user interaction, input validation, and ANSI colour codes to be able to enhance the visual output. Whether it is a simple geometrical shape such as square and triangle or a symbol such as heart, arrow, and trapezium, this assignment has provided us with an opportunity to use our theoretical knowledge in a real-life channel in a very kind of interesting way. This time we were also able to work together and analyse each other’s codes to improve our working knowledge of assembly language system call, memory management, looping, conditional, and branching. Reusability and readability in coding was improved by modularizing functions that repeatedly performed activities such as setting colours, and drawing. Good functional programs visible to the user were the actual products of the promising effort to program in assembly language. Somewhere along the way we enhanced our skill at programming, and as a bi-product we were being fostered to think critically and to work as a team.

REFERENCES

die.net. (n.d.). *write(2) - Linux man page*. Retrieved from die.net: https://linux.die.net/man/2/write

FILLIPO VALSORDA. (n.d.). *Searchable Linux Syscall Table*. Retrieved from FILLIPO VALSORDA: https://filippo.io/linux-syscall-table/

GeeksforGeeks. (2024, May 1). *Create Text File Using Nano*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/linux-unix/create-text-file-using-nano/

GeeksforGeeks. (2024, March 20). *How to Undo in Nano Editor*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/linux-unix/how-to-undo-in-nano-editor/

GeeksforGeeks. (2024, June 5). *Nano Text Editor in Linux*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/linux-unix/nano-text-editor-in-linux/

NASM. (n.d.). *NASM Tutorial*. Retrieved from cs.lmu.edu: https://cs.lmu.edu/~ray/notes/nasmtutorial/

tutorialspoint. (n.d.). *Assembly Programming Tutorial*. Retrieved from tutorialspoint: https://www.tutorialspoint.com/assembly\_programming/index.htm

Wikipedia. (n.d.). *ANSI escape code*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/ANSI\_escape\_code

WORKLOAD MATRIX

|  |  |  |
| --- | --- | --- |
| NAME | TASK | SIGNATURE |
| AHMED MIRAHUSAIN ALVI (TP084807) | * **Hexagon** * **Ellipse** * **Table of Contents** | *AHMED ALVI* |
| MOHAMMED YOUSEF MOHAMMED MOHAMMED (TP085042) | * **Alphabet X** * **Circle** * **References** | **MO** |
| SULTAN ABDULLA OMAR TAKRORI (TP085327) | * **Diamond** * **Pentagon** * **Introduction** | **Stk** |
| MAHRUS SHAMSUL AHSAN (TP085562) | * **Square** * **Rectangle** * **Main Menu** | ***~~MSA~~*** |
| ABDALLAH MOHAMED MAHMOUD MOHAMED MAHMOUD (TP085097) | * **Right-Angled Triangle** * **Line** * **Conclusion** | **Abdo** |
| ABDUL AZIZ VAYANI (TP080822) | * **Right Arrow** * **Triangle** * **Workload Matrix** | ***Aziz*** |