ELEMENTS OF COMPUTING-II

A THESIS
Submitted by

Team17
Group Members

BARANIDHARAN SELVARAJ
[CB.SC.U4AIE23015]

KEERTHIVASAN S V
[CB.SC.U4AIE23037]

MOPURU SAI BAVESH REDDY
[CB.SC.U4AIE23044]

SUTHEKSHAN KARUPPUSAMI
[CB.SC.U4AIE23067]

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY
IN
CSE(AI)

Centre for Computational Engineering and Networking
AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE
AMRITA VISHWA VIDYAPEETHAM
COIMBATORE - 641 112 (INDIA)
JUNE- 2024

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

BONAFIDE CERTIFICATE

This is to certify that the thesis entitled "Image Processing in MIPS" submitted by Baranidharan Selvaraj [CB.SC.U4AIE23015], Keerthivasan S V [CB.SC.U4AIE23037], Mopuru Sai Bavesh Reddy [CB.SC.U4AIE23044], Suthekshan Karuppusami [CB.SC.U4AIE23067], for the award of the Degree of Bachelor of Technology in the "CSE(AI) " is a bonafide record of the work carried out by her under our guidance and supervision at Amrita School of Artificial Intelligence, Coimbatore.

ACKNOWLEDGEMENT

We would like to express our special thanks of gratitude to our teacher (MS. SREELAKSHMI K ma’am), who gave us the golden opportunity to do this wonderful project on the topic “Image Processing in MIPS”, which also helped us in doing a lot of Research and we came to know about so many new things. We are thankful for the opportunity given.

We would also like to thank our group members, as without their cooperation, we would not have been able to complete the project within the prescribed time.

ABSTRACT

Image processing is a fundamental area in computer science and engineering, widely used in applications such as computer vision, medical imaging, and multimedia systems. Implementing image processing algorithms in MIPS assembly provides a deeper understanding of the low-level operations and optimizations needed for efficient execution on resource-constrained systems.

The project's scope includes a range of fundamental image processing techniques, including Image rotation, grayscale conversion, green scale conversion, image compression, edge detection, and threshold binarization. Each aspect will be meticulously crafted in MIPS assembly, emphasizing the efficiency and optimization strategies required for execution on resource-constrained systems. The choice of MIPS assembly provides insights into low-level operations and optimizations crucial for understanding computational tasks at the hardware level.

This project showcases how MIPS assembly language can be practically applied in processing images. By delving into essential algorithms at the assembly level, it offers valuable insights into low-level programming, optimization methods, and the difficulties of executing complex tasks on limited hardware. Ultimately, this endeavour contributes to a deeper understanding of leveraging MIPS architecture for efficient image processing applications.

TABLE OF CONTENTS

List of Figures

1. INTRODUCTION

1.1. Overview

The Image Processor project is a comprehensive system designed to perform a variety of image processing operations seamlessly. It provides users with a platform to manipulate images with precision and efficiency, offering a wide range of functionalities tailored to meet diverse requirements. The project encompasses several key components like the Image Loader module for importing images, the Processor module for executing processing operations, the display module for displaying modified images in the Bitmap Display, and various auxiliary modules for supporting functionalities. These components work in harmony to enable users to perform operations such as rotation, flipping, colour manipulation, and advanced techniques like edge detection and threshold binarization. This report elucidates the architecture and functionality of each module, highlighting their interconnections to deliver a robust and versatile image processing solution.

1.2. Objectives

The primary objectives of this project are to:

- Implement Image Processing Operations: Develop algorithms to perform various image processing tasks, including rotation, flipping, color manipulation, and advanced techniques like edge detection and threshold binarization.
- Optimize MIPS Assembly Implementation: Utilize efficient MIPS assembly instructions to execute image processing algorithms, ensuring optimal performance and resource utilization.
- Manage Memory and Data Handling: Implement memory management techniques to handle image data efficiently, minimizing memory usage and maximizing processing speed.

2. INTRODUCTION TO IMAGE PROCESSING ALGORITHMS

2.1. Rotate Image

This operation entails rearranging the pixel data of an image matrix so that the columns become rows, effectively changing the orientation of the image. The rotate left operation in image processing involves transforming an image by 90 degrees counterclockwise. The algorithm typically follows a two-step process: transposition and flipping. First, the image matrix is transposed, swapping rows and columns. Then, the transposed matrix is flipped along the horizontal axis to complete the rotation. This approach ensures that each pixel retains its position relative to the image's orientation, resulting in a smooth and accurate rotation.

2.2. Grayscale Conversion

This grayscale conversion algorithm is relatively straightforward and computationally efficient. It preserves the luminance information from the original colour image while discarding colour information, resulting in a grayscale representation that is suitable for various image processing tasks, such as edge detection, image enhancement, and feature extraction. Additionally, the algorithm's simplicity makes it easy to implement in various programming languages and platforms.

2.3. Image Blurring

This algorithm iteratively calculates the average rgb intensities of pixels within an interpolation matrix around each pixel in the original image. By averaging the intensities, it creates a blurred effect in the destination image. This approach provides a basic understanding of image blurring techniques in the context of low-level mips assembly programming.

2.4. Edge Detection

Edge detection is an image-processing technique that is used to identify the boundaries (edges) of objects or regions within an image. Edges are among the most important features associated with images. We know the underlying structure of an image through its edges. Sudden changes in pixel intensity characterize edges. We need to look for such changes in the neighboring pixels to detect edges. The edge operations implemented Sobel and Prewitt operators.

2.5. Binarization

Binarization Binarization is a digital image processing technique used to convert a grayscale image or a colour image into a binary image. The binary image created as a result of binarization contains only two pixel values, typically 0 and 1, where 0 represents the background (usually black) and 1 represents the foreground or the object of interest (usually white). Binarization is a fundamental preprocessing step in various computer vision and image analysis applications, particularly in the areas of document analysis, optical character recognition (OCR), and pattern recognition.

3. METHODOLOGY

3.1. Rotate left

This operation entails rearranging the pixel data of an image matrix so that the columns become rows, effectively changing the orientation of the image. The rotate left operation in image processing involves transforming an image by 90 degrees counterclockwise. The algorithm typically follows a two-step process: transposition and flipping. First, the image matrix is transposed, swapping rows and columns. Then, the transposed matrix is flipped along the horizontal axis to complete the rotation. This approach ensures that each pixel retains its position relative to the image's orientation, resulting in a smooth and accurate rotation.
Implementing the rotate left algorithm in involves working with memory addresses, loop structures, and register operations.

The algorithm for rotating an image 90 degrees to the left involves two primary steps: matrix transposition and horizontal flipping.
The **rotate90lCall** label orchestrates the rotation process by calling two subroutines: **rotate90l** and **flipX**.

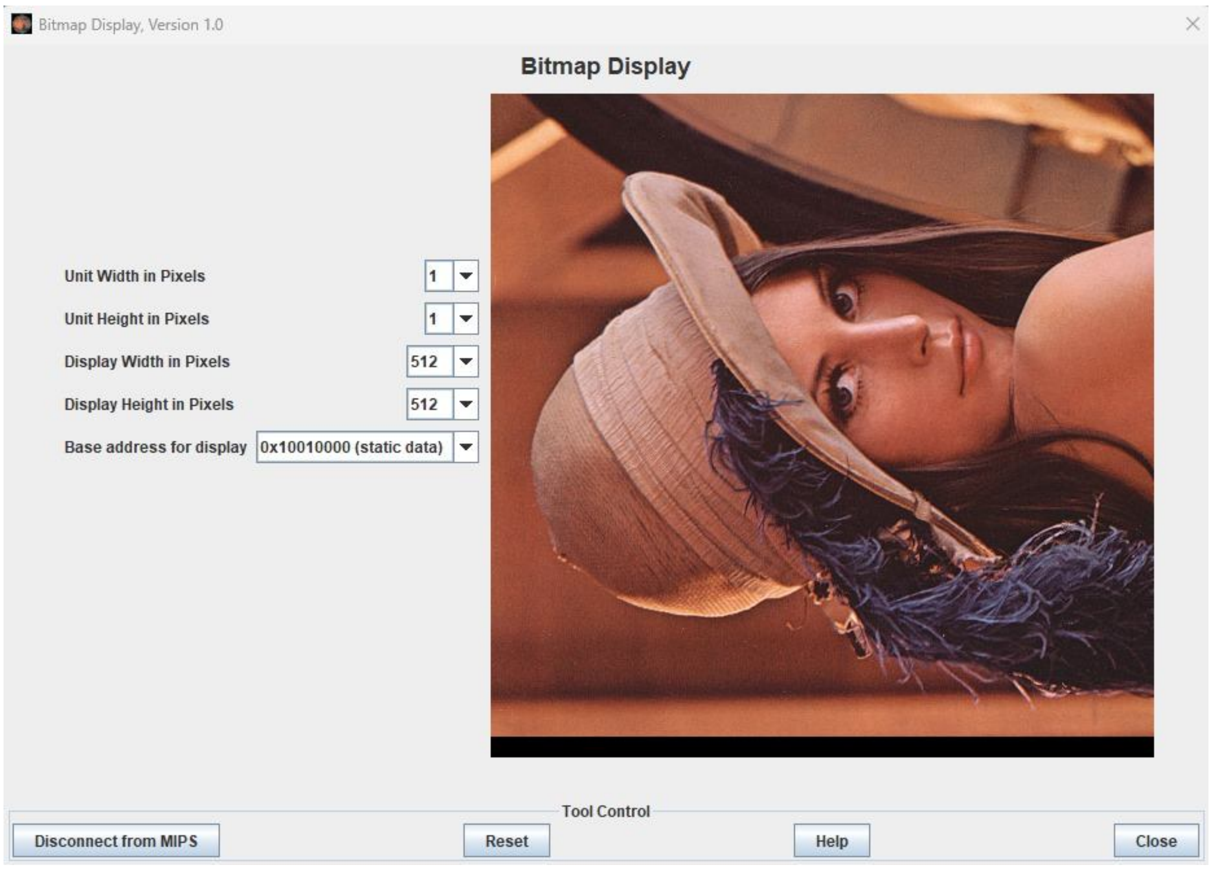
i. **rotate90l label :**

In the **rotate90l** subroutine, the image rotation to the left is accomplished through meticulous memory management and iterative indexing. The routine initializes variables such as the line index **n** and matrix size **N**, utilizing nested loops to iterate through each element pair **(n, m)** in the image matrix. Within these loops, elements are swapped to transpose the matrix effectively.

ii. **flipX label:**

The flipX subroutine is responsible for horizontally flipping the image along the x-axis. Leveraging registers for iterative indexing and memory management, it initializes variables like column (t0) and line indices (t2), along with calculating iteration limits. The main loop, loop_flipX, traverses each column of the image, swapping upper and lower byte values to achieve horizontal flipping. Conditional branching, like refreshFlipX, ensures proper handling of line breaks, applying the flipping operation to each row until completion.

In essence, these subroutines work in tandem: rotate90l transposes the matrix to prepare for rotation, and flipX completes the rotation by flipping the transposed image horizontally. Through efficient memory management and meticulous iteration, the algorithm successfully achieves a 90-degree rotation to the left, providing a comprehensive solution for image manipulation in MIPS assembly language.



Unit Width in Pixels: 1
Unit Height in Pixels: 1
Display Width in Pixels: 512
Display Height in Pixels: 512
Base address for display: 0x10010000 (static data)

Fig.i Rotate left

3.2. Rotate right

This operation involves rearranging the pixel data of an image matrix so that the columns become rows, effectively changing the orientation of the image. The rotate right operation in image processing entails transforming an image by 90 degrees clockwise. The algorithm typically follows a two-step process: transposition and flipping. First, the image matrix is transposed, swapping rows and columns. Then, the transposed matrix is flipped along the horizontal axis to complete the rotation.

This approach ensures that each pixel retains its position relative to the image's orientation, resulting in a smooth and accurate rotation. Implementing the rotate right algorithm involves working with memory addresses, loop structures, and register operations.

The algorithm for rotating an image 90 degrees to the left involves two primary steps: matrix transposition and horizontal flipping.

The rotate90rCall label initiates the rotation process by calling two essential subroutines: rotate90r and flipX.

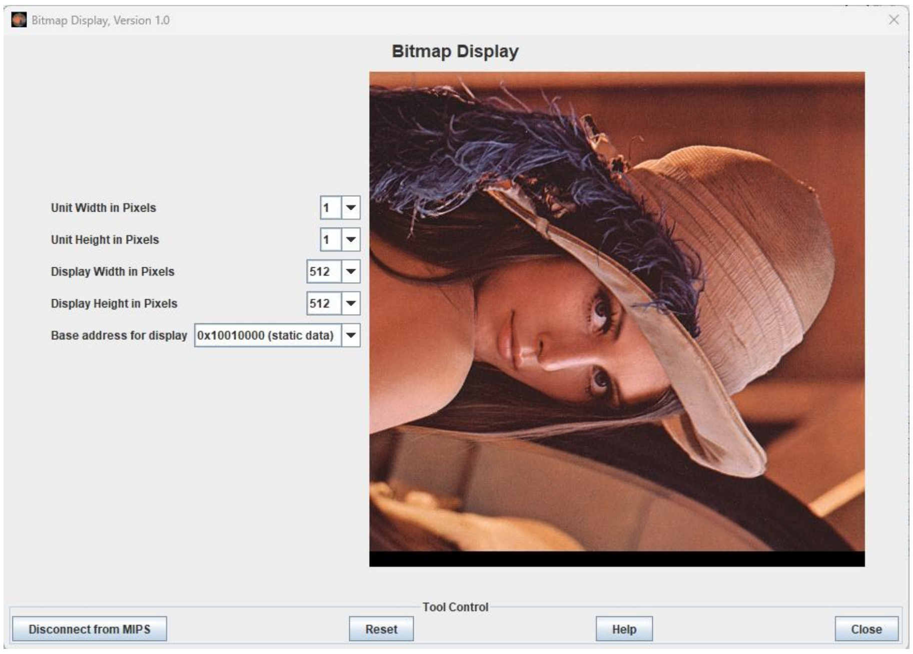
i. **rotate90r label :**

   In the rotate90r subroutine, the image rotation to the right is achieved through meticulous memory management and iterative indexing. Variables such as the line index n and matrix size N are initialized, facilitating nested loops to iterate through each element pair (n, m) in the image matrix. Within these loops, elements are swapped to transpose the matrix effectively.

ii. **flipX label:**

   The flipX subroutine, as in the left rotation, plays a crucial role in horizontally flipping the image along the x-axis. By leveraging registers for iterative indexing and memory management, it initializes variables like column (t0) and line indices (t2), determining iteration limits. The main loop, loop_flipX, processes each column of the image, swapping upper and lower byte values to achieve horizontal flipping. Conditional branching, such as refreshFlipX, ensures the proper handling of line breaks, applying the flipping operation to each row until completion.

   Together, these subroutines work seamlessly: rotate90r prepares the matrix for rotation by transposing it, while flipX completes the rotation by flipping the transposed image horizontally. Through efficient memory management and meticulous iteration, the algorithm successfully achieves a 90-degree rotation to the right, offering a comprehensive solution for image manipulation in MIPS assembly language.


Bitmap Display, Version 1.0

Bitmap Display

Unit Width in Pixels 1

Unit Height in Pixels 1

Display Width in Pixels 512

Display Height in Pixels 512

Base address for display 0x10010000 (static data)

Fig.ii Rotate right

3.3. Horizontal flip:

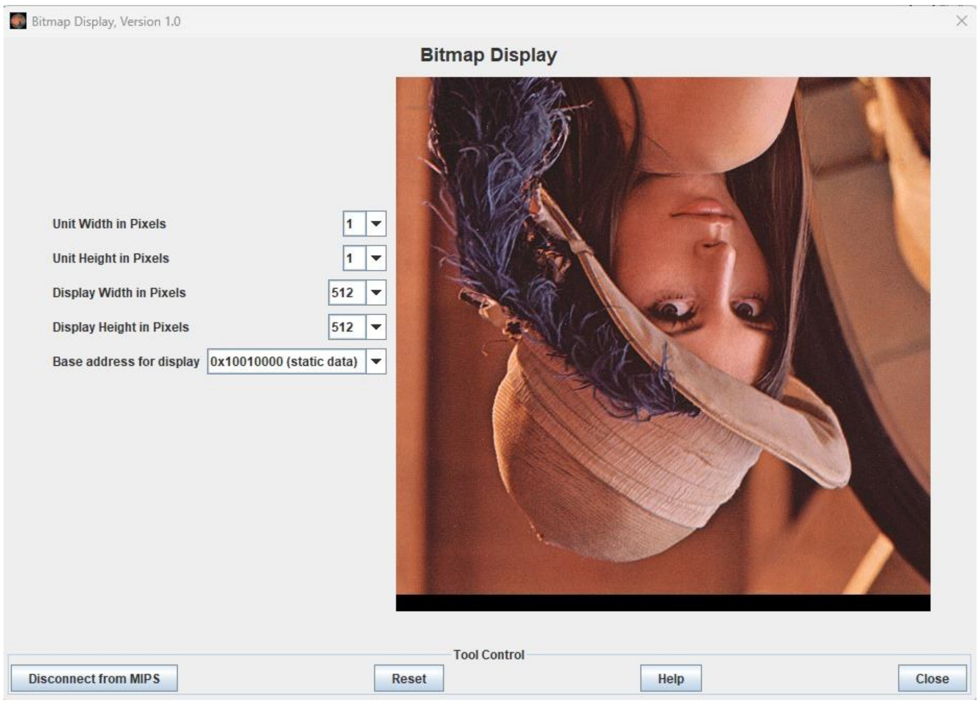
In image processing, geometric transformations like flipping play a crucial role in altering the orientation of images. One such transformation is flipping an image along the x-axis, which effectively mirrors the image horizontally. In MIPS assembly language, this operation is facilitated by the flipX subroutine. By rearranging pixel data, flipX mirrors the image horizontally, preserving its overall structure while changing its orientation.

The flipXCall label serves as the entry point to invoke this transformation

Upon entry into the subroutine, the initial values of various registers are set to manage iterative indexing. These registers include **t0** for column iteration index, **t1** for the column limit of iterations (width), **t2** for the line iteration index, and **t3** for the line limit of iterations (height/2). Additionally, memory addresses **t4** and **t5** are utilized to store the upper and lower byte addresses, respectively, of the image pixels.

The subroutine then enters a loop (**loop_flipX**) where it iterates over each column of the image up to the specified column limit. Within this loop, the subroutine swaps the values of the upper and lower bytes of each pixel in the current column, effectively flipping the image horizontally. After each iteration, the column index (**t0**) is incremented to process the next column until all columns have been processed.

The conditional branch (**refreshFlipX**) is included to handle line breaks, ensuring that the flipping operation is applied to each row of the image. Once all columns have been processed for a given row, the column index is reset, and the memory address for the lower byte is adjusted to move to the next row. This process continues until all rows have been processed, after which control returns from the subroutine.



Unit Width in Pixels: 1
Unit Height in Pixels: 1
Display Width in Pixels: 512
Display Height in Pixels: 512
Base address for display: 0x10010000 (static data)

Fig.iii Flip along x axis

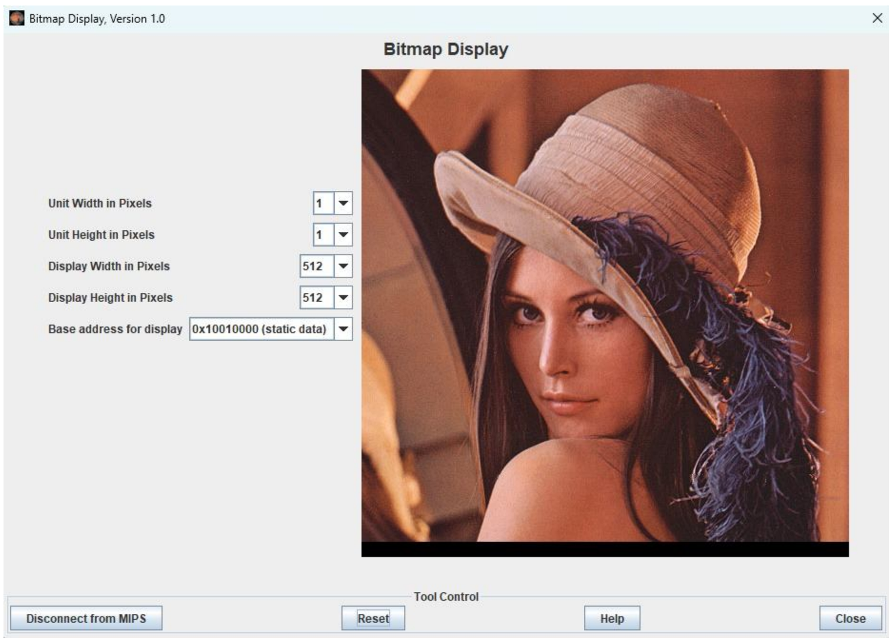
3.4.Vertical flip:

In image processing, geometric transformations like flipping play a crucial role in altering the orientation of images. One such transformation is flipping an image along the y-axis, which effectively mirrors the image vertically. In MIPS assembly language, this operation is facilitated by the flipY subroutine. By rearranging pixel data, flipY mirrors the image vertically, preserving its overall structure while changing its orientation.

The flipYCall label serves as the entry point to invoke this transformation. Upon entry into the subroutine, the initial values of various registers are set to manage iterative indexing. These registers include t0 for the line iteration index, t1 for the line limit of iterations (height), t2 for the column iteration index, and t3 for the column limit of iterations (width/2). Additionally, memory addresses t4 and t5 are utilized to store the left and right byte addresses, respectively, of the image pixels.

The subroutine then enters a loop (loop_flipY) where it iterates over each row of the image up to the specified row limit. Within this loop, the subroutine swaps the values of the left and right bytes of each pixel in the current row, effectively flipping the image vertically. After each iteration, the row index (t0) is incremented to process the next row until all rows have been processed.

The conditional branch (refreshFlipY) is included to handle column breaks, ensuring that the flipping operation is applied to each column of the image. Once all rows have been processed for a given column, the row index is reset, and the memory address for the right byte is adjusted to move to the next column. This process continues until all columns have been processed, after which control returns from the subroutine.



Unit Width in Pixels 1
Unit Height in Pixels 1
Display Width in Pixels 512
Display Height in Pixels 512
Base address for display 0x10010000 (static data)

Fig.iv Flip along y axis

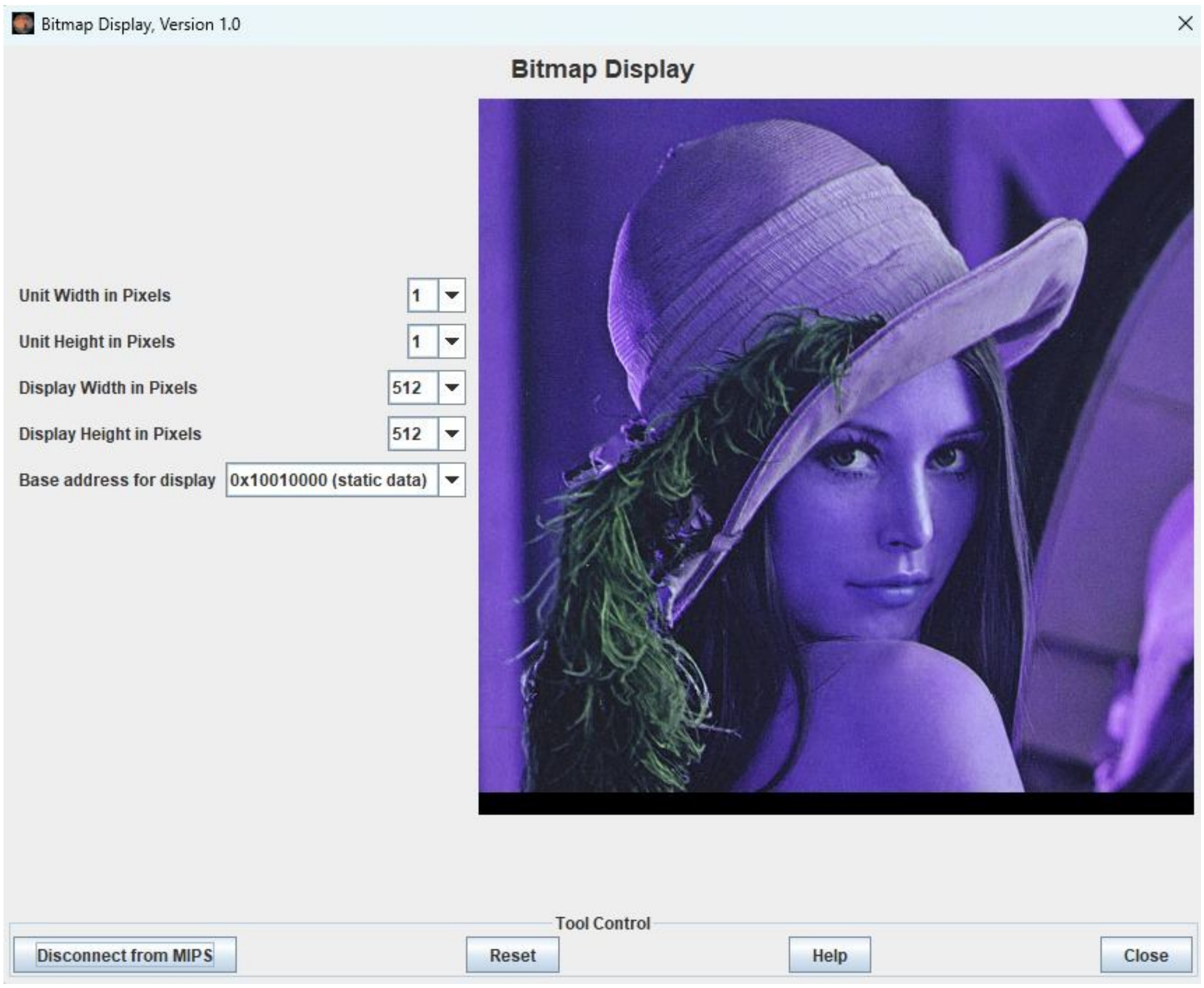
3.5. Rotate Colour :

The rotateColoursCall subroutine is a pivotal component of an image processing

application written in MIPS assembly language. This label acts as the entry point for applying a colour rotation filter to an image. When invoked, it retrieves the image properties from the register $a0 and the image data's starting address from the register $a1. It then proceeds to call the rotateColours subroutine to execute the colour rotation filter operation.
Within the rotateColours subroutine, various registers are utilized for efficient memory management and iterative indexing. Initially, the subroutine initializes the registers $t0 and $t2 to store backup copies of the image properties and the starting address of the image data, respectively. It retrieves the height and width of the image from the image properties and calculates the total number of iterations required to process all pixels in the image.

The main processing loop iterates over each pixel in the image, beginning at the starting address of the image data stored in memory. Within this loop, each pixel's colour channels are rearranged to apply the colour rotation filter. Specifically, the subroutine reads the colour channels of each pixel, manipulates them according to the desired colour rotation effect, and stores the modified pixel value back into memory.

Once all pixels have been processed, the subroutine concludes its operation and returns control to the main program flow. The execution then proceeds to the next instruction after the rotateColoursCall subroutine, typically directing the user back to the menu options screen to continue interacting with the image processing application. In summary, the rotateColoursCall and rotateColours subroutines collectively facilitate the application of colour rotation filters, enabling diverse and dynamic image manipulation capabilities within MIPS assembly language.



Unit Width in Pixels: 1
Unit Height in Pixels: 1
Display Width in Pixels: 512
Display Height in Pixels: 512
Base address for display: 0x10010000 (static data)

Fig.v. Colour Rotation

3.6. Colour Inversion:

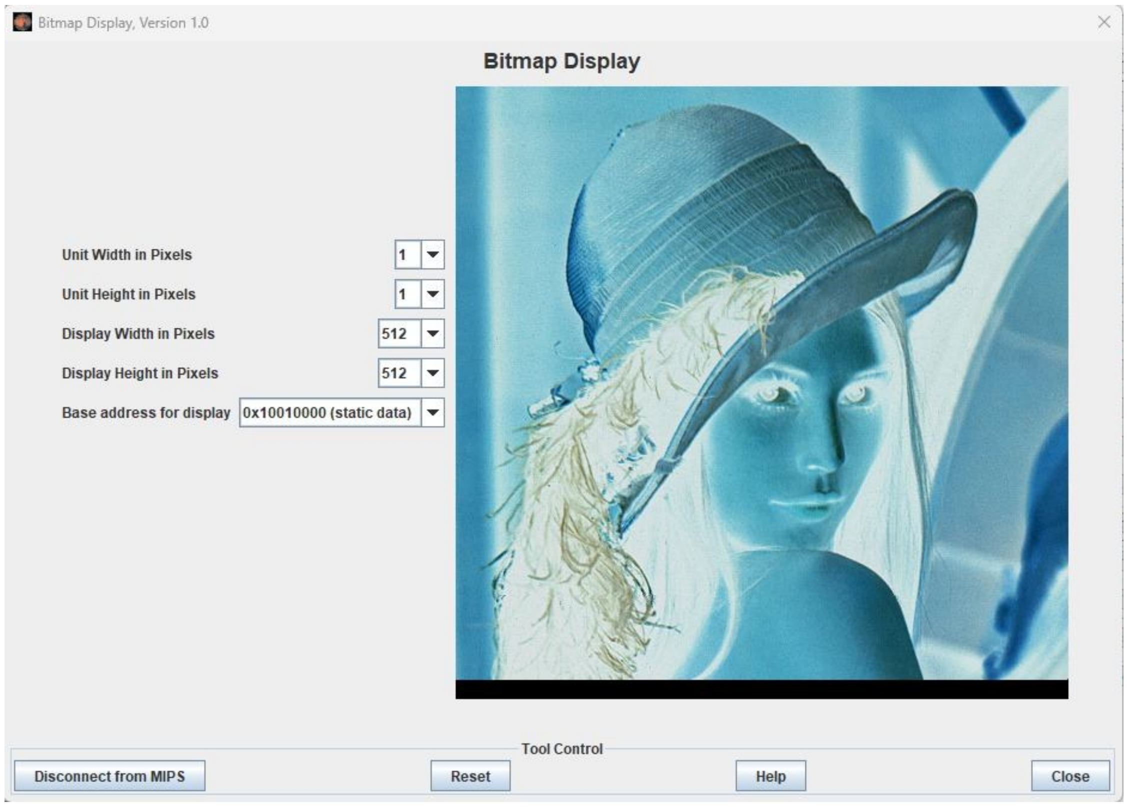
The invertColoursCall subroutine serves as a pivotal entry point within an image processing application written in MIPS assembly language. This label is responsible for initiating the invert colours filter, which involves reversing the colour values of each pixel in the image. Upon invocation, the subroutine retrieves the image properties from register \$a0 and the starting address of the image data from register \$a1. It then proceeds to call the invertColours subroutine to execute the colour inversion filter operation.

Within the invertColours subroutine, various registers are utilized for efficient memory management and iterative indexing. Initially, the subroutine creates backup copies of the image properties and data addresses in registers \$t0 and \$t1, respectively. Additionally, it sets register \$t9 to the value 255, which will be used to calculate the inverted colour values.

The main processing loop iterates over each pixel in the image, starting from the memory address defined by \$t2. Within this loop, the colour values of each pixel are inverted using the equation R

= (255-R), G = (255-G), and B = (255-B), where R, G, and B represent the red, green, and blue colour channels, respectively. These inverted colour values are then stored back into memory at the appropriate pixel address.

Once all pixels have been processed, the subroutine concludes its operation and returns control to the main program flow. Typically, this leads to the execution of the next instruction after the invertColoursCall subroutine, which may involve returning to the menu options screen to continue interacting with the image processing application. In summary, the invertColoursCall and invertColours subroutines together facilitate the application of colour inversion filters, enabling dynamic and versatile image manipulation capabilities within MIPS assembly language.



Unit Width in Pixels: 1
Unit Height in Pixels: 1
Display Width in Pixels: 512
Display Height in Pixels: 512
Base address for display: 0x10010000 (static data)

Fig.vi Image Colour Inversion

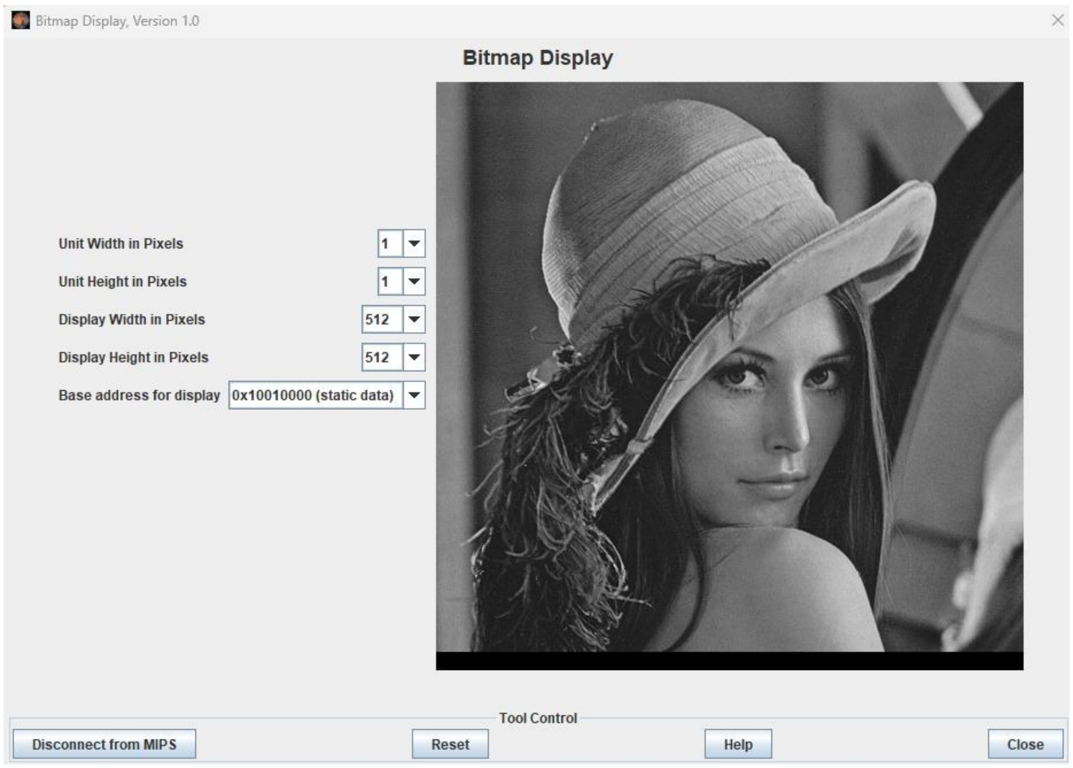
3.7. GreyScale

The greyScaleCall subroutine is a crucial entry point within the image processing program . Its purpose is to apply a grayscale filter to the input image, converting it from colour to grayscale. This subroutine reads the image properties from register $a0 and the starting address of the image data from register $a1. It then calls the greyScale subroutine to execute the grayscale conversion operation.

Within the greyScale subroutine, various registers are utilized for efficient memory management and iterative indexing. Initially, backup copies of the image properties and data addresses are created in registers $t0 and $t1, respectively. The subroutine also sets the starting address for iterating through the image data in register $t2.

The main processing loop iterates over each pixel in the image, starting from the memory address defined by $t2. Within this loop, the grayscale value for each pixel is calculated using the formula I = 0.2989R + 0.5870G + 0.1140*B, where R, G, and B represent the red, green, and blue colour channels, respectively. These grayscale values are then stored back into memory at the appropriate pixel address.

Once all pixels have been processed, the subroutine concludes its operation and returns control to the main program flow. Typically, this leads to the execution of the next instruction after the greyScaleCall subroutine, which may involve returning to the menu options screen for further interaction with the image processing application. In summary, the greyScaleCall and greyScale subroutines together facilitate the application of grayscale conversion filters, enhancing the versatility of image manipulation capabilities within MIPS assembly language.



Unit Width in Pixels 1
Unit Height in Pixels 1
Display Width in Pixels 512
Display Height in Pixels 512
Base address for display 0x10010000 (static data)

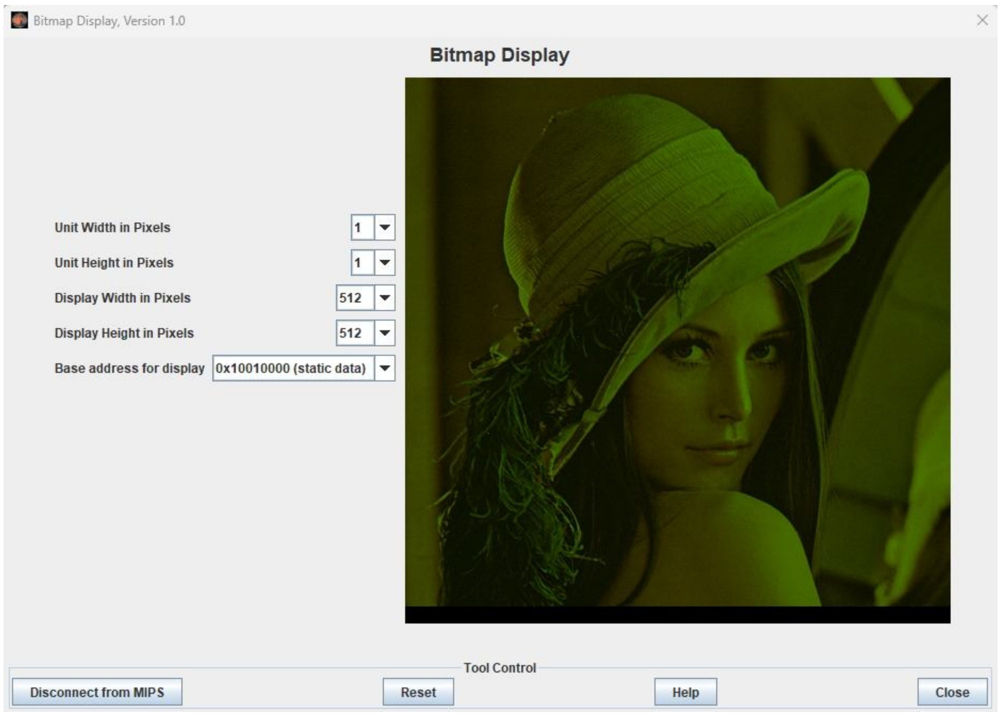
Fig.vii Image Grey Scale Conversion

3.8. GreenScale:

The GreenScaleCall subroutine is designed to apply a unique green-scale filter to an image. This transformation manipulates the colour properties of the image, primarily focusing on the green channel, and is an unintended result of modifying the grayscale filter. The greenScaleCall label reads image properties from register $a0 and the image data from register $a1, then invokes the greenScale subroutine to apply the filter.

In the greenScale subroutine, several registers are employed for managing memory and iterative processing. Initially, the image properties and data addresses are backed up in registers $t0 and $t1. The starting address for processing the image data is set in register $t2.

The main loop, loop_greenScale, iterates over each pixel in the image. Within this loop, the subroutine reads the red, green, and blue colour values of each pixel. The green channel value is multiplied by 5870 and then divided by 10000 to scale it appropriately. The red and blue channels are similarly processed with their respective scaling factors. These processed values are then combined to form the new pixel value, which emphasizes the green component more heavily. Finally, the processed pixel value is stored back in memory, and a null byte is added to the subsequent memory location, ensuring the image format integrity. The loop continues until all pixels have been processed, at which point the subroutine terminates and control returns to the main program, which may involve returning to the menu options screen.
This greenScale subroutine, through its manipulation of pixel values, highlights the green channel, providing a unique visual effect on the processed image.



Unit Width in Pixels 1
Unit Height in Pixels 1
Display Width in Pixels 512
Display Height in Pixels 512
Base address for display 0x10010000 (static data)

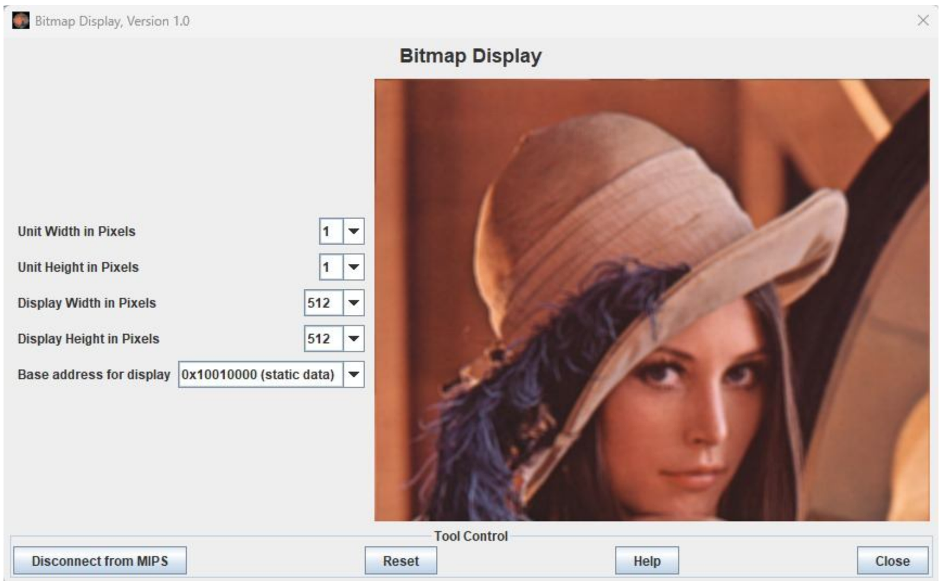
Fig.viii Image Green Scale Conversion

3.9.Image Blur

Blurring algorithm for an image by averaging the colour values of pixels within a specified degree of interpolation is implemented. The program starts by loading the addresses of the original and new images into registers and prompts the user to input the degree of blurring, which determines the size of the interpolation matrix. It then sets up necessary parameters, including calculating the number of bytes per line and setting constants to navigate the memory positions during the interpolation process.

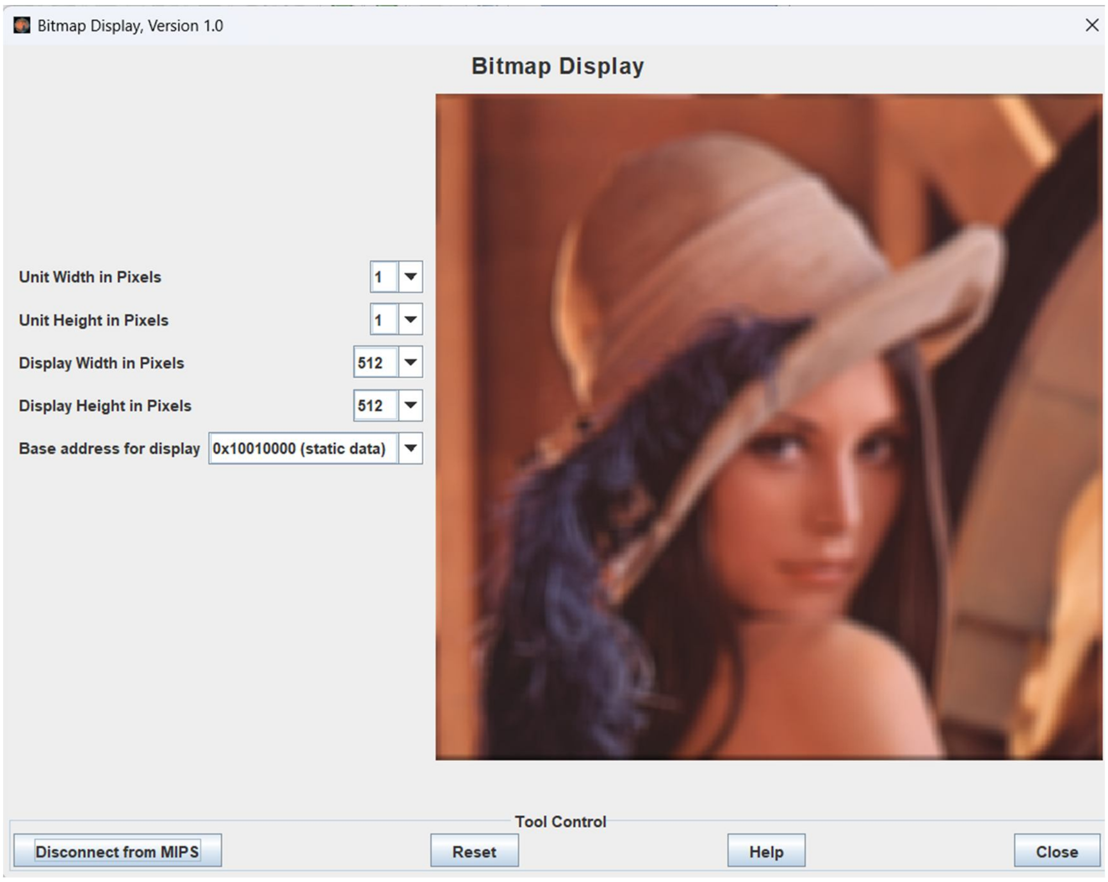
The main loop iterates through each pixel in the image, adjusting the position to start at the beginning of the interpolation matrix. Within a nested loop, the code sums the blue, green, and red intensity values of the pixels in the matrix. When the end of a line within the matrix is reached, it moves to the next line and resets the column counter. After processing all lines, the program computes the average colour values by dividing the sum of each colour component by the number of pixels.

The averaged blue, green, and red values are combined into a single 32-bit value and written to the new image at the corresponding position. The process repeats for all pixels, resulting in a blurred image. This code illustrates a low-level approach to image processing using MIPS assembly, demonstrating the manipulation of memory and arithmetic operations to achieve blurring.



Unit Width in Pixels: 1
Unit Height in Pixels: 1
Display Width in Pixels: 512
Display Height in Pixels: 512
Base address for display: 0x10010000 (static data)

Fig.ix Image Blurring at level 2



Unit Width in Pixels: 1
Unit Height in Pixels: 1
Display Width in Pixels: 512
Display Height in Pixels: 512
Base address for display: 0x10010000 (static data)

Fig.x Blurring at level 5

3.10. Edge Detection

The edge detection algorithm for the given input image, is implemented using two methods: Prewitt and Sobel. It starts by setting up the stack, saving the necessary registers, and creating space for variables. The code then loads the bytes per line of the image, calculates the number of pixels per line, and stores the height of the image.

The main loop, **repeatEdge**, prompts the user to choose the edge detection method (Prewitt or Sobel) or return to the previous menu. Depending on the user's choice, it loads the appropriate kernel (either Prewitt or Sobel) for both the X and Y directions and sets the number of elements to be averaged.

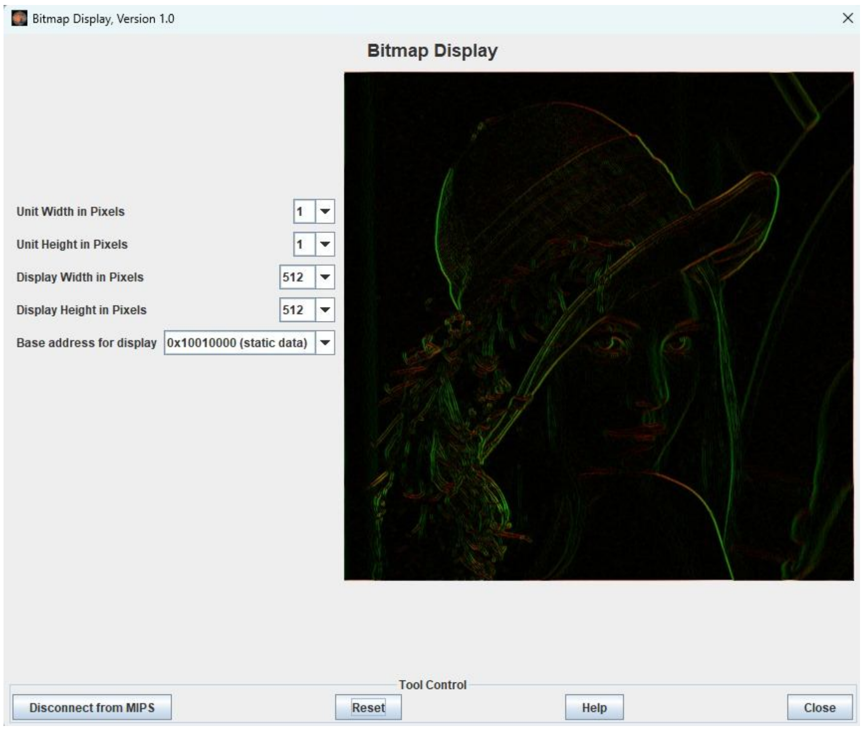
The **EdgeLoop** calculates the initial position of the kernel, with the origin at the centre, and sets cursors for the X and Y positions. The nested loop, **loopExBorda**, iterates through the image grid, calculating the position in memory using the cursor coordinates. It loads the address of the pixel in the original image and calculates the pixel value using the X grid by calling the **calculatePixel** function. This function computes the weighted sum of the green values of the neighboring pixels, using the specified kernel, and returns the result. The code ensures that the value is non-negative, and the result is stored in **$s0**.

Similarly, the pixel value is calculated using the Y grid, and the result is stored in **$s1**. The code then combines the results for the X and Y components by positioning them correctly in the final pixel value and stores the result in the destination image.

The cursor is incremented, and the loop checks if it has reached the end of the line or the end of the image. If there are more pixels to process, it continues; otherwise, it moves to the next line or ends the loop.

After completing the edge detection, the code restores the stack variables and registers, frees up the stack space, and jumps to the **print** label.

The **calculatePixel** function is responsible for computing the weighted sum of the pixel values using the specified kernel. It iterates through the kernel grid, calculates the position of each pixel in memory, extracts the green component, and multiplies it by the corresponding kernel weight. The results are accumulated, and after processing the entire kernel, the accumulator value is divided by the number of elements to normalize it. The final result is returned in **$v0**.


Bitmap Display, Version 1.0

Bitmap Display

Unit Width in Pixels 1

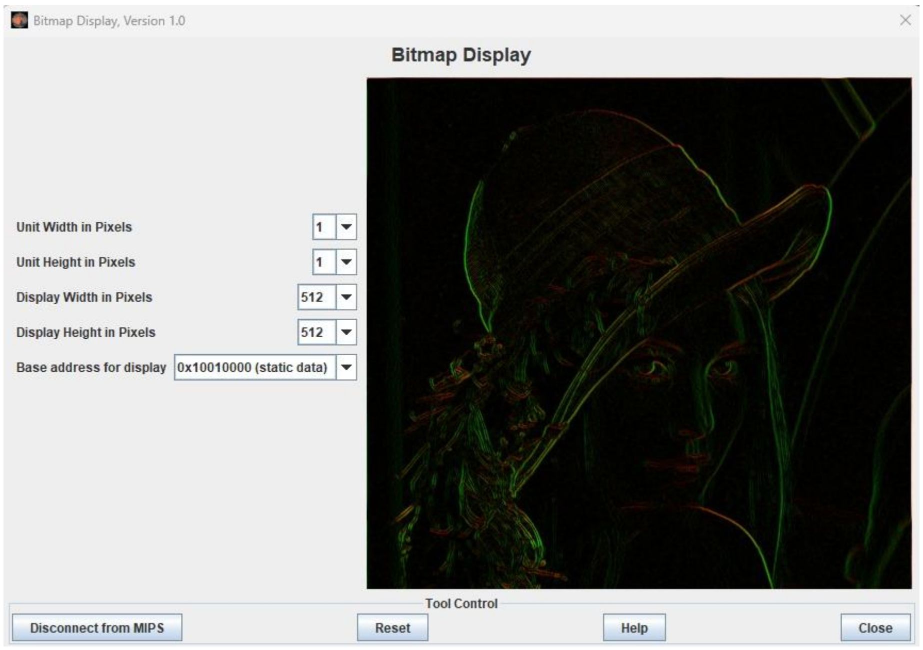
Unit Height in Pixels 1

Display Width in Pixels 512

Display Height in Pixels 512

Base address for display 0x10010000 (static data)

Fig.xi Edge Detection using Prewitt


Bitmap Display, Version 1.0

Bitmap Display

Unit Width in Pixels 1

Unit Height in Pixels 1

Display Width in Pixels 512

Display Height in Pixels 512

Base address for display 0x10010000 (static data)

Fig.xii Edge Detection using Sobel

3.11. Threshold binarization

The code begins by prompting the user to enter a threshold value for binarization. This is done using a system call to print a message (Threshold_msg) asking the user for the input, followed by another system call to read the input value from the user. This value is stored in the register $s2, which will be used as the threshold for deciding whether a pixel should be white or black.

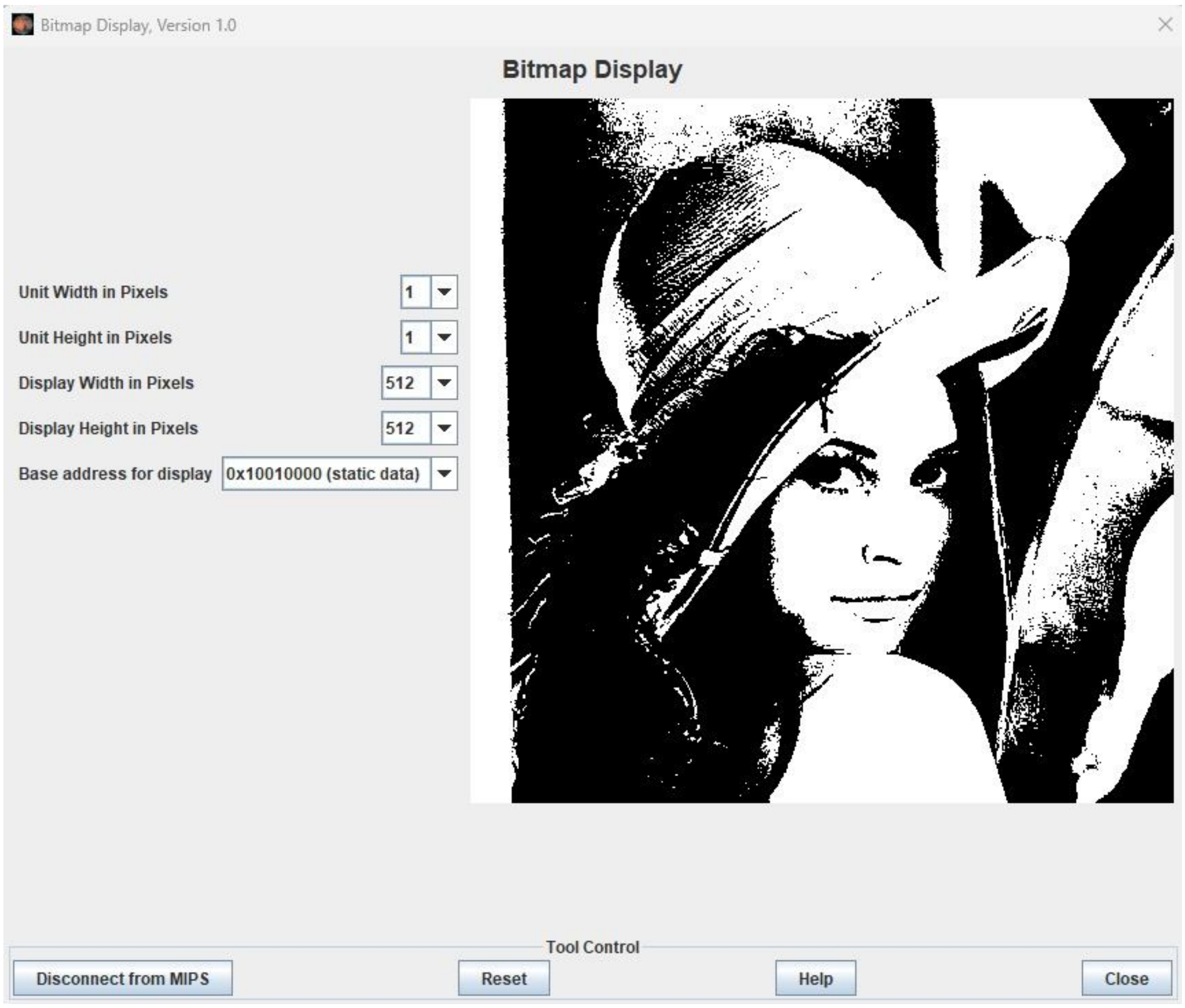
Next, the code prints a processing message (Processing_msg) to inform the user that the binarization process is starting. It then initializes two pointers: $s1 pointing to the start of the original image data and $s3 pointing to the start of the destination image data (the bitmap display). The register $s4 is initialized to zero and used as a loop counter.

The main processing loop (loop4) begins by checking if the loop counter $s4 has reached the total number of pixels in the image (stored in $s0). If so, it jumps to the print label to end the process. If not, it proceeds to load the green component of the current pixel. This is achieved by incrementing $s1 by 2 to point to the green byte of the pixel data and then loading the byte into $t0.

The value of the green component in $t0 is compared with the threshold value stored in $s2. If the green component is greater than the threshold, the pixel is set to white (0x00FFFFFF) by loading this value into $t1. If the green component is less than or equal to the threshold, the pixel is set to black (0x00000000) by loading zero into $t1.

After determining the colour, the value in $t1 is stored at the current position pointed to by $s3 in the destination image. Both pointers $s1 and $s3 are then incremented by 4 to move to the next pixel, and the loop counter $s4 is incremented by 4. The loop then jumps back to loop4 to process the next pixel.

This process continues until all pixels have been processed. Once the loop is complete, control jumps to the print label to end the binarization operation. The overall effect of this code is to produce a black and white image where each pixel’s colour is determined by comparing its green component to the user-provided threshold. If the green component is above the threshold, the pixel is set to white; otherwise, it is set to black. This approach effectively highlights areas of the image based on brightness, which is typical for binarization tasks.



Unit Width in Pixels 1
Unit Height in Pixels 1
Display Width in Pixels 512
Display Height in Pixels 512
Base address for display 0x10010000 (static data)

Fig.xiii Threshold Binarization at 150

4. CONCLUSION

4.1. Results

The project successfully achieved its objectives, efficiently implementing a diverse range of image processing operations using MIPS assembly language. The MIPS implementation ensured optimal performance and resource utilization, enhancing the overall efficiency of the system. The image processing pipeline operated seamlessly, seamlessly managing image importation, processing, and exportation tasks without encountering any issues.

5. BIBLIOGRAPHY

(1) Nisan, N., & Schocken, S. (2008). The Elements of Computing Systems: Building a Modern Computer from First Principles.

(2) Smith, J., & Goodwin, M. (2010). Enhancing Learning in Computer Architecture through Project- Based Learning

(3) assembly - mips rotate left pseudo code translation - Stack Overflow. https://stackoverflow.com/questions/28444984/mips-rotate-left-pseudo-code-translation.

(4) assembly - Translating a mips pseudo instruction 'rol ... - Stack Overflow. https://stackoverflow.com/questions/24542657/translating-a-mips-pseudo-instruction-rol.

(5) 3.10: Shift Operations - Engineering LibreTexts. https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_Languages/Introduction_To_MIPS_Assembly_Language_Programming_%28Kann%29/03%3A_MIPS_Arithmetic_and_Logical_Operators/3.10%3A_Shift_Operations.

(6) assembly - Getting colour of pixel in MIPS - Stack Overflow. https://stackoverflow.com/questions/47889965/getting-colour-of-pixel-in-mips.

(7) How Do You Write a MIPS Function to Horizontally Flip an Image?. https://www.physicsforums.com/threads/how-do-you-write-a-mips-function-to-horizontally-flip-an-image.392583/.

(8) How do I flip the most significant bit in MIPS? - Stack Overflow. https://stackoverflow.com/questions/29358563/how-do-i-flip-the-most-significant-bit-in-mips.

(9) Reversing the bits of a number in MIPS assembly. https://stackoverflow.com/questions/33675141/reversing-the-bits-of-a-number-in-mips-assembly.

(10) COMP 411: Lab 7: More Advanced Assembly - University of North Carolina .... https://www.cs.unc.edu/~porter/courses/comp411/s18/lab7.html.

(11) assembly - Assuming that you had a MIPS processer with PIPELINE but .... https://stackoverflow.com/questions/72015536/assuming-that-you-had-a-mips-processer-with-pipeline-but-without-hazard-preventi.

(12) Title: Image Thresholding in MIPS Assembly Image ... - Numerade. https://www.numerade.com/ask/question/mips-assembly-image-thresholding-is-a-simple-way-of-partitioning-an-image-into-a-foreground-and-background-this-image-analysis-

technique-is-a-type-of-image-segmentation-that-isolates-objects-13016/.
(13) Assembly Mips replace "and" with rotation and shift commands. https://stackoverflow.com/questions/31264514/assembly-mips-replace-and-with-rotation-and-shift-commands.
(14) MIPS Assembly Language Programming - California State University .... https://www.cs.csub.edu/~eddie/cmps2240/doc/britton-mips-text.pdf.
(15) Shift and rotate instructions facilitate manipulations of data (that is .... https://personal.utdallas.edu/~dodge/EE2310/lec14.pdf.
(16) assembly - How do you shift an image up by 50 pixels in MIPS? - Stack .... https://stackoverflow.com/questions/70326202/how-do-you-shift-an-image-up-by-50-pixels-in-mips.
(17) assembly - Initializing a bmp file in mips - Stack Overflow. https://stackoverflow.com/questions/67523940/initializing-a-bmp-file-in-mips.
(18) 3/9/06 MIPS Shift and Rotate Instructions - University of Northern Iowa. http://www.cs.uni.edu/~fienup/cs041s06/lectures/lec18_logical_shift_instrs.pdf.
(19) How do I reverse an integer in MIPS? - Stack Overflow. https://stackoverflow.com/questions/13392563/how-do-i-reverse-an-integer-in-mips.
(20) assembly - Greater than, less than equal, greater than equal in MIPS .... https://stackoverflow.com/questions/15183346/greater-than-less-than-equal-greater-than-equal-in-mips.
(21) MIPS Assembly Language Guide - University of Northern Iowa. http://www.cs.uni.edu/~fienup/cs041s08/lectures/lec20_MIPS.pdf.
(22)https://www.ti.com/lit/an/bpra059/bpra059.pdf?ts=1717351778695&ref_url=https%253A%252F%252Fwww.google.com%252F
(23) https://www.geeksforgeeks.org/image-edge-detection-operators-in-digital-image-processing/
(24)https://www.analyticsvidhya.com/blog/2022/07/a-brief-study-of-image-thresholding-algorithms/