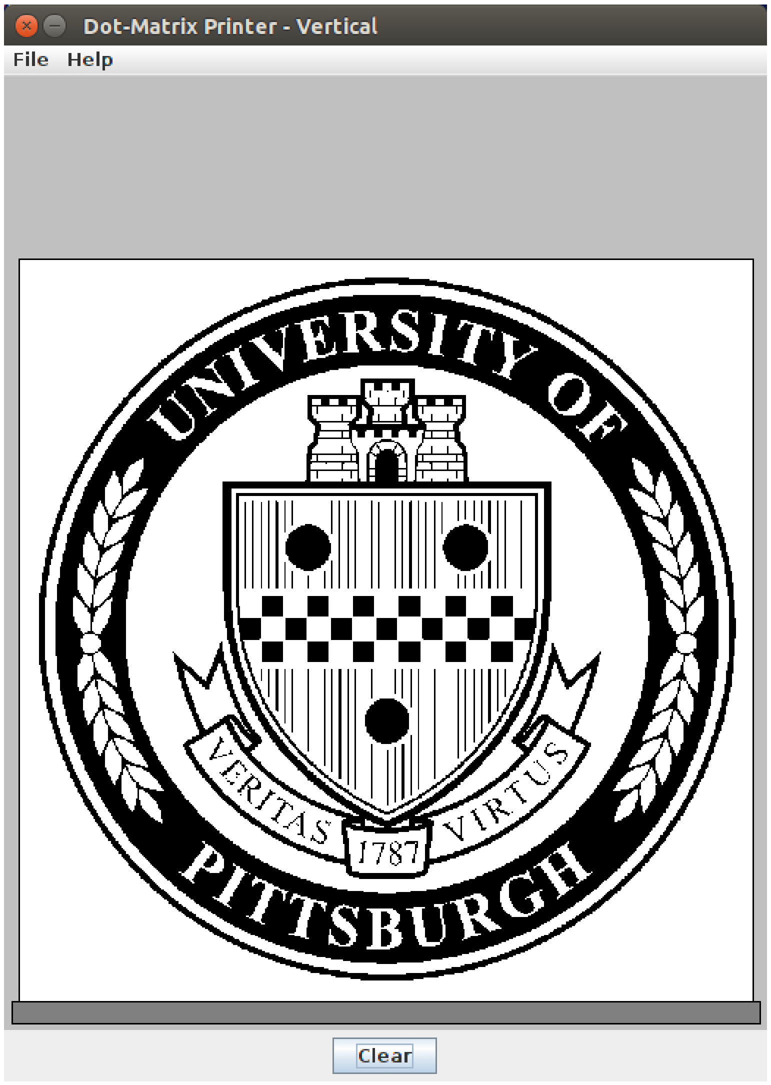
Project 2 - Dot-Matrix Printer (Vertical)

CS 0447 — Computer Organization & Assembly Language

Check the Due Date on the CourseWeb

The purpose of this project is for you to practice writing assembly language to interact with output hardware. The hardware for this project is a primitive dot-matrix printer. The Dot-Matrix Printer (Mars Tool) that we are going to use for this project is shown below. This tool can be found in `DotMatrixPrinterVerticalRegister.zip` located in the CourseWeb under this project. Extract all files to your `[..]/mars4_5/mars/tools` directory. If you extract all files to the right directory, when you run the MARS program, you should see "Dot-Matrix Printer - Vertical (Register) V0.1" under the "Tools" menu.



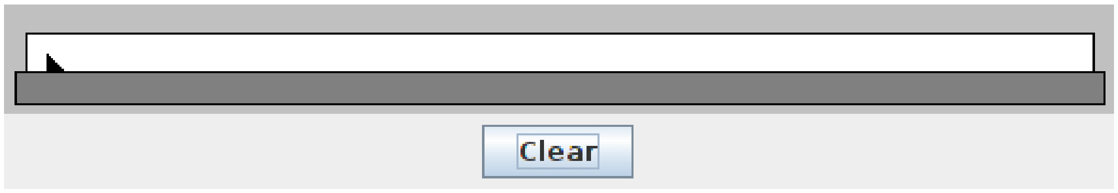
Introduction to the Dot-Matrix Printer (Mars Tool)

The Dot-Matrix Printer - Vertical (Mars Tool) is a very primitive printer. The print head consists of only 8 pins arranged as a vertical line. Therefore, it can only print at most eight vertical dots at a time. This print head only prints when it moves from left to right. Once the print head moves all the way to the right, it will stop printing, move the print head back all the way to the left, feed the paper up just a little bit (equal to the height of eight dots), and then continue printing. For this printer, each row consists of 480 dots.

To print to this printer, we have to write an 8-bit data to the register `$t8` and then set the register `$t9` to 1 to tell the printer that the data is available to be printed. When the printer finishes printing the given 8-bit data, it will set the register `$t9` to 0 to let you know that it is ready to receive the next data. An 8-bit data will be mapped to 8 dots on the paper. A 1 represents a black dot, and a 0 represents no dots. For example, consider the following program:

```
.text
      add  $t9, $zero, $zero   # Clear $t9 to 0
      ori  $s0, $zero, 0xff    # $s0 contains 8-bit data to be printed
      add  $s1, $zero, $zero   # Counter
      addi $s2, $zero, 8       # $s2 = 8
loop:
      beq  $s1, $s2, done      # Check whether counter == 8
      add  $t8, $zero, $s0     # Set $t8 to 8-bit data to be printed
      addi $t9, $zero, 1       # Set $t9 to 1 to print
wait: bne  $t9, $zero, wait    # Wait until $t9 is back to 0
      srl  $s0, $s0, 1         # Change the 8-bit data by shifting it to the right by 1
      addi $s1, $s1, 1         # Increase the counter by 1
      j    loop                # Go back to loop
done:
      addi $v0, $zero, 10      # Syscall 10: Terminate program
      syscall                  # Terminate the program
```

The result of the above program is shown below:



As you may notice, the MSB of an 8-bit data corresponds to the dot at the top and the LSB corresponds to the dot at the bottom. Let's look at the data that we set to the register `$t8` iteration-by-iteration:

```
10000000
11000000
11100000
11110000
11111000
11111100
```

```
11111110
11111111
^      ^
|      |
|      +- $t8 (last iteration) is 0x01
+- $t8 (first iteration) is 0xff
```

So, every time you set the register `$t8` to something and set `$t9` to 1, the printer will print one vertical line consisting of eight dots. The next time you set `$t9` to 1 again, it will print another vertical line next to the previous line. However, if the printer already print the right-most vertical line, it will move the print head back all the way to the left, feed the paper up just a little bit (equal to the height of eight dots).

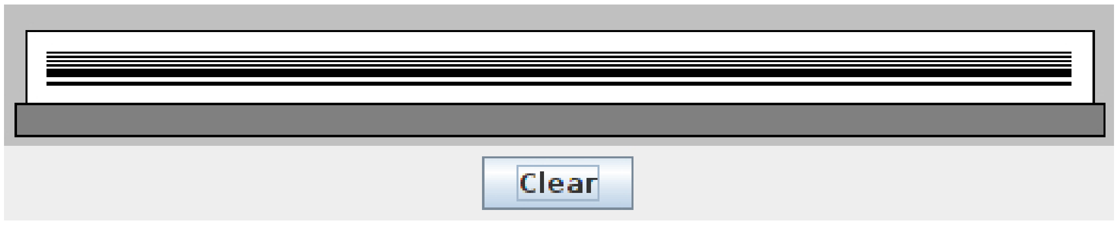
Horizontally, there are total of 480 dots. To print the whole line, you need to send the total of 480 8-bit data to the printer. From the above code example, instead of terminate the program, if you set `$t8` to 0 and send it to the printer (set `$t9` to 1 and wait until it is back to 0) for $480 - 8 = 472$ times, the print head will be right under the first vertical line (all the way to the left).

The "Clear" button is used to reset the printer. Note that you should stop your program before click the "Clear" button. Otherwise, the printer may keep printing.

Let's look at another example. Consider the following code:

```
.text
       add  $t9, $zero, $zero      # Clear $t9 to 0
       add  $s0, $zero, $zero      # $s0 is a counter
       addi $s1, $zero, 480        # $s1 = 480
       ori  $t8, $zero, 0xaa       # $s0 = 0...010101010
loop1: beq  $s0, $s1, done1        # Check whether $s0 == $s1
       addi $t9, $zero, 1          # Set $t9 to 1 (to print)
wait1: bne  $t9, $zero, wait1      # Wait until $t9 is 0
       addi $s0, $s0, 1            # Increase counter by 1
       j    loop1                  # Go back to loop 1
done1: add  $s0, $zero, $zero      # $s0 = 0 (counter)
       ori  $t8, $zero, 0xf3       # $s0 = 0...011110011
loop2: beq  $s0, $s1, done2        # Check whether $s0 == $s1
       addi $t9, $zero, 1          # Set $t9 to 1 (to print)
wait2: bne  $t9, $zero, wait2      # Wait until $t9 is 0
       addi $s0, $s0, 1            # Increase counter by 1
       j    loop2                  # Go back to loop 1
done2: addi $v0, $zero, 10         # Syscall 10: Terminate program
       syscall                     # Terminate the program
```

The result of the above program is shown below:



IMPORTANT Make sure you are familiar with this printer hardware. Think about a simple pattern and try to write a program to print it to ensure that you understand how to control this printer correctly. Do not forget to press the "Clear" button before you run your program.

What you need to do for this project is to print an image onto this printer hardware. The image file format that we are going to use is a simple BMP (indexed with only black and white) format.

Read Data From a File

This project requires you to read data from bitmap (BMP) files. So, let's see how to read data from a file in MIPS assembly.

In this section, we are going to learn how to read the file named `testFile.bin` which contains a 14-byte data as shown below

```
42 4D 7B 00 00 00 0B 00 16 00 C8 01 00 00
```

The first two bytes contains two character `'B'` (0x42 in ASCII) and `'M'` (0x4D in ASCII). The next four bytes is a 4-byte integer. Note that MIPS is a little-endian. The four byte integer shown above is `7B 00 00 00` which is 0x0000007B in little-endian format. This number is 123 in decimal. The next two bytes is a 2-byte integer 0x000B (again in little-endian) which is 11 in decimal. The next two byte is another 2-byte integer 0x0016 which is 22 in decimal. The last four bytes is a 4-byte integer 0x000001C8 which is 456 in decimal.

To read a file, we need to open the file first. This can be done by system call number 13. To open a file, set `$v0` to 13, set `$a0` to the address of null-terminated string containing filename, set `$a1` to 0, set `$a2` to 0, and execute the `syscall` instruction. For example, to open the file `test.bin`, simply use the following sequence of instructions:

```
.data
       filename:  .asciiz  "testFile.bin"
.text
       addi $v0, $zero, 13          # Syscall 13: Open file
       la   $a0, filename           # $a0 is the address of filename
       add  $a1, $zero, $zero       # $a1 = 0
       add  $a2, $zero, $zero       # $a2 = 0
       syscall                      # Open file
       add  $s0, $zero, $v0         # Copy the file descriptor to $s0
```

The result is a **positive** integer represents a file descriptor in `$v0`. Note that if the file descriptor in `$v0` is a negative value, it means the program cannot open the file. Most likely cause of this problem is the file cannot be found. You have to make sure that the file that you want to open is located in the root directory of your Mars program. So, it is a good idea to always check the file descriptor. Your program should notify user and possibly terminate the program if the file descriptor is negative.

Once the file is opened, you can simply read the file using the system call 14 with the returned file descriptor. The system call 14 needs three arguments; (1) the file descriptor in `$a0`, (2) the address of input buffer in `$a1`, and (3) the maximum number of characters (bytes) to read in `$a2`. The following sequence of instructions read the first two bytes of the file `testFile.bin`, put the data in the buffer named `firstTwo` assuming that the file is successfully opened and the file descriptor is stored in `$s0`, and print the first two bytes as character using syscall number 11:

```
.data
       firstTwo:  .space  2
.text
       :
       addi $v0, $zero, 14          # Syscall 14: Read file
       add  $a0, $zero, $s0         # $a0 is the file descriptor
       la   $a1, firstTwo           # $a1 is the address of a buffer (firstTwo)
       addi $a2, $zero, 2           # $s2 is the number of bytes to read
       syscall                      # Read file
       la   $s1, firstTwo           # Set $s1 to the address of firstTwo
       addi $v0, $zero, 11          # Syscall 11: Print character
       lb   $a0, 0($s1)             # $a0 is the first byte of firstTwo
       syscall                      # Print a character
       lb   $a0, 1($s1)             # $a0 is the second byte of firstTwo
       syscall                      # Print a character
```

Now, we are going to read the next 12 bytes in one system call. Note that there are two 4-byte integers (words). For simplicity, we need a 12-bytes buffer but it **MUST** align with word boundary so that we can simply load the whole word at once using the `lw` instruction. The following sequence of instruction read the last twelve bytes and print them out according to data size:

```
.data
       :
       .align 2
       lastTwelve:  .space  12
.text
       :
       addi $v0, $zero, 14          # Syscall 14: Read file
       add  $a0, $zero, $s0         # $a0 is the file descriptor
       la   $a1, lastTwelve         # $a1 is the address of a buffer (lastTwelve)
       addi $a2, $zero, 12          # $s2 is the number of bytes to read
       syscall                      # Read file
       la   $s1, lastTwelve         # Set $s1 to the address of lastTwelve
       addi $v0, $zero, 1           # Syscall 1: Print integer
       lw   $a0, 0($s1)             # $a0 is the first 4-byte integer
       syscall                      # Print an integer
       lh   $a0, 4($s1)             # $a0 is the first 2-byte integer
       syscall                      # Print an integer
       lh   $a0, 6($s1)             # $a0 is the second 2-byte integer
       syscall                      # Print an integer
       lw   $a0, 8($s1)             # $a0 is the second 4-byte integer
       syscall                      # Print an integer
```

When you are done reading the file, the file must be closed. This can be done by system call 16. Simply set the file descriptor of the file to be closed in `$a0`, set `$v0` to 16, and execute the syscall instruction as shown below:

```
        add  $v0, $zero, 16          # Syscall 16: Close file
        add  $a0, $zero, $s0         # $a0 is the file descriptor
        syscall                      # Close file
```

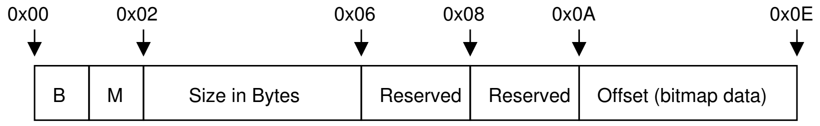
The complete code of the above program `readFileExample.asm` including the file `testFile.bin` can be found on the CourseWeb under this project.

Introduction to BMP File Format

The first two bytes of a BMP file should begin with the magic numbers 0x42 followed by 0x4D. These two numbers are simply the character `'B'` followed by the character `'M'`. The next 12 bytes is a header that contains the following data:

Offset	Size (bytes)	Description
0x02	4	Size of this BMP file in bytes
0x06	2	Reserved
0x08	2	Reserved
0x1A	4	Offset of the bitmap data

Note that offset 0x02 means 2 bytes from the beginning of the file. So, if we put the magic numbers together with the head it will look like the following:



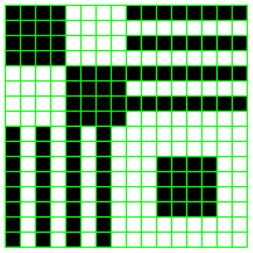
Note that BMP files use little-endian format which compatible with MIPS. So, to read a 4-byte data as an integer, simply read the whole world as shown in previous section. **Be careful with alignment. You may need to use `.align` to ensure that your buffer aligned with word boundary**.

Immediately after the header at the offset 0x0E, it is the beginning of Bitmap Information (DIB) header. A DIB header does not have a specific size. Therefore, the first four bytes of a DIB header contains an integer value represents the size of the DIB header (including this 4 bytes). This will allow programmer to correctly allocate memory and read the whole DIB header. **Note** that you need to allocate memory using system call number 9 using the size specified in the first four bytes of the DIB header but you need to subtracted by 4 since the size includes the first four bytes. Similar to the header, DIB header contain bitmap information as shown below:

Offset	Size (bytes)	Description
0x0E	4	Size of DIB header
0x12	4	Image width (in number of pixels)
0x16	4	Image height (in number of pixels)
0x1A	2	Number of color planes being used
0x1C	2	Number of bits per pixel
0x1E	4	Compression method (0 for no compression)
0x22	4	Size of the raw bitmap data
0x26	4	Horizontal resolution in pixels per meter
0x2A	4	Vertical resolution in pixels per meter
0x2E	4	Number of colors in the color palette
0x32	4	Number of important colors used

Immediately after the DIB header is an array of colors. A color consists of four value (one byte each), blue, green, red, and alpha. Since we are working with 1-bit color (black and white), the array of colors should only contains two colors, black (0, 0, 0, 0) and white (255, 255, 255, 0). **Note** that the order of colors does matter. If black is the first color, it means index 0 is for black and index 1 is for white.

For better understanding of BMP file (indexed and black and white), let's consider the black and white image below:



The above image is a 16×16 black and white image (`testPattern01.bmp`). Green lines are used to separate pixels. The content of the file is as follows:

```
0x00 | 42 4D C2 00 00 00 00 00 00 00 82 00 00 00 6C 00
0x10 | 00 00 10 00 00 00 10 00 00 00 01 00 01 00 00 00
0x20 | 00 00 40 00 00 00 13 0B 00 00 13 0B 00 00 02 00
0x30 | 00 00 02 00 00 00 42 47 52 73 00 00 00 00 00 00
0x40 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x50 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x60 | 00 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00
0x70 | 00 00 00 00 00 00 00 00 00 00 FF FF FF 00 00 00
0x80 | 00 00 AA 00 00 00 AA 00 00 00 AA 3C 00 00 AA 3C
0x90 | 00 00 AA 3C 00 00 AA 3C 00 00 AA 00 00 00 AA 00
0xA0 | 00 00 0F 00 00 00 0F FF 00 00 0F 00 00 00 0F FF
0xB0 | 00 00 F0 00 00 00 F0 FF 00 00 F0 00 00 00 F0 FF
0xC0 | 00 00
```

Numbers on the left column are offsets from the beginning of the file. Now, consider the first 14 bytes (the magic numbers and header)

Offset	Size (bytes)	Short Description	Hexadecimal (little endian)	Decimal
0x00	2	Magic Number	42 4D	`BM`
0x02	4	File Size	C2 00 00 00	194
0x06	2	Reserved	00 00	0
0x08	2	Reserved	00 00	0
0x0A	4	Offset (data)	82 00 00 00	130

The above information tells us that this is a BMP file (magic number), the file size is 194 bytes, and the bitmap data is located at 130 byte from the beginning of the file.

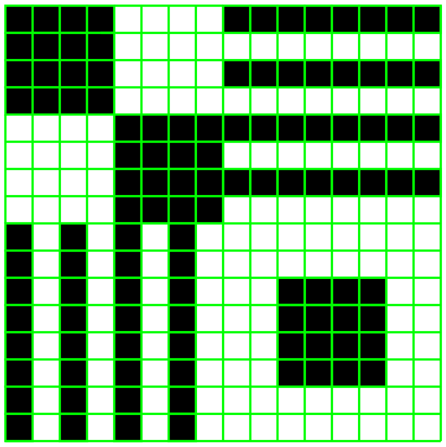
The next four bytes at the index 0x0E represents the size of DIB header. In this case, the value (hexadecimal little endian) is 6C 00 00 00 which is 108 in decimal. Note that once you see the size of the DIB header, you should read the whole DIB header at once for simplicity. Note that the number of bytes to read is now 108 - 4 = 104 since we already read the first four bytes. So, let's put the first 40 bytes of the DIB header into a table, we have

Offset	Size (bytes)	Short Description	Hexadecimal (little endian)	Decimal
0x0E	4	DIB size	6C 00 00 00	108
0x12	4	Width	10 00 00 00	16
0x16	4	Height	10 00 00 00	16
0x1A	2	Color plane	01 00	1
0x1C	2	Bit per pixel	01 00	1
0x1E	4	Compression Method	00 00 00 00	0
0x22	4	Size of bitmap data	40 00 00 00	64
0x26	4	Horizontal resolution	13 0B 00 00	2835
0x2A	4	Vertical resolution	13 0B 00 00	2835
0x2E	4	Number of colors	02 00 00 00	2
0x32	4	Colors used	02 00 00 00	2

Note that the above information stated that the number of colors is 2. So, the next $4 \times 2 = 8$ bytes is the array of colors (4 bytes for each color). In this case, the offset of the color array is $2 + 12 + 108 = 122 = $ 0x7A bytes from the beginning of the file. At 0x7A, we have two colors FF FF FF 00 and 00 00 00 00. This states that the color at index 0 is white and the color at index 1 is black. Now we are at the beginning of the bitmap data at the offset 0x82 which is also stated in the header at offset 0x0A.

For the bitmap data, each row of pixels must be a multiple of four bytes. An easy way to find the number of bytes for each row of pixel is to divide the size of the raw bitmap data (stated at offset 0x22) by the image height (stated at offset 0x16). In this example, the size of bit map data is 64 and the image height is 16. Thus, each row of pixels is $64/16 = 4$ bytes. **IMPORTANT, the bitmap data is reversed. The first row of the image (top row) is at the end of the bitmap data and the last row of the image (bottom row) is at the beginning of the bitmap data**. Let's put all 64 bytes of bitmap data in the **reverse order** (of row) together with its binary representation and the actual image:

```
Offset | Hexadecimal | Binary
 0xBE  | F0 FF 00 00 | 1111000011111111 0...0
 0xBA  | F0 00 00 00 | 1111000000000000 0...0
 0xB6  | F0 FF 00 00 | 1111000011111111 0...0
 0xB2  | F0 00 00 00 | 1111000000000000 0...0
 0xAE  | 0F FF 00 00 | 0000111111111111 0...0
 0xAA  | 0F 00 00 00 | 0000111100000000 0...0
 0xA6  | 0F FF 00 00 | 0000111111111111 0...0
 0xA2  | 0F 00 00 00 | 0000111100000000 0...0
 0x9E  | AA 00 00 00 | 1010101000000000 0...0
 0x9A  | AA 00 00 00 | 1010101000000000 0...0
 0x96  | AA 3C 00 00 | 1010101000111100 0...0
 0x92  | AA 3C 00 00 | 1010101000111100 0...0
 0x8E  | AA 3C 00 00 | 1010101000111100 0...0
 0x8A  | AA 3C 00 00 | 1010101000111100 0...0
 0x86  | AA 00 00 00 | 1010101000000000 0...0
 0x82  | AA 00 00 00 | 1010101000000000 0...0
```



Recall that index 0 is white and index 1 is black. Do you see how the image on the right is represented by the bitmap data on the left?

Note that this description about BMP file format is only for indexed black and white. There are other different format BMP format. So, test your program with given BMP files only. Your program may not work with other BMP file.

What to do?

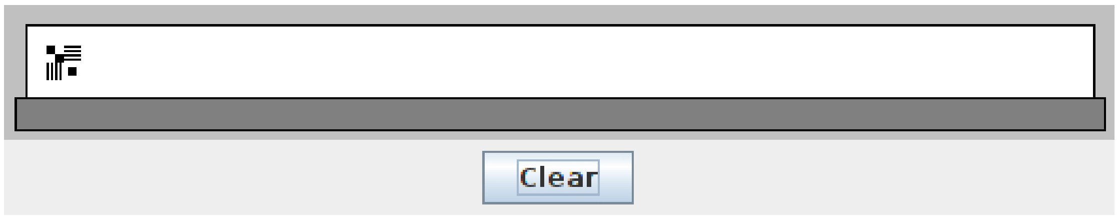
For this project, there are two parts:

1. (50 points) Ask a user to enter a name of a BMP file and display information about the given BMP file on MARS console screen. The following is an example of the output when a user enter an image from previous section `testPattern01.bmp`:

   ```
   Enter a filename: testPattern01.bmp
   The first two characters: BM
   The size of the BMP file (bytes): 194
   The starting address of image data: 130
   Image width (pixels): 16
   Image height (pixels): 16
   The number of color planes: 1
   The number of bits per pixel: 1
   The compression method: 0
   The size of raw bitmap data (bytes): 64
   The horizontal resolution (pixels/meter): 2835
   The vertical resolution (pixels/meter): 2835
   The number of colors in the color palette: 2
   The number of important colors used: 2
   The color at index 0 (B G R): 255 255 255
   The color at index 1 (B G R): 0 0 0
   ```

   Note that your program must work with all given BMP files.

2. (50 points) Print the image of all given BMP files to the printer hardware. For example, the output of the file `testPattern01.bmp` should be as shown below:



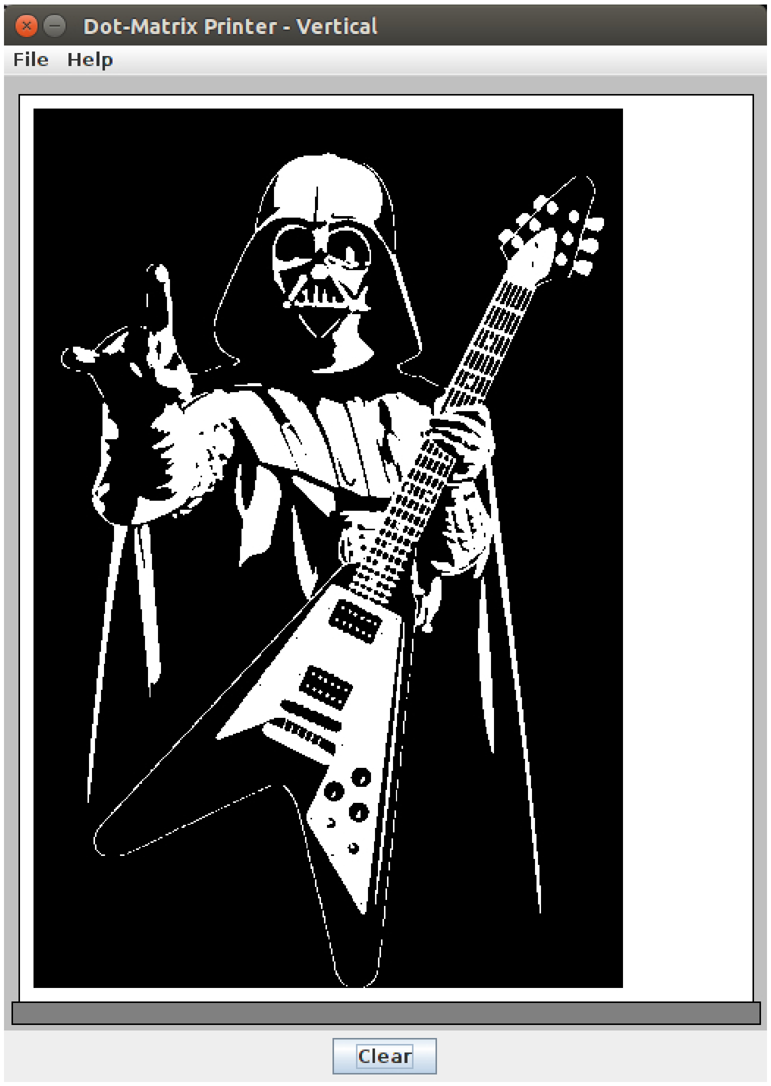
Requirements

The following are requirements that you **must** follow:

- Your program should be named `printer.asm`
- Since the Dot-Matrix Printer - Vertical (Mars Tool) uses registers `$t8` and `$t9`, **do not use** these registers for any other purpose.

HINTS

1. For simplicity, you should separate your program into various functions based on its functionality. Focus on **register sharing** since you may need quite a number of registers. Make sure you follow all calling conventions.
2. **DO NOT FORGET** that the bitmap data of a BMP file is in a reverse order. The last row comes first and the first row comes last.
3. Do not forget about the color indexes. 0 may be black or white depending on the index of color. Check the color index first.
4. **START EARLY**



Submission

The due date of this project is stated on the CourseWeb. Late submissions will not be accepted. You should submit the file `printer.asm` via CourseWeb.