Lab 4

EE 277A Embedded SoC Design

AHB VGA Peripheral

Submitted By

Bhavin Patel

SJSU-ID 015954770

MSEE – Spring 2023

Date – Mar 19, 2023

San Jose State University

Department of Electrical Engineering

INDEX

1. Introduction

This lab involves the implementation of an AHB VGA peripheral and writing a simple program to display images on a VGA monitor using the SoC consisting of Cortex-M0 processor, AHB-Lite bus, and AHB peripherals (Program memory and VGA) on an FPGA. The hardware design involves implementing the Cortex-M0 processor core, AHB-lite bus, memory, and VGA peripheral on an FPGA board. The VGA peripheral has an AHB bus interface, image buffer, and text console. The software design includes initializing the interrupt vector, displaying a text string at the console region, and displaying an image in the image region. The lab aims to provide hands-on experience in implementing a simple SoC, modifying and compiling an assembly code, and analyzing the behavior of the VGA peripheral.

2. Lab Background

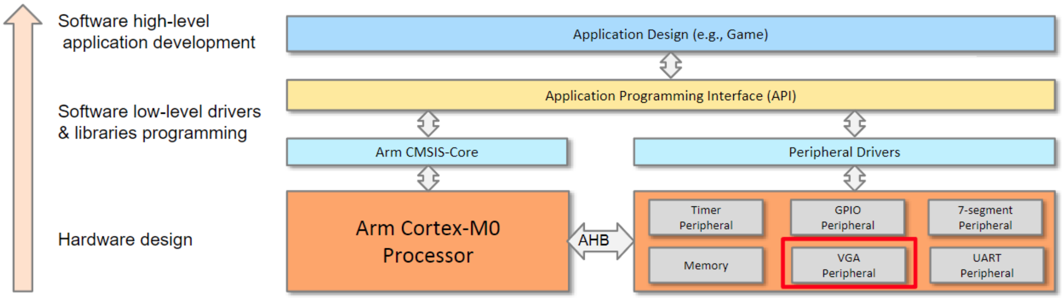

Software high-level application development
Software low-level drivers & libraries programming
Hardware design
Application Design (e.g., Game)
Application Programming Interface (API)
Arm CMSIS-Core
Peripheral Drivers
Arm Cortex-M0 Processor
AHB
Timer Peripheral
GPIO Peripheral
7-segment Peripheral
Memory
VGA Peripheral
UART Peripheral

Fig. 1 Lab Setup

Below is the list of files included for the VGA peripherals in given lab files

VGA Peripherals

1. vga_image.v : The frame buffer in the image region. The module implements frame buffer to hold the image to be displayed on the VGA console. It receives the pixel location and data which are written to the frame buffer. The same data is read by the VGA peripheral to display the image on screen.

2. vga_console.v : The module implements the text region of the VGA interface. Used to display characters in the console region. It receives the ASCII value from the program. The same text is read by VGA peripheral to display text on the screen

3. vga_sync.v : Used to generate VGA synchronization signals. The VGA interface requires vertical synchronization pulses to display the frame in an interlaced manner. The module implements the synchronization pulses.

4. font_rom.v : The ROM used to store the pixels of a character. The module is used to control the text region of the VGA interface. dual_port_ram_sync.v A dual-port synchronized on-chip RAM. The dual port ram is used to store the image and text data to be used by the VGA interface

3. Design Approach

In this lab, I planned to display images on the VGA peripheral with assembly programming. Later, it was extended to an attempt to show video as well. The implementation planning will be discussed in this chapter.

3.1 Data Memory Peripheral

To display an image, it was necessary to include another memory module peripheral to act as a data memory. In Harvard architecture, we use code and data memory instead of just one memory block which stores both. So, I decided to use a separate data memory and not reuse the already existing code memory block.

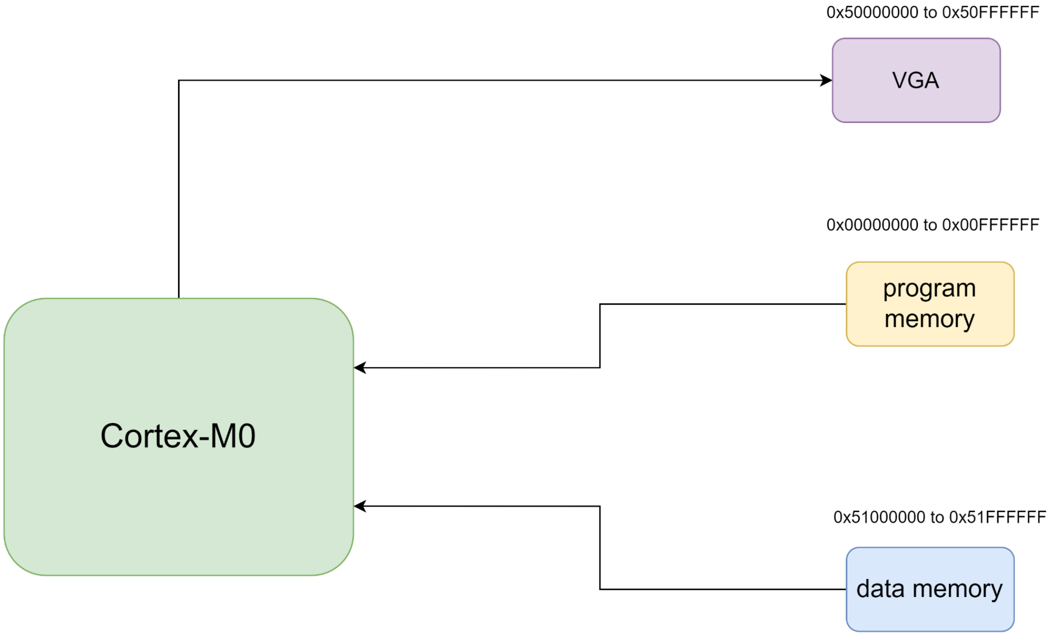


Fig. 2 Adding Data Memory Peripheral.

Moreover, to connect this data memory peripheral, I attached it as a 3rd peripheral to the cortex M0 as shown in the above block diagram. The data memory gets its own memory map and it is connected with the multiplexer and decoder in the main module.

3.2 Memory Map for Data Memory

Operation	Memory Map
Data Read	`0x51000000 to 0x51ffffff`

The data memory is made read only, as we are only reading the images and text and displaying it on VGA monitor. We are not required to write any data back to the data memory. The assembly program can directly read from any given address. Although we can support a maximum of 16 MB data memory with this mapping, it was still good enough to hold multiple images and text information as the resolution of VGA monitor is only 120*140 pixels.

4. Custom Data Format

In order to properly interpret the text and images from the data memory, I defined my own format to store the text and image data in memory. Below is the format I have used.

4.1 Data Element

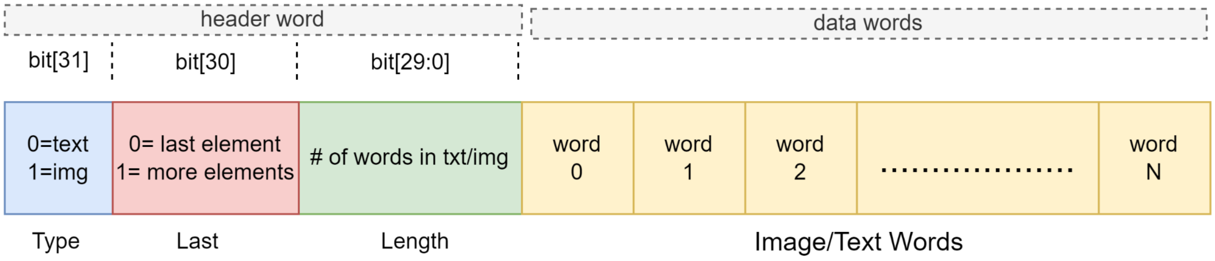


Fig. 3 Data Format Element

This format is inspired by many of the data transfer protocols including Ethernet and CSI2. These protocols uses a header word and following data words. In general,. Compared to these protocols, the format used in this project is relatively simple, but uses the same idea. The reason it is kept simple is because the data memory is on-chip and therefore does not need error checking information embedded in the header. More importantly, the application of just showing the image does not require a lot of information to be put in the header field. In the header word show in in Fig. 3, we have 3 fields which indicate some information about the following image/text data.

Table. 1 Data Element Field Description

Field	Bit Index	Description
Type	31	Indication to interpret data words as text/image
Last	30	Indicates if there is more elements following the current
Length	0 to 29	Indicates how many data words are there
Image/Text Words	-	Each word contains a character(text) or a pixel(image)

Although we have 29 bits for Length field, we only need 14 bits as 120*100 pixels size image requires 12000 pixels and 14 bits are enough to accommodate 2^14=16384 pixels.

4.2 Data Element Arrangement

The final data memory block looks like the below figure, where multiple elements described in chapter 4 section 1 will be stacked one after the other. This is made possible by the use of **Last** bit. On the last element, the Last bit will be 1 to tell the processor to stop reading further bytes.

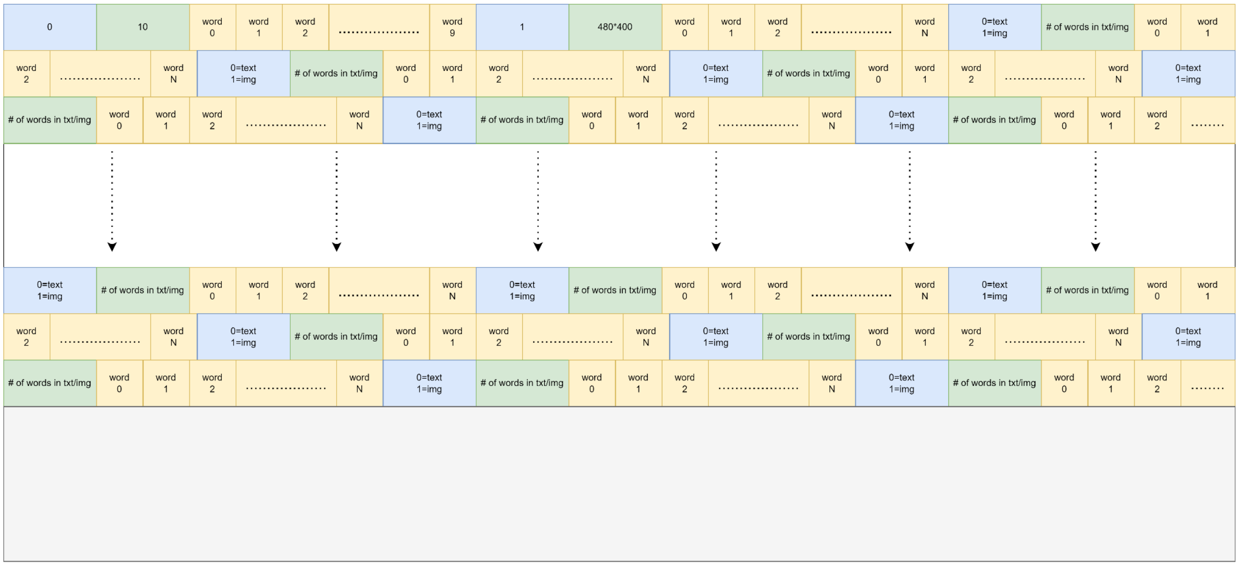

0 | 10 | word 0 | word 1 | word 2 | .... | word 9 | 1 | 480*400 | word 0 | word 1 | word 2 | .... | word N | 0=text 1=img | # of words in txt/img | word 0 | word 1
word 2 | .... | word N | 0=text 1=img | # of words in txt/img | word 0 | word 1 | word 2 | .... | word N | 0=text 1=img | # of words in txt/img | word 0 | word 1 | word 2 | .... | word N | 0=text 1=img
# of words in txt/img | word 0 | word 1 | word 2 | .... | word N | 0=text 1=img | # of words in txt/img | word 0 | word 1 | word 2 | .... | word N | 0=text 1=img | # of words in txt/img | word 0 | word 1 | word 2 | ....
0=text 1=img | # of words in txt/img | word 0 | word 1 | word 2 | .... | word N | 0=text 1=img | # of words in txt/img | word 0 | word 1 | word 2 | .... | word N | 0=text 1=img | # of words in txt/img | word 0 | word 1
word 2 | .... | word N | 0=text 1=img | # of words in txt/img | word 0 | word 1 | word 2 | .... | word N | 0=text 1=img | # of words in txt/img | word 0 | word 1 | word 2 | .... | word N | 0=text 1=img
# of words in txt/img | word 0 | word 1 | word 2 | .... | word N | 0=text 1=img | # of words in txt/img | word 0 | word 1 | word 2 | .... | word N | 0=text 1=img | # of words in txt/img | word 0 | word 1 | word 2 | ....

Fig. 4 Data Element Arrangement

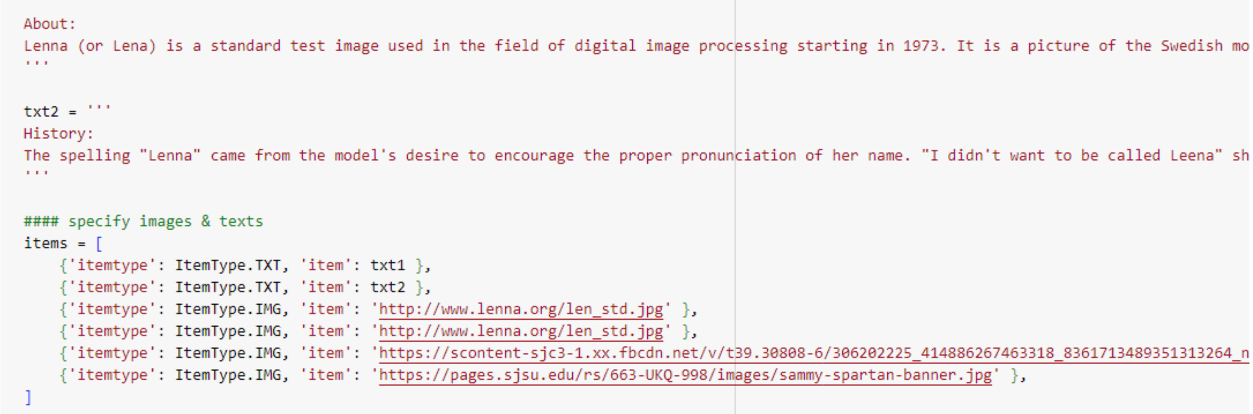
The processor starts with the first element, displays it to the screen, and moves on to the next element as long as the last bit of current element is zero. This type of arrangement makes it possible to extend the support of showing images to also include videos. The only change that was required was to add a frame delay between two image elements in the above format. The hardware timers of Cortex-M0 were used to generate this frame delay. The following chapter will explain how this data memory peripheral is loaded with the image and text data.

5. Generating the Data Memory

To support the format specified in chapter 5, I have written a Python script. The script can automatically generate a HEX file which can directly be loaded in the data memory in Vivado in the same manner we load the code.hex. Below are some of the features of the script.

- Supports Online Images from URL and texts
- Automatically adjusts the image resolution and aspect ratio supported by VGA
- Easier to integrate, it allows you to define sequence in which text and images appear

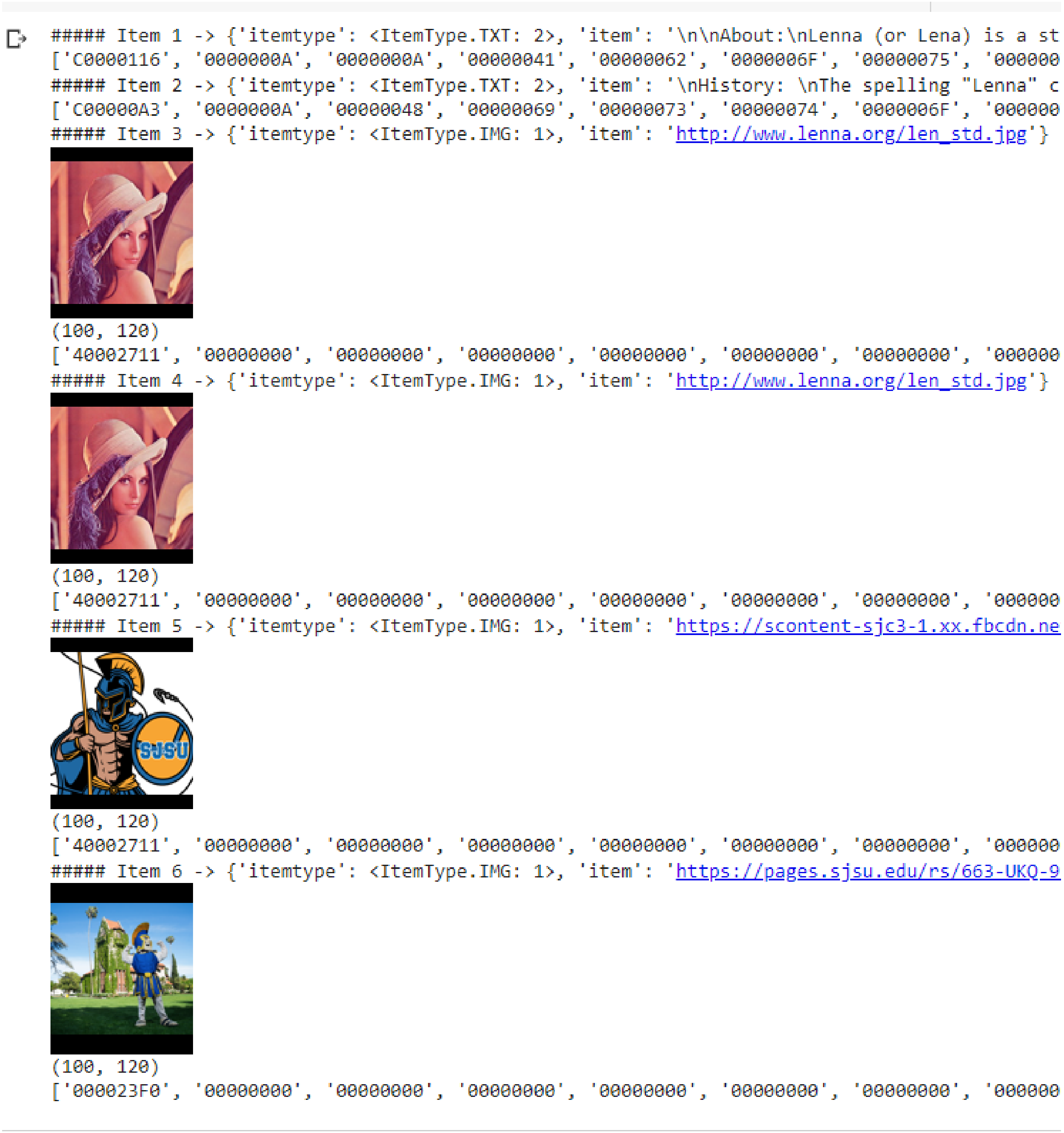
For this lab, I specified below the sequence of images in the script which was then directly transferred to the hex file.



```
About:
Lenna (or Lena) is a standard test image used in the field of digital image processing starting in 1973. It is a picture of the Swedish mo
'''

txt2 = '''
History:
The spelling "Lenna" came from the model's desire to encourage the proper pronunciation of her name. "I didn't want to be called Leena" sh
'''

#### specify images & texts
items = [
    {'itemtype': ItemType.TXT, 'item': txt1 },
    {'itemtype': ItemType.TXT, 'item': txt2 },
    {'itemtype': ItemType.IMG, 'item': 'http://www.lenna.org/len_std.jpg' },
    {'itemtype': ItemType.IMG, 'item': 'http://www.lenna.org/len_std.jpg' },
    {'itemtype': ItemType.IMG, 'item': 'https://scontent-sjc3-1.xx.fbcdn.net/v/t39.30808-6/306202225_414886267463318_8361713489351313264_n
    {'itemtype': ItemType.IMG, 'item': 'https://pages.sjsu.edu/rs/663-UKQ-998/images/sammy-spartan-banner.jpg' },
]
```

Fig. 5 Loading Images and Texts in Python Script

In above, I fill an items array with an element that specifies the type(image/text) and the text string or URL of the image. The program following it then uses this information to retrieve the image from URL and text to convert it in the supported format.



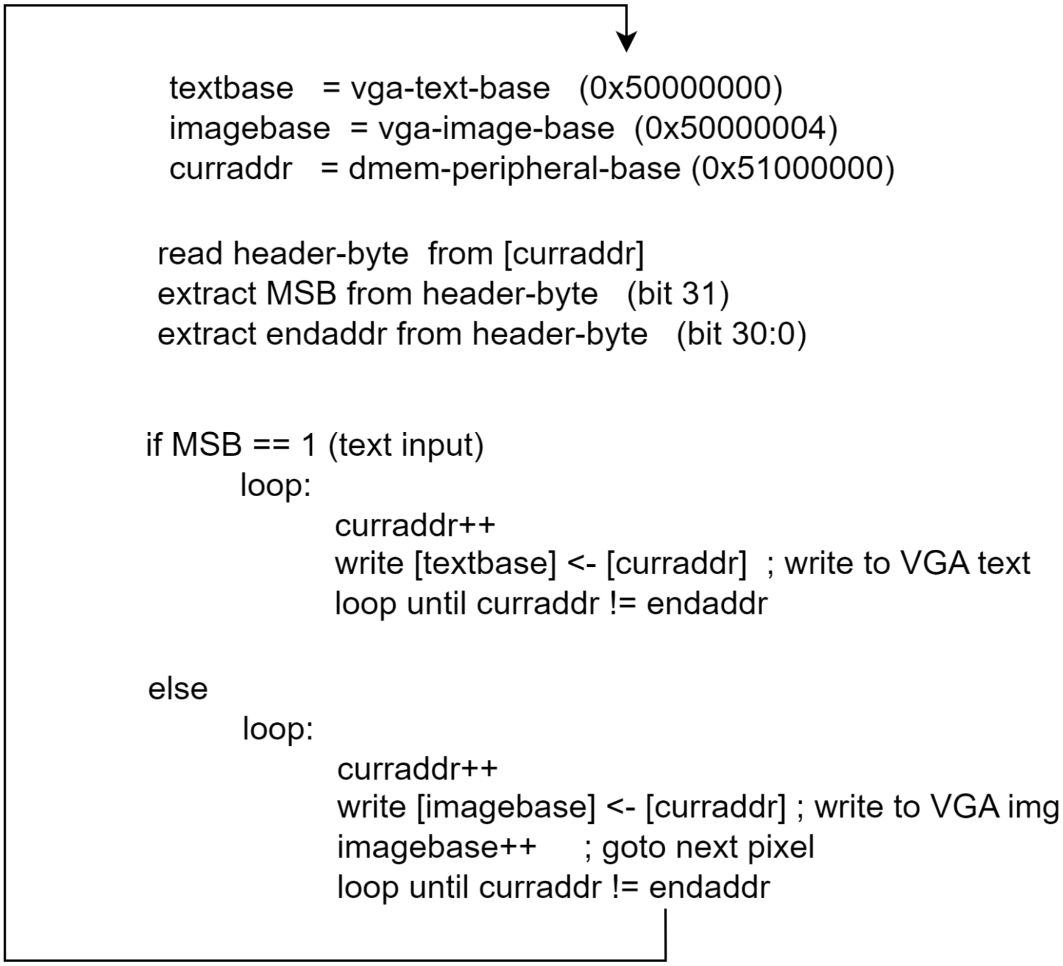
```
##### Item 1 -> {'itemtype': <ItemType.TXT: 2>, 'item': '\n\nAbout:\nLenna (or Lena) is a st
['C0000116', '0000000A', '0000000A', '00000041', '00000062', '0000006F', '00000075', '000000
##### Item 2 -> {'itemtype': <ItemType.TXT: 2>, 'item': '\nHistory: \nThe spelling "Lenna" c
['C00000A3', '0000000A', '00000048', '00000069', '00000073', '00000074', '0000006F', '000000
##### Item 3 -> {'itemtype': <ItemType.IMG: 1>, 'item': 'http://www.lenna.org/len_std.jpg'}
(100, 120)
['40002711', '00000000', '00000000', '00000000', '00000000', '00000000', '00000000', '000000
##### Item 4 -> {'itemtype': <ItemType.IMG: 1>, 'item': 'http://www.lenna.org/len_std.jpg'}
(100, 120)
['40002711', '00000000', '00000000', '00000000', '00000000', '00000000', '00000000', '000000
##### Item 5 -> {'itemtype': <ItemType.IMG: 1>, 'item': 'https://scontent-sjc3-1.xx.fbcdn.ne
(100, 120)
['40002711', '00000000', '00000000', '00000000', '00000000', '00000000', '00000000', '000000
##### Item 6 -> {'itemtype': <ItemType.IMG: 1>, 'item': 'https://pages.sjsu.edu/rs/663-UKQ-9
(100, 120)
['000023F0', '00000000', '00000000', '00000000', '00000000', '00000000', '00000000', '000000
```

Fig. 6 Encoded the Image & Text elements

Once the given image and text data is retrieved, it is converted to hex data with the format specified in chapter 4. The above screenshot shows what data will go in data.hex. The full Python Colab Notebook can be found in the lab submission files.

6. Program Logic

In this lab, assembly programming was used to retrieve the data from data memory and show it in the text and image region of VGA display. Below image shows the program logic.



```
textbase   = vga-text-base   (0x50000000)
imagebase  = vga-image-base  (0x50000004)
curraddr   = dmem-peripheral-base (0x51000000)

read header-byte  from [curraddr]
extract MSB from header-byte   (bit 31)
extract endaddr from header-byte   (bit 30:0)


if MSB == 1 (text input)
        loop:
                curraddr++
                write [textbase] <- [curraddr]  ; write to VGA text
                loop until curraddr != endaddr

else
        loop:
                curraddr++
                write [imagebase] <- [curraddr] ; write to VGA img
                imagebase++     ; goto next pixel
                loop until curraddr != endaddr
```

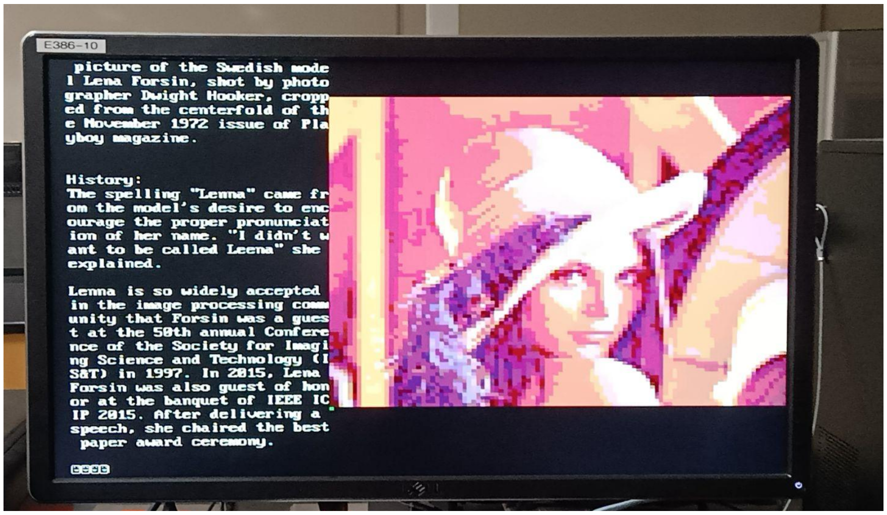
Fig. 7 Program Logic

In this program, first we initialize the base addresses and we start reading the first byte of data memory which is the header byte of the first element. Then we check if it is an image or text data and accordingly branch to the loop which is used to read the consecutive data from data memory and put it to the VGA image/text base addresses. We also keep the Length field from header byte, and based on that, when we are done reading all the byte, we go to the next element. Before we branch, we first check if there is one more element following it, and if not, we end the program. The full assembly code is attached in the lab files for reference, the code is slightly modified to move the loops inside separate functions.

There are mainly 3 functions in the program, **showtxt** loop to read a character from data memory and show it in the text region. **SHOWLINE** function reads 100 words from memory and shows it to the image region. Additionally, **SHOW_BLANKING** function is used to consider the blanking region of 28 pixels . Finally, the **showimg** loop repeats consecutive calls of SHOWLINE followed by SHOW_BLANKING to display each line of the image(120 lines) and then branch back to the next element.

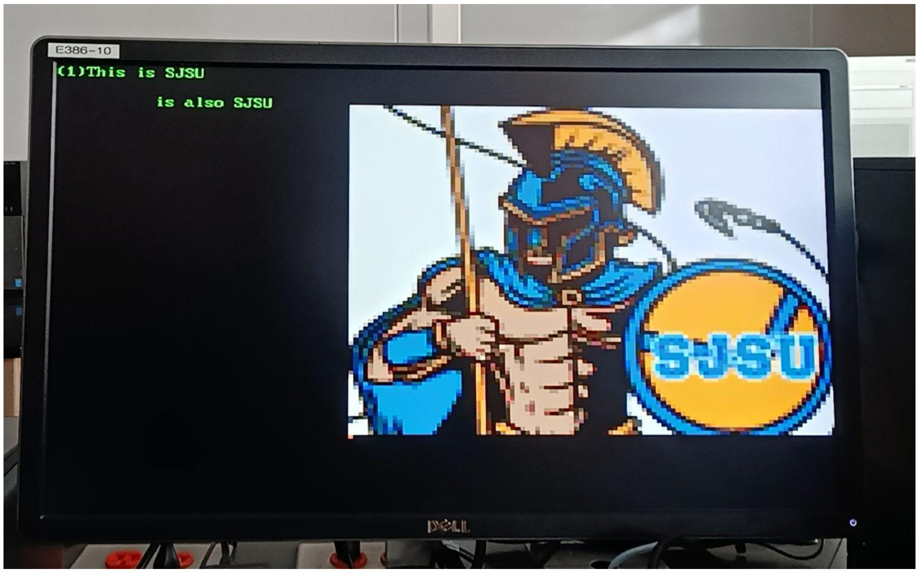
7. Results

The above results show the image that was displayed with the assembly code. The format also supported a delay, and I used a DELAY_LOOP to generate the frame delay and then show consecutive images in between, but it could not work probably because the V-Sync of VGA was not getting triggered. Therefore it only shows one image at a time with text. The text region, however, supported multiple elements. The below two images with text I displayed, one is the famous Lenna image, the other is SJSU Spartan.



```
 picture of the Swedish mode
l Lena Forsin, shot by photo
grapher Dwight Hooker, cropp
ed from the centerfold of th
e November 1972 issue of Pla
yboy magazine.


History:
The spelling "Lenna" came fr
om the model's desire to enc
ourage the proper pronunciat
ion of her name. "I didn't w
ant to be called Leena" she
explained.

Lenna is so widely accepted
 in the image processing comm
unity that Forsin was a gues
t at the 50th annual Confere
nce of the Society for Imagi
ng Science and Technology (I
S&T) in 1997. In 2015, Lena
Forsin was also guest of hon
or at the banquet of IEEE IC
IP 2015. After delivering a
speech, she chaired the best
 paper award ceremony.
```

Fig. 8 Lenna Image with Text



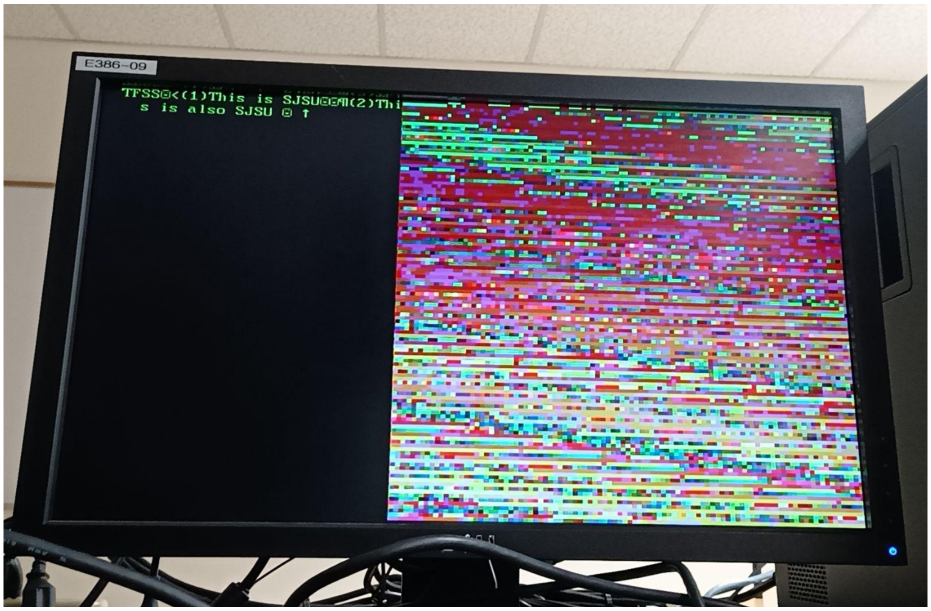
```
(1)This is SJSU

          is also SJSU
```

Fig. 9 SJSU Spartan Image with Text

8. Challenges

There were many other issues while coding the program, and interfacing the peripherals, such as limited number of registers available to write the whole program, trouble using the Vivado *updatemem* to update the data memory block with more images and text. However, the below are two of the main challenges I encountered, and it was interesting to solve them.

8.1 Pixel Color Format



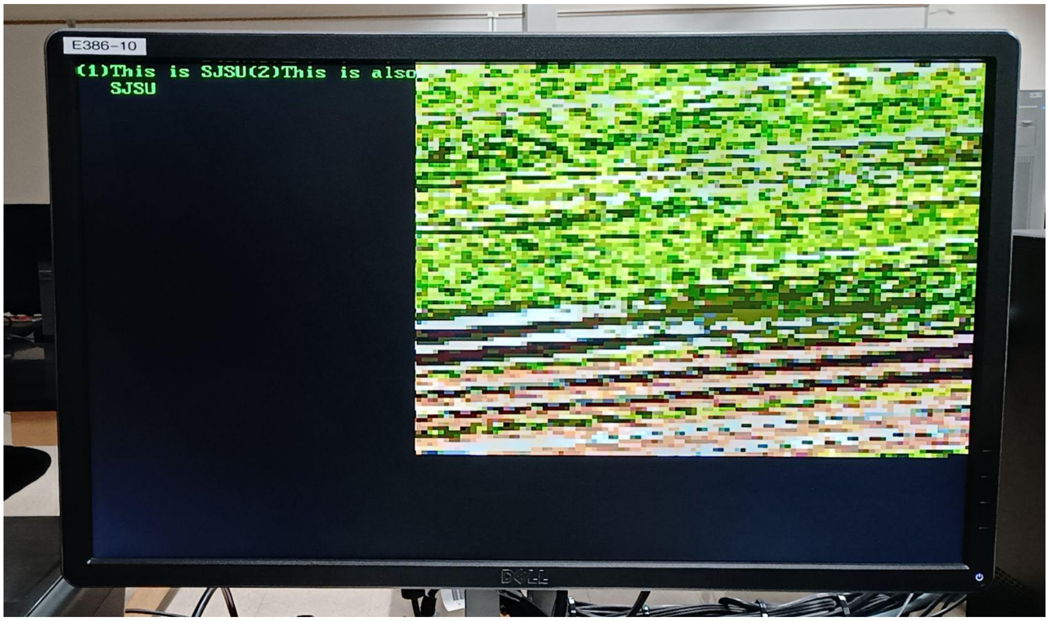
```
TFSS⊡<(1)This is SJSU⊡⊡M(2)Thi
  s is also SJSU ⊡ ↑
```

Fig 10. Issue with RGB Format

The VGA supports RGB332 format. But initially I did not consider it and instead I generated data.hex with RGB 888 format. Therefore, as you can see in the above image, the pixel values are incorrect. However, it can also be observed that there is a pattern which indicates the presence of an image. Once the issue was identified, the Python script was updated to produce a data.hex file with RGB332 pixel encoding.

8.2 Blanking Region Misalignment

The VGA image region addresses for a frame are not continuous. Every line of 100 pixels of image is followed by 28 blanking pixels. Therefore, the first line starts at 0x00000, second line at 0x000200, third line at 0x000400 and so on. Ever line occupies 0x200 addresses out of which 0x190 are used for image pixels and remaining 0x70 are used for blanking pixels.



```
(1)This is SJSU(2)This is also
   SJSU
```

Fig. 11 Image Line Misalignment due to Ignoring Blanking Region

In the above image, it can be seen that the lines are skewed because the 28 pixels at the end of each line are lost in the blanking region, so every other line has 28 pixels right shifted from the line before them. A significant amount of time was spent on finding the fault, and then adjusting the assembly code to account for the blanking pixels. It was fixed by separating the image display loop with two functions SHOW_LINE and SHOW_BLANKING. For every image line, the main image display loop(showimg) calls SHOW_LINE followed by SHOW_BLANKING until all the lines are displayed.

7. Conclusion

The lab was successfully completed by displaying an image on a VGA monitor using assembly programming. It helped understand the VGA format, how to interface additional peripherals including data memory. It also taught how the data memory and code memory is used in Harvard architecture for various applications. The lab had many challenging parts including understanding the VGA format. Another challenge was to write the program in assembly language. The limitation was the lower number of registers available for many instructions and keeping track of all the registers using stack and function calls. Overall the lab helped a lot in learning how to interface extra peripherals and use them with assembly programming, and how challenging it is to write assembly code for larger applications. The following labs including CMSIS further supported the latter point in more detail.

APPENDIX

```
;-------------------------------------------------------------------------------
--------------------------
; Design and Implementation of an AHB VGA Peripheral
; 1)Display text string: "TEST" on VGA.
; 2)Change the color of the four corners of the image region.
;-------------------------------------------------------------------------------
--------------------------

; Vector Table Mapped to Address 0 at Reset

                        PRESERVE8
                        THUMB

                          AREA    RESET, DATA, READONLY               ; First 32
WORDS is VECTOR TABLE
                        EXPORT  __Vectors

__Vectors               DCD     0x00003FFC
                        DCD     Reset_Handler
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0

                        ; External Interrupts

                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
                        DCD     0
```

```
                AREA |.text|, CODE, READONLY
;Reset Handler
Reset_Handler   PROC
                GLOBAL Reset_Handler
                ENTRY



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   IMAGE  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                  LDR      R2, =0x51000000     ; data-mem-base (constant) MEMORY
MAP --> 0x5100_0000 to 0x51FF_FFFF  16MB

showstart
                  LDR     R0, =0x50000000     ; vga-text-base (constant)  MEMORY
MAP --> 0x5000_0000
                  LDR     R1, =0x50000004     ; vga-image-base (constant) MEMORY
MAP --> 0x5000_0004 to 0x50FF_FFFF

                ; read header byte [R3]
                LDR     R3, [R2]

                ; extract image/text indication (from MSB of header) [R5]
                ; 0 = image  1 = text
                LSRS    R5, R3, #31

                ; determine hasMore [R4]
                LSRS    R4, R3, #30
                MOVS    R7, #1
                ANDS    R4, R4, R7

                          ;  determine  address  of  last  char/pixel  from  offset
field(bit[29:0] of header) [R6]
                   ; offset is in #of chars/pixels, each chars/pixel occupy one
word
                LDR     R6, =0x3fffffff         ; bitmask for offset
                ANDS    R6, R3, R6              ; offset in #of words


                ; do looptxt or loopimg based on image/text indication
                CMP     R5, #0
                BEQ     showimg             ; jump for image input


showtxt
                ADDS    R2, R2, #4          ; data-mem-base++
                LDR     R3, [R2]            ; load byte from data-mem-base

                STR     R3, [R0]            ; put byte to vga-text-base

                SUBS    R6, #1
                  CMP     R6, #0                ; check if data-mem-base reached
end address
                BNE     showtxt
                ;BL     DELAY_LOOP      ; debug

                ADDS    R2, R2, #4          ; data-mem-base++ for next element
                CMP     R4, #0              ; end if no more left
                BEQ     showend
                B       showstart


; 0x0004 to 0x190(active) 0x194 to 0x200(blanking)
; 0x0204 to 0x390(active) 0x394 to 0x400(blanking)
; 0x0404 to 0x590(active) 0x594 to 0x600(blanking)
; ...
showimg
                BL      SHOW_LINE
                BL      SHOW_BLANKING

                  CMP      R6, #0               ; check if data-mem-base reached
end address
                BGE     showimg

                ;BL     DELAY_LOOP

                ADDS    R2, R2, #4          ; data-mem-base++ for next element
                CMP     R4, #0              ; end if no more left
                BEQ     showend
                B       showstart

                ENDP


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Shows one line of the image and returns.
; Each line has 100 pixels (0x190/4 = 100)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
SHOW_LINE       PROC
                PUSH    {R3, R4, LR}
                   LDR      R4, =0x190            ; num of pixels in line actual
#pixels = 0x190 / 4
nextpixel
                ; show the next pixel
                ADDS    R2, R2, #4          ; data-mem-base++
                LDR     R3, [R2]            ; load pixel from data-mem-base
                STR     R3, [R1]            ; put pixel to vga-image-base
                ADDS    R1, R1, #4          ; vga-image-base++

                ; return if all pixels in image displayed
                SUBS    R6, R6, #1
                CMP     R6, #0
                BEQ     nomorepixels

                ; return when all the pixels in line displayed
                SUBS    R4, R4, #4
                CMP     R4, #0
                BGT     nextpixel
nomorepixels
                POP     {R3, R4, PC}
                ENDP
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Shows the blanking pixels after each line and returns.
; Each line has 28 blanking pixels (0x70/4 = 28)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
SHOW_BLANKING   PROC
                PUSH    {R3, R4, LR}
                  LDR     R3, =0x70           ; num of pixels in blanking (0x200
- 0x190)
nextblankingpixel
                ; skip blanking pixel
                ADDS    R1, R1, #4          ; vga-image-base++

                ; return when all blanking pixels are done
                SUBS    R3, R3, #4
                CMP     R3, #0
                BNE     nextblankingpixel

                POP     {R3, R4, PC}
                ENDP


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; To generate delay between two frames of a video
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
DELAY_LOOP      PROC
                PUSH    {R3, LR}
                LDR     R3, =0x132DCD5      ;(33.333333 ms)/(10ns) = 0x32DCD5
dloop
                SUBS    R3, R3, #1
                CMP     R3, #0
                BNE     dloop
                POP     {R3, PC}
                ENDP

showend
                ; this pixel appears after the last line of previous image
                ; and not from the first line of new image
                LDR     R3, =0xFF
                LDR     R0, =0x50000004
                STR     R3, [R0]            ;debug

                ALIGN       4                    ; Align to a word boundary

        END
```

<The Python Notebook used to generate the data.hex file containing image and text data is attached with the lab submission files.>