

Digital Logic Design – ECEN 3233 Spring 2009
Oklahoma State University
James E. Stine, Jr., Harsha Choday, and Jun Chen

Project Part II: Building a PacMan Game

Introduction

This document introduces the second part of the project. You need to build a hardware interface between the MIPS processor and the LCD screen. MIPS instructions will be used to fill out the memory buffer and the hardware you design will read from the filled memory to show things on the screen. There will be a simplified Pacman game program provided for you to test your hardware design.

Digital Video with VGA ports

Video has really been something that has revolutionized the electronics industry. At most electronic device stores, video has become a major staple in consumers' purchases. Consequently, you may have noticed that a large number of vendors have produced digital devices that produce digital displays. In fact, video displays have increased orders of magnitude over the last ten years and have been a major asset to many electronics producers. We hope this part of the project will help you acquaint yourself with a little digital video.

Video has traditionally been produced via analog methods. However, due to the advent of digital devices that are able to generate bit streams to accurately represent analog waveforms, most video systems are now produced digitally. We will be working with digital video produced from a 15-pin D-subminiature VGA connector. VGA typically means Video Graphics Arrays and although many digital devices now used alternative methods to output video, the VGA port is still a popular interface on most systems. Figure 1 shows a diagram of the dsub15 connector.

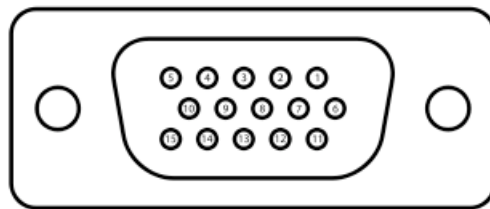


Figure 1: DEF15 DSUB15 VGA Port Connector

VGA connectors were typically common connections, because they could easily attach to most monitors to produce a 640x480 display. The first number in a display usually refers to its width and the second to its height, where the height is typically 2/3 of the width to allow the display to look okay on a television. This width to height ratio is sometimes called an aspect ratio. Today, this aspect ratio is quite different for many different displays, mainly due to the advent of high-definition displays.

For VGA connections, many of the displays are displayed in terms of primary colors. That is, each color can be displayed according to a composite image of Red (R), Green (G), and Blue (B) colors. Most often, RGB colors are patterned in terms of 8-bits making them a total of 24 bits. However, today's images usually are composed of millions of colors. The pin diagram for the VGA connect is listed in Table 1.

Pin Number	Name	Description
1	Red	Red video
2	Green	Green video
3	Blue	Blue video
4	N/C	Not connected
5	Gnd	Ground (HSync)
6	Red_Rtn	Red return (Gnd)
7	Green_Rtn	Green return (Gnd)
8	Blue_Rtn	Blue return (Gnd)
9	+5VDC	+5 V (DDC)
10	Gnd	Ground (VSync, DDC)
11	N/C	Not connected
12	SDA	PC data
13	HSync	Horizontal Sync
14	VSync	Vertical Sync
15	SCL	PC Clock

Table 1 DE15 DSUB15 VGA Connections

For this part of the project, you will interface your VGA connector to gain familiarity with digital video. And, it will also allow you to see the use of something called a driver. A driver is a piece of code that allows software to interface to new hardware, such as a digital display. For example, many video games use drivers or Application Programming Interface (API) calls to allow their software to talk to the hardware on video displays. These APIs are typically called drivers.

A 'Driver' can be a piece of software or hardware that provides an interface between your application and an external component. Maybe you can recollect that Microsoft ® Windows automatically installs a driver each time you connect your FPGA board to the PC. So the Xilinx software sends out the data to the USB cable driver, which in turn, reformats the data into the USB standard format before sending it out on the USB cable. The VGA driver in your project is the hardware that will be responsible for driving signals to the screen. The MIPS processor will write proper color data to the memory and then the VGA driver will read this color data and output it to the screen.

Your screen is composed of several hundred horizontal lines of tiny dots or pixels capable of displaying a color. An image is formed on the screen by giving a color to each pixel on the screen. The color that is displayed in each pixel is determined by a 3-bit binary value for simplicity, however, more bits could be potentially used for more color or depth. Therefore, each color has a 3-bit binary value associated with it (e.g. Blue is represented by '001'). Shown below is a screen that has 640 pixels in each line and 480 such lines on the entire screen. Notice that the origin is in the top-left corner.

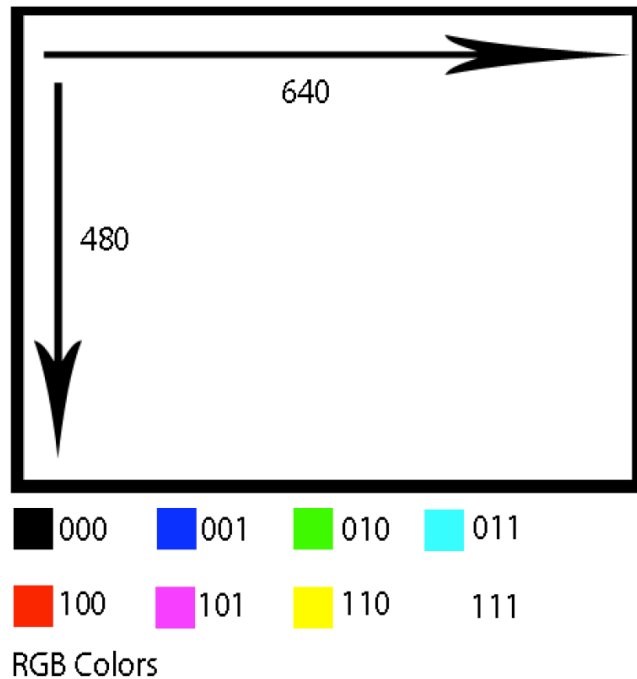


Figure 2: ECEN 3233 Video Display

A VGA driver needs to send color information to the display one line at a time until it reaches to the last line of the screen and then repeat this process again. This is repeated so fast that our eye cannot detect any of this happening. The VGA driver uses two signals to synchronize this operation:

1. A signal to indicate the end of a horizontal line: *hsync*.
2. A signal to indicate the end of the screen: *vsync*.

The first signal (*hsync*), also called as Horizontal sync tells the display that the driver has finished sending the color data for that line. When this signal is asserted by the driver, the display would know that the driver is now going to send the color data for the next line. Once the last line is also displayed, it is time to go back to the first line to begin the display process again. This is indicated by Vertical sync or *vsync*. When the driver asserts *vsync*, the display will know that the next set of color data arriving from the driver would be for the first line.

Video Implementation and Memory Mapping

This design will have the MIPS architecture from part I of this project (including the two new instructions), the VGA driver and three pieces of memory. The memory modules required are: instruction memory, data memory, and storage memory. The instruction memory can be generated in the similar fashion as in the first part of the project. The data memory is an 8K word RAM with some initial data in it. The storage memory is a piece of small ROM which stores small pictures required for the game. The latter two pieces of memory are generated with Single-port Block Memory using the core generator.

Video games have been well touted as a billion-dollar business. They are growing in popularity as well as in complexity. However, many computer games are composed of graphics, code, and digital logic. The digital logic processes the code to move around graphics on the display.

A graphic within a video game is typically called a sprite. The earliest sprites from the 1980s and even the 1990s used 8x8 bit blocks of memory that represented a tile, similar to a checkerboard. For example, if a video game had a monster in it, a piece of memory would hold the many faces of the monster including any movement it would have. That way, when simulating a monster moving, it would just move the sprite for each movement by reference the correct sprite in memory. In today's video games, artists create many of these graphics for monsters or characters by hand. Then, artists transfer images to memory by using an imaging tablet that transfers their designs into memory automatically. However, the process is virtually the same as we are attempting in this project.

Since this course is about digital design, we are giving you the code you will need to use to form the PacMan game, including the sprites. This code will be given to you as a .coe file. However, due to the limited memory size on the FPGA board, the VGA part will drive only 1/4 size of the LCD screen is used. The visible size of the screen is 320 pixels by 200 pixels. The game will be in the lower part of the visible zone.

In the specification of the FPGA board, the VGA port supports 3-bit colors or 8 colors. In the ROM, each pixel is stored using 4 bits with the most significant bit unused. Thus, in the word addressable ROM, each word (32 bits) can hold 8 pixels. Each picture in the ROM is designed to be 8 pixels by 8 pixels. Hence, the screen can be looked as 40 grids by 25 grids. Figure 2 illustrates the dimensions of a screen in terms of grids and pixels as well as the color schemes.

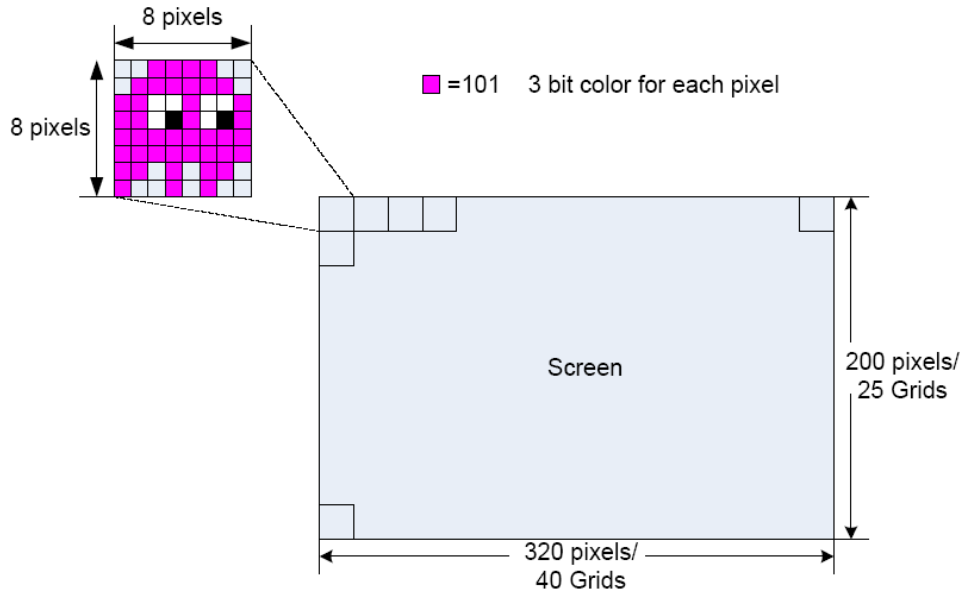


Figure 3: Sprite Data on ECEN 3233 Display

For this project to work correctly, the PacMan program tries to load the data stored in the ROM and write them to the memory buffer. The program uses “lw” (load word) and “sw” (store word) to read and write the memory, respectively. Remember, a word holds 8 pixels and the pictures are 8x8-pixel grids, each word will present a line of a grid.

For the VGA driver, it scans horizontally from pixel 0 to 320 and then vertically, line by line, until all 200 lines of pixels are displayed. As you can imagine, drawing the screen does take a considerable amount of time and for any computer game. The trick is to have fast digital logic that processes all the artificial intelligence associated with a game before a screen is drawn. This process simulates a real-time experience, even though many of these calculations are not done in real-time. The hardware for this project should extract every 3-bit of information in a 32-bit word and assign them to three outputs for a proper display to occur: `red_output`, `green_output` and `blue_output`. Figure 3 shows how a 32-bit word contains the 8 pixel information. Due to the lack of conversion between the video display and the monitor, this board only supports 1 bit per primary color or 8 colors.

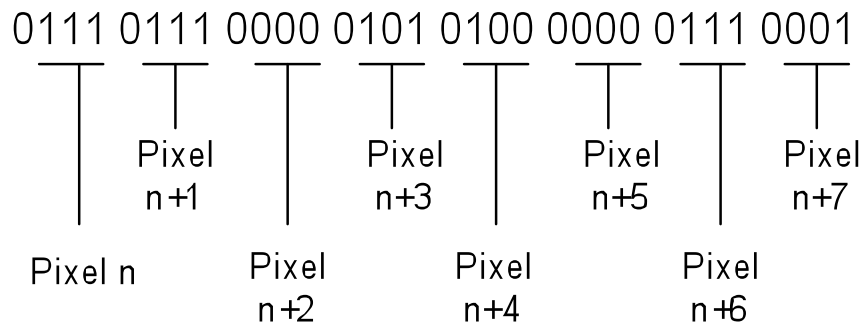


Figure 4 : Tile Memory Map

For this project, the memory is continuous, not like a screen, which will wrap around. Each pixel on the screen will have a corresponding location in the memory buffer, as shown in Figure 4. To accomplish this memory mapping correctly, the screen is 40 grids wide, or 40 words wide, and any pixel information in the memory can be computed using the equation below:

$$\text{Memory location} = \text{row} \times 40 + \text{column}/8.$$

However, it should be noted that row and column are in the units of pixel. To make the math simpler for each row, “column/8” takes only the integer part of the quotient. When the address (location) is computed, the data is fetched and fed to the output driver (red, green and blue). Since one word contains 8-bits of pixel information, the expression to assign the outputs is given as shown below.

```
red_out  <= data[30-(column%8)*4];
green_out <= data[29-(column%8)*4];
blue_out <= data[28-(column%8)*4];
```

The % is the modulo operator, which takes the remainder from divide. However, it is important to remember to set the values to 0 when a reset is asserted.

Another important part of this design is a joystick, which is important since it makes the game player involved with game play. The joystick module should read the user input and converting the direction input into a number. To make things easier and to gain familiarity from previous labs, the joystick will be taken from the four slide switches that each of you used in previous labs. The joystick encoding is given in Table 2.

Command	Operation
Up	0001 ₂
Right	0010 ₂
Down	0100 ₂
Left	1000 ₂

Table 2 : Joystick Encoding

You will need a register to store the number and make it visible to the MIPS processor so it can read that information. The switches on the FPGA board are going to be used for the direction input. It is important to remember that each user needs to assign the package pins accordingly, otherwise the directions will be confused by the processor. (You may have your own choice of having the switches corresponding to the direction, e.g. switch0: up, switch1: right, switch2: down, switch3: left.)

Atari CX40 Joystick Interface

One of the first and most popular joysticks was the Atari 2600 or CX40 joystick. The joystick is quite simple in that it has simple Left, Right, Up, and Down directions along with a Fire button. The joystick uses a simple 9-pin digital interface using the following pinouts:

Pin	Color	Function
1	White	Up
2	Blue	Down
3	Green	Left
4	Brown	Right
5	None	N/C
6	Orange	Fire Button
7	Red	+5 VDC, max. 50 mA
8	Black	Ground
9	None	N/C

The +5 Volts is optional for non-standard Atari joysticks for special-type functions, like auto-fire. The connector of the Atari joystick is designed to work with a D-subminiature connector or Dsub-9 connection, similar to RS-232 connections on slightly older PCs. Although the project can work with the pushbuttons, you must interface one of the Atari joysticks (for extra credit, feel free to use old-joysticks hanging around your house – they work just like the Atari joystick). The joystick commands, such as Up or Down, are simple ON and OFF inputs. However, if its off, it tends to leave it floating. Therefore, typical joysticks are wired to a pull-up resistor, so you might need a 1 k Ω resistor to pull the signal up to VDD (see your TA for help here). In order to protect your circuit, use a 74244 buffer circuit to shield your joystick from the FPGA board. You can use a latch, as well. Your TA can supply you with a 74244 chip.

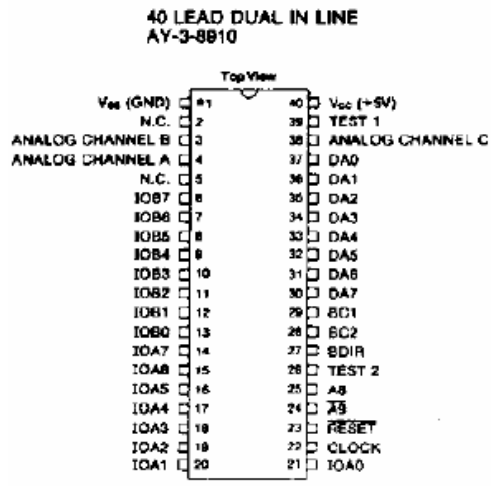
Sound (optional – extra credit available)

One of the coolest things about video games is that they incorporate many facets of engineering. Sound in digital devices is a fairly recent advance and has made video games and their digital television cousins a surreal experience. In fact, many movies now boast tremendous sound capabilities with THX, Dolby, and other types of enhancements to sounds to bolster a movie's feel and look. It can probably be said any video game without any sound can ruin or potentially negatively impact a release. Therefore, understand a little about sound is a very good idea.

Sound has always been quite simple in its implementation. Most systems will involve a clock or oscillator that puts different frequencies out in the range that are audible. For most humans, perception of sound occurs within the 20 Hz – 20 kHz audio range. Even

at the upper and lower parts of this range, most humans only hear within 90% of this audio range. In fact, systems that use compression, such as Dolby, use the human ear's capability to discern only large differences in audio to its advantage in creating better aural surrounding sound. For this project, you will be asked to use the AY 3-89XX sound chip. The AY 3-89XX is a great system made originally by General Instruments in three flavors, 8910, 8912, and 8913 sound chips. These chips are great for early video games and they are built within a 40-pin package and have 8-bit I/O ports to handle generating the appropriate sound.

To generate sound with the 89XX chip, you will need to attach it to the FPGA board similar to how you created the joystick interface. The AY 3-89XX chip is designed to be used with a microprocessor control bus and is quite odd, but luckily is simple to interface. To operate the chip, you just have to control registers accessed through the chip's interface. The pin diagram for the 40-pin AY 3-89XX chip is shown below:



To control the chip you will need to use the chip as a simple asynchronous playback element. That is, connect the output of the FPGA board out to the chip through a simple I/O interface. When making a sound, just output a value to one of the I/O ports and have all three channels of the AY 3-89XX chip connected to a speaker. You can control how often you change the tone, but it would probably be advisable to playback a tone when you detect a joystick movement. This process is typically called a digitization processor sometimes a playback process. That is, each sound is generated by a value coming from the digital device in terms of bits or bytes (e.g. 23, 45, 90, ...) and the higher the value the higher the tone gets generated.

The easiest way to generate a tone is to create a module, similar to the joystick module, that outputs a value to the AY 3-89XX chip through the DA0-DA7 (i.e. pin 30-37) inputs. Please note that you may not have enough pins to accommodate this 8-bit port, however, you only have to generate 1 tone to get extra credit. Each value of the inputs to DA0-DA7 represent a register that controls the tone generated from the chip. Although it seems complicated, it is quite easy in that an 8-bit command. The only trick part is

making sure the A8 and A9 inputs (pin 25 and 24, respectively) get asserted or deasserted properly to allow the register to load properly.

Most sound chips work with an oscillator or clock. You will be using a crystal oscillator that is a type of electronic oscillator. There are many different types of oscillators including relaxation oscillators and harmonic oscillators, however, this project will simplify the process by using a crystal-based oscillator. This oscillator uses a vibrating crystal material inside the device to create a mechanical resonance. This form of electricity has a long history including research dating back to 1880 with Pierre Curie and Sonar generation during World War I. Although there are many oscillators, you will be using a 4 MHz crystal, which should be adequate to generate all the tones necessary for your device.

Implementation Items

The basis of this project is to take the added instruction from part I of this project and implement the code that this is provided to you. Then, get the complete digital system designed and implemented into your Spartan 3E development board. The top.v and memory modules will be different than previous implementations, therefore, these will be provided to you. For extra credit, you can add more functionality to the design by adding more features. However, please consult the TAs or the instructor before you begin, so we can give you ideas of what will and will not work. For this project, you will be required to document your experience either through a poster in ECEN design day or in a project report. Please follow the guidelines for either the poster or project documented on the ecen3233 website. Most importantly, have fun!