# Interfaces on the DE2 board

Oct 26 2015

The following tutorials are meant to help the user become more familiar with the different interfaces on the Altera DE2 board.

The objective of each tutorial is to learn and understand how each interface works and using Verilog code to write drivers. We will use the MSO-X-3024A oscilloscope to verify and look at various signals associated with the interface.

The DE2 board uses the following FPGA [Cyclone II EP2C35F673C]. For more information on this FPGA, go to the following link;

http://www.altera.com/products/fpga/cyclone-series/cyclone-ii/support.html

There are 8 different interfaces on the DE2 board.

1) USB interface – both host and device connectors
2) Audio interface – mic input , line input, speaker output
3) Composite video input – connects to a camcorder with composite video output
4) VGA video output – connector to a VGA monitor
5) Ethernet interface
6) UART interface
7) PS2 interface
8) 40 pin general purpose ports – GPIO-0 (JP1), GPIO-1(JP2)

**Figure 1** shows where all the interfaces are located.

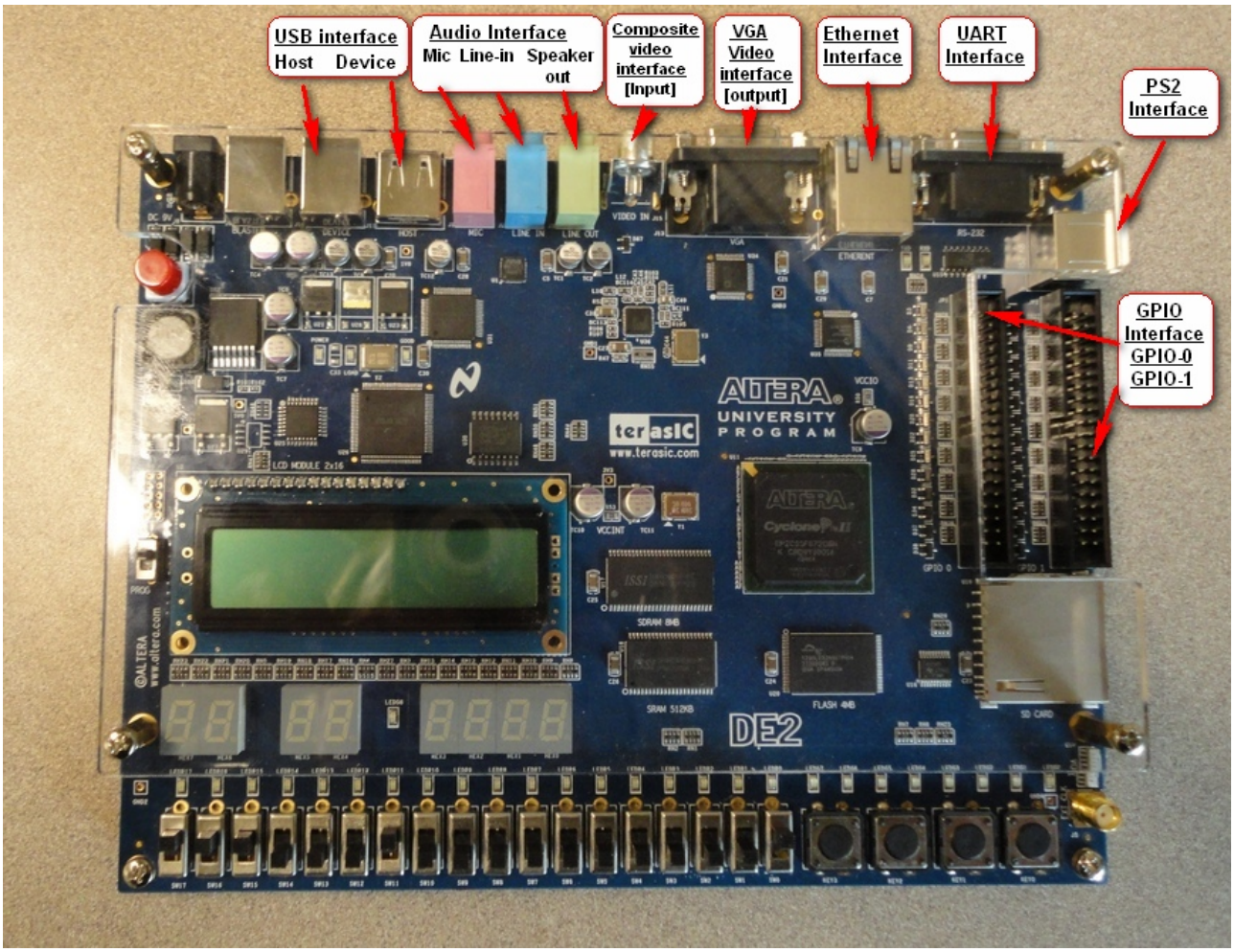


**Figure 1-Description and location of interfaces.**

Let us examine each of the interfaces individually.

## Composite Video Interface

The chip used on the DE2 board is an Analog Devices ADV7181B. For a description of this interface follow the link below;

http://www-ug.eecg.utoronto.ca/desl/manuals/ADV7181.pdf

**Figure 2** is a block diagram of how the interface is connected to the FPGA on the DE2 board. Video is collected from a composite video source such a camcorder with a composite video output.

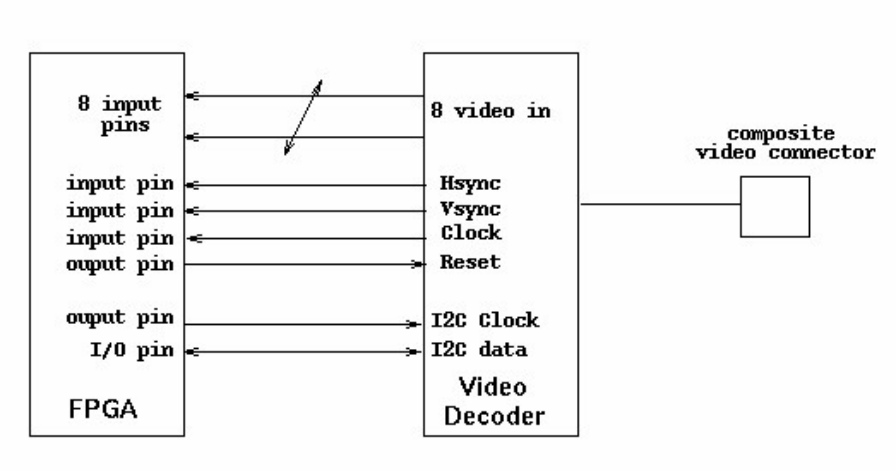


**Figure 2-block diagram of the DE2 FPGA to video Decoder**

**Table 1** is the pin assignments

| Signal Name | FPGA Pin No. | Description |
|---|---|---|
| TD_Data[0] | Pin_J9 | TV Decoder Data[0] |
| TD_Data[1] | Pin_E8 | TV Decoder Data[1] |
| TD_Data[2] | Pin_H8 | TV Decoder Data[2] |
| TD_Data[3] | Pin_H10 | TV Decoder Data[3] |
| TD_Data[4] | Pin_G9 | TV Decoder Data[4] |
| TD_Data[5] | Pin_F9 | TV Decoder Data[5] |
| TD_Data[6] | Pin_D7 | TV Decoder Data[6] |
| TD_Data[7] | Pin_C7 | TV Decoder Data[7] |
| TD_HS | Pin_D5 | TV Decoder H_SYNC |
| TD_VS | Pin_K9 | TV Decoder V_SYNC |
| TD_CLK27 | Pin_D13 | TV Decoder Clock Input |
| TD_RESET | Pin_C4 | TV Decoder Reset |
| TD_SCIk | Pin_A6 | I2C Clock |
| TD_SDAT | Pin B6 | I2C Data |

**Table 1- pin assignments on DE2 FPGA**

- [TV Decoder Data (7:0)]- 8 bits of video data are connected from the video chip to the FPGA. Pins are assigned according to **table 1**.
- [TV Decoder H_SYNC] -Horizontal sync pulse which is generated by the ADV7181 video decoder chip. Pin is assigned according to **table 1**.
- [TV Decoder V_SYNC]- Vertical sync pulse which is generated by the ADV7181 video decoder chip. Pin is assigned according to **table 1.**
- [TV Decoder Clock input]- This is a 27 MHZ clock which is generated by the ADV7181 video chip. In order to enable the clock the TV Decoder Reset pin must be asserted to a high logic level. Note that later versions of the DE2 board the [TV decoder Clock Input]

pin is given a different pin assignment. For all boards in the DESL lab the pin assignment in **table 1** is correct.

- I2C Data- This is a bi-directional serial data bus pin to program the internal serial register of the ADV7181 video decoder. More detailed information about the I2C serial protocol will be given later in this tutorial.
- I2C Clock- This is the serial clock pin that is used to clock the serial data. The frequency that must be generated is typically below 400 KHz. More detailed information about the I2C serial protocol will be given later in this tutorial.

## Background Video Decoding signals.

As described above the ADV7181 video decoder generates 3 pulse signals, the Clock pulse, the horizontal sync pulse and the vertical sync pulse. **Figure 3** shows multiple vertical sync pulses. Note that there are many horizontal and single pixel clock pulses between each vertical clock pulse.

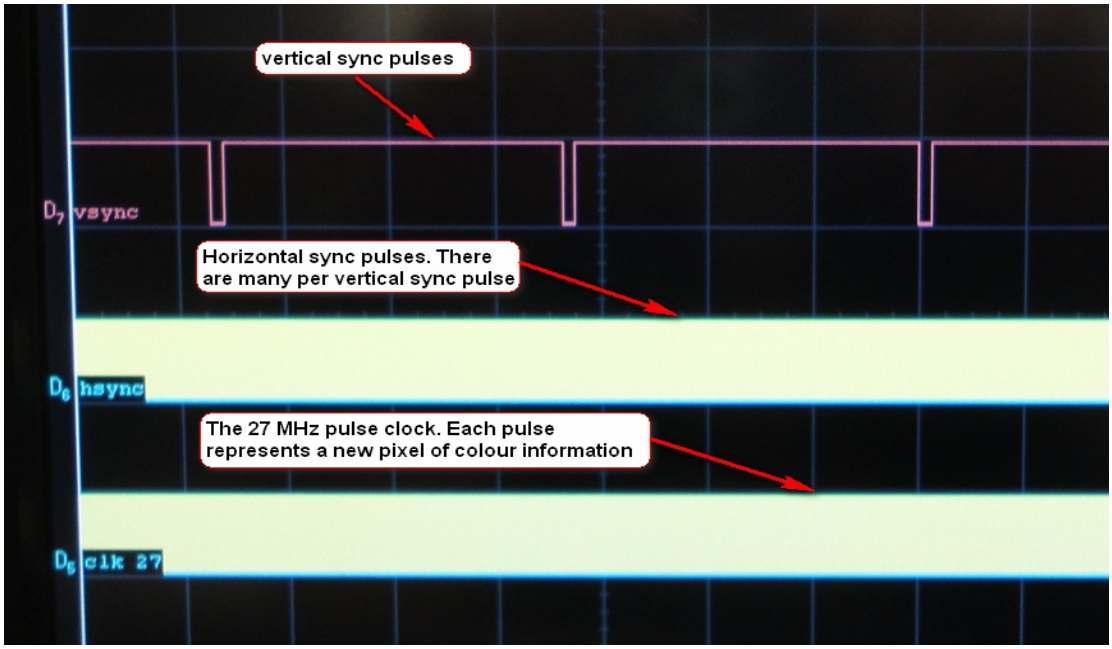


**Figure 3-video decoder output signals (vertical, horizontal and clock pulses)**

**Figure 4** shows a single vertical clock pulse several horizontal pulses and many single pixel clock pulses.

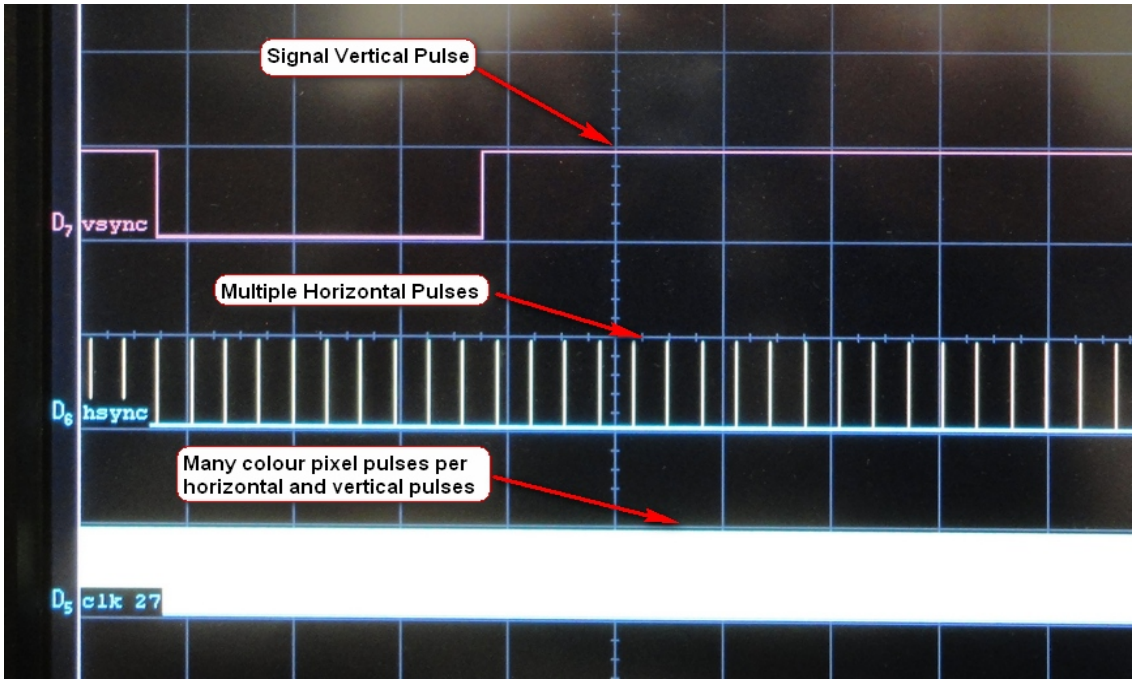


Figure 4-zoomed in view horizontal pulses

**Figure 5** shows the beginning of a video decoder frame

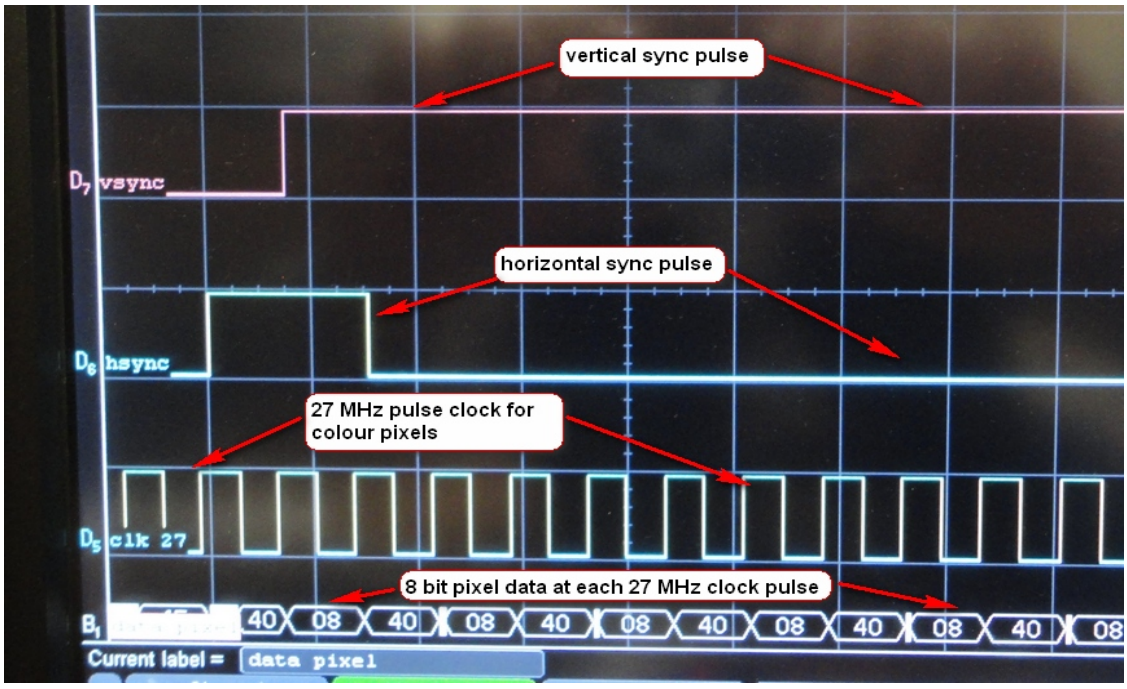


Figure 5-zoomed in view pixels pulses

The 8 bits of pixel data found at the bottom of **figure 5** represents the pixel colour value being sent by the video source, in this case it is a camcorder.

A frame equates to one picture frame of video. **Figure 6** shows a pictorial view of a frame, which is "X" number video bits per horizontal row and "Y" number of vertical rows.

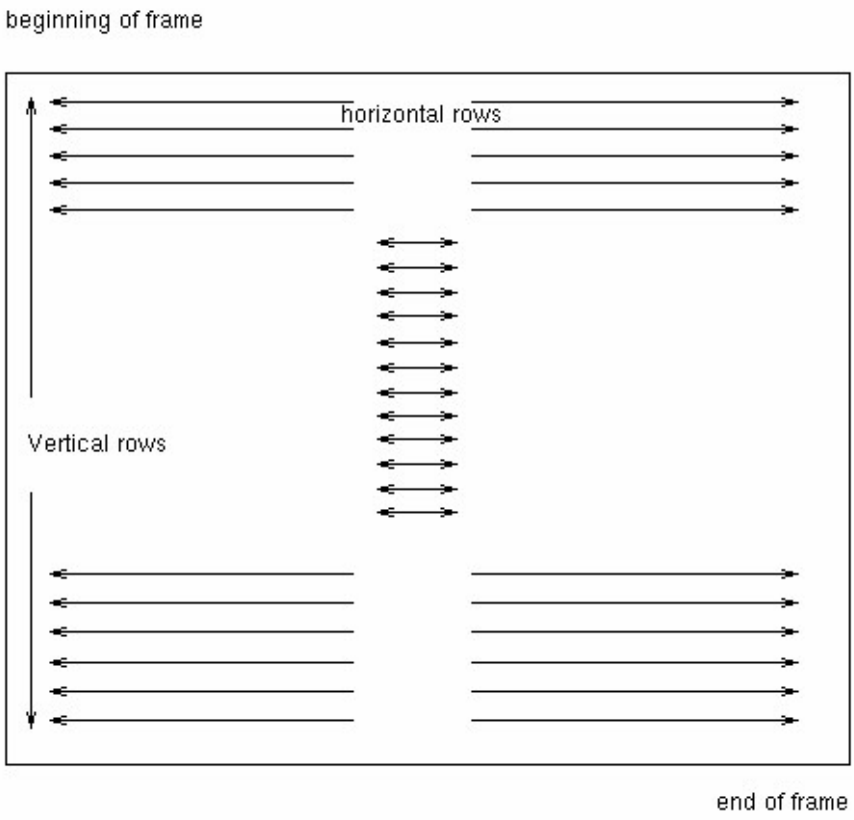


**Figure 6- block view of a video frame**

As an example say each horizontal row has **640** pixels and there are **480** vertical rows. This means the frame size is **640 X 480 = 307200**. This is important to know because it tells you how much memory is needed to save one frame of video data. So in this case we would need ~ 307k x8 or 307K bytes. The 8 represents the 8 bits of colour data per pixel.

## NTSC- National Television System committee

The format in which video is received by the ADV7181 decoder is NTSC. This standard was originally established by the FCC (Federal communications council) in 1940. The format has gone through many changes over the years. For more information go to the following link:

http://en.wikipedia.org/wiki/NTSC

It is slowing being phased out in place of the more common Digital Video format. However the DE2 still uses the NTSC format. The key features of the format are as follows:

- The clock frequency is 27MHz.
- The cycle frequency is 60 Hertz.
- Video is sent interlaced. This means two frame cycles are required to capture a full video display, where the first frame are all the odd horizontal lines and the second frame are all the even lines. This then get repeated. See **figure 7 below.**

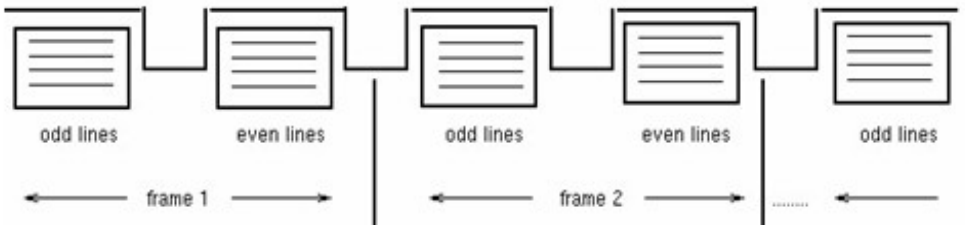


**Figure 7- interlacing frames**

## I2C- Inter-Integrated Circuit

This is a very common serial protocol that is used in industry to serially program internal registers. The DE2 board has two devices that use this protocol, the video decoder chip ADV7181 and the audio CODEC chip WM8731. The serial bus is shared by both chips but each chip has its own command select address. **Table 2** shows the command addresses and their function.

| Command address | chip | function |
|---|---|---|
| HEX 0X40 | Video chip ADV7181 | Write data to the internal video registers |
| HEX 0X41 | Video chip ADV7181 | Read data from the internal video registers |
| HEX 0X34 | Audio chip WM8731 | Write data to the internal audio registers |
| HEX 0X35 | Audio chip WM8731 | Read data from the internal audio register |

**Table 2 address value for command read and write**

F**igure 8** shows how the bus is connected pictorially. The serial clocks and serial data lines are connected to the FPGA I/O pin at the location specified in **table 1**. The serial clock and serial data will have to be generated using Verilog. The FPGA is generating the clock and data signals, so they are called the master device. The video encoder and the audio CODEC are called the slave devices. This is because they are receiving the serial data and serial clock signals.

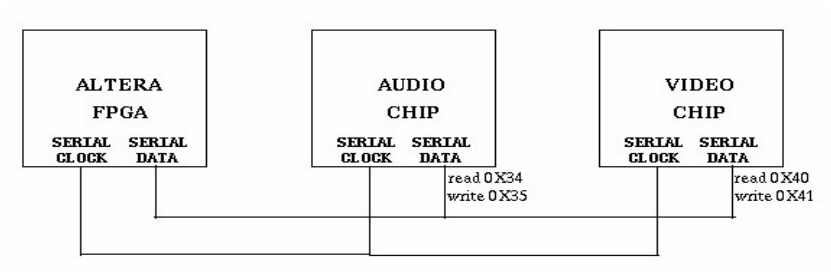


**Figure 8-interface of I2C devices**

The timing diagram for the serial data and serial clock lines must look like **figure 9**. The timing is very specific otherwise the values will not get loaded properly to the slave device. At the end of the transfer the slave device sends an Acknowledge pulse to indicate that it properly received the serial data.

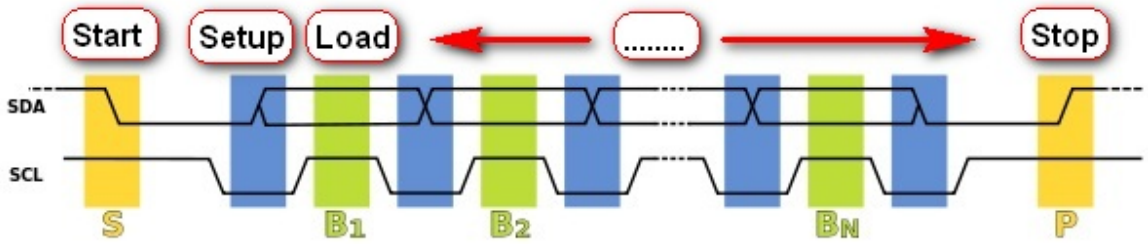


**Figure 9- timing for serial data and clock**

- **Start**- represents the beginning of serial load cycle. The data line goes from a high to a low transition and then the next cycle the clock line goes from a high to a low transition.
- **Setup**- represents the period that the data bit is set low or high
- **Load** – The serial clock goes from a low transition to a high transition. At that time the data value on the data bus is accepted by the internal data register of the slave device. This continues on until the last bit is loaded into the internal register.
- After all the data bits have been clocked into the slave device it sends an acknowledge pulse.
- **Stop**- This indicates the end of the load. The clock goes high first and the cycle after that the data line goes high. Once this is complete another load can occur using the same procedure just described above.

**Figure 10** shows an example of a single load to the video encoder (ADV7181) chip on the DE2 board.

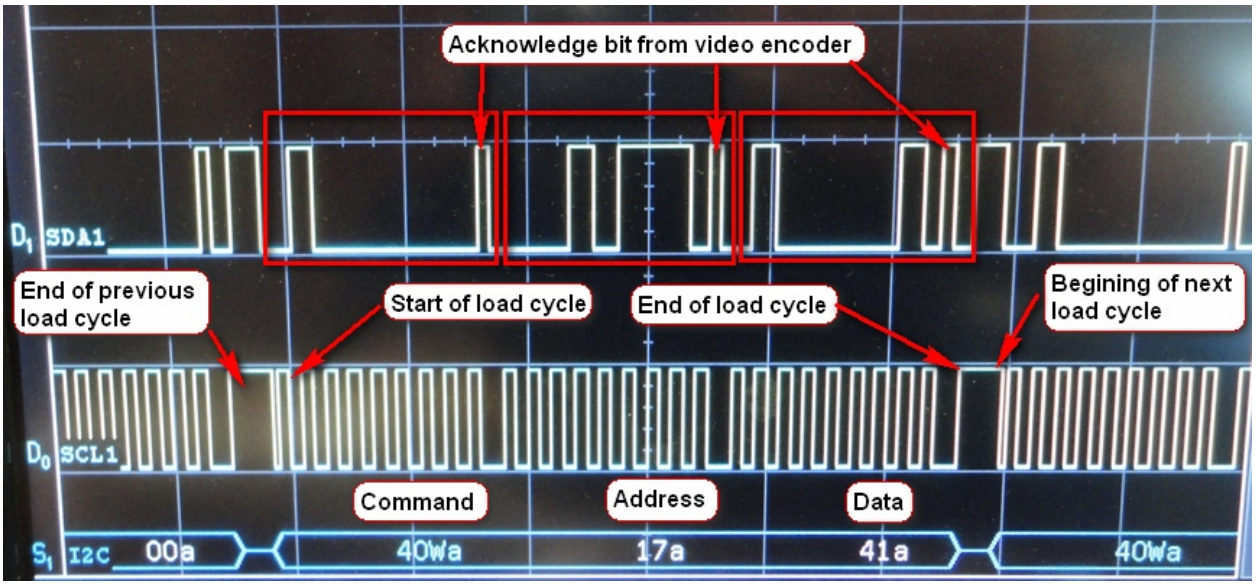


**Figure 10 – single serial load I2C protocol**

As you can see each load consists of a command, address, and data.

- **Command** serves two purposes. It acts as the identifier for the serial device that will be loaded. It also tells the device if it will read or write a value to or from the chip. In this case **40** indicates a **write** to the **video encoder** chip. ( see **table 2** for explanation of command values)
- **Address** serves as the location where the data will be loaded.
- **Data** is the value that will be written into that address location**.**
- **Acknowledge** is the bit that the slave device sends back to the master device to indicate all 8 bits were sent properly.

**Figure 11** shows multiple cycles. In this case we see that three write commands have been sent to the video encoder [address 15, data 00], [address 17, data 41], [address 3A].

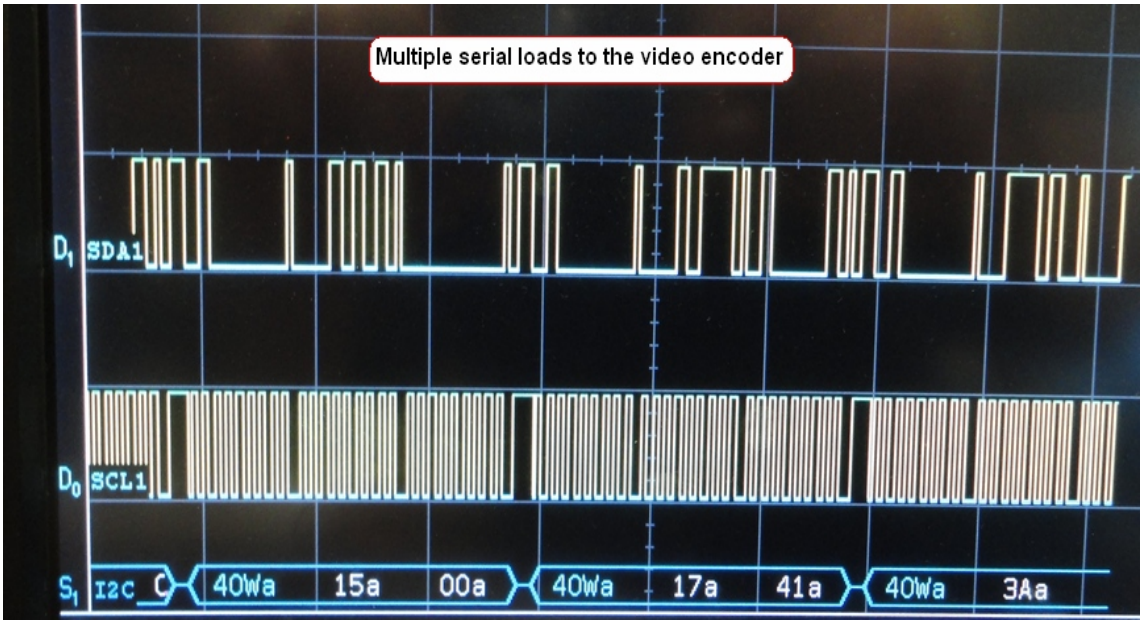


**Figure 11- multiple loads of I2C protocol**

## Tutorial 1 – Investigating and developing code for I2C on Video decoder.

The Verilog code will be written and compiled using Quartus and the result will be displayed using the Agilent MSO-X-3024A.
Please note that in order to do the following tutorial you will have to have knowledge of the following;

- Altera CAD package Quartus. We are using version 11.1 service pack 2
- Verilog programming language. If you are not familiar with Verilog go to the following link for tutorials and example;

http://www-ug.eecg.utoronto.ca/desl

**Select** NIOS II (DE2/DE1)>reference>Verilog

- Operation of the MSO-X-30024A. For reference on how to use this scope follow the link below, which has both an explanation and tutorial on how to use it;

http://www-ug.eecg.utoronto.ca/desl

**Select** Equipment>scopes>Instructions and Tutorials [under MSO-X-3024A]

**Table 3** identifies the signals on the video decoder ADV7181B that will be needed for this tutorial.

| | | |
|---|---|---|
| TD_CLK27 | Pin_D13 | TD_Decoder Clock Input |
| TD_Reset | Pin_C4 | TD_Decoder Reset |
| I2C_SCLK | Pin_A6 | I2C Clock |
| I2C_Data | Pin_B6 | I2C_Data |

**Table 3- pin assignments to generate 40 KHz clock**

## Part 1- The following tasks will need to be accomplished.

1) Enable the 27 MHz Clock. (**TD_CLK27**). In order for this to happen we need to assert a logic level high to the (TD_RESET) input pin. We can use one of the switches on the DE2 board for this purpose.
2) Write Verilog code, to divide the 27 MHz clock so we get a clock frequency that is compatible with the SCLK. Page 8[table2] of the ADV7181 data sheet (the link can be found on page 2 of the manual), says that the Max frequency for the SCLK is 400 KHz. For the purpose of this tutorial we will reduce the frequency to 40 KHz.
3) Finally display the 40 KHZ clock on the MSO-X-3024A scope for verification. This will be done by routing the 40 KHz clock signal to the output of one of the 40 pin GPIO headers.

   The Verilog code should look similar to the following link;

http://www-ug.eecg.utoronto.ca/desl

**Select**- DE2>DESL Online Tutorials>clock_generator.v

The pin assignments can be found at the same web location;

**Selec**t- DE2>DESL Online Tutorials>clock_generator.qsf

Create a new project using the **new project wizard** in Quartus. Once the Verilog code has been compiled and downloaded to the DE2 board, connect the digital leads of the MSO-X-3024A as in **figure 12.**

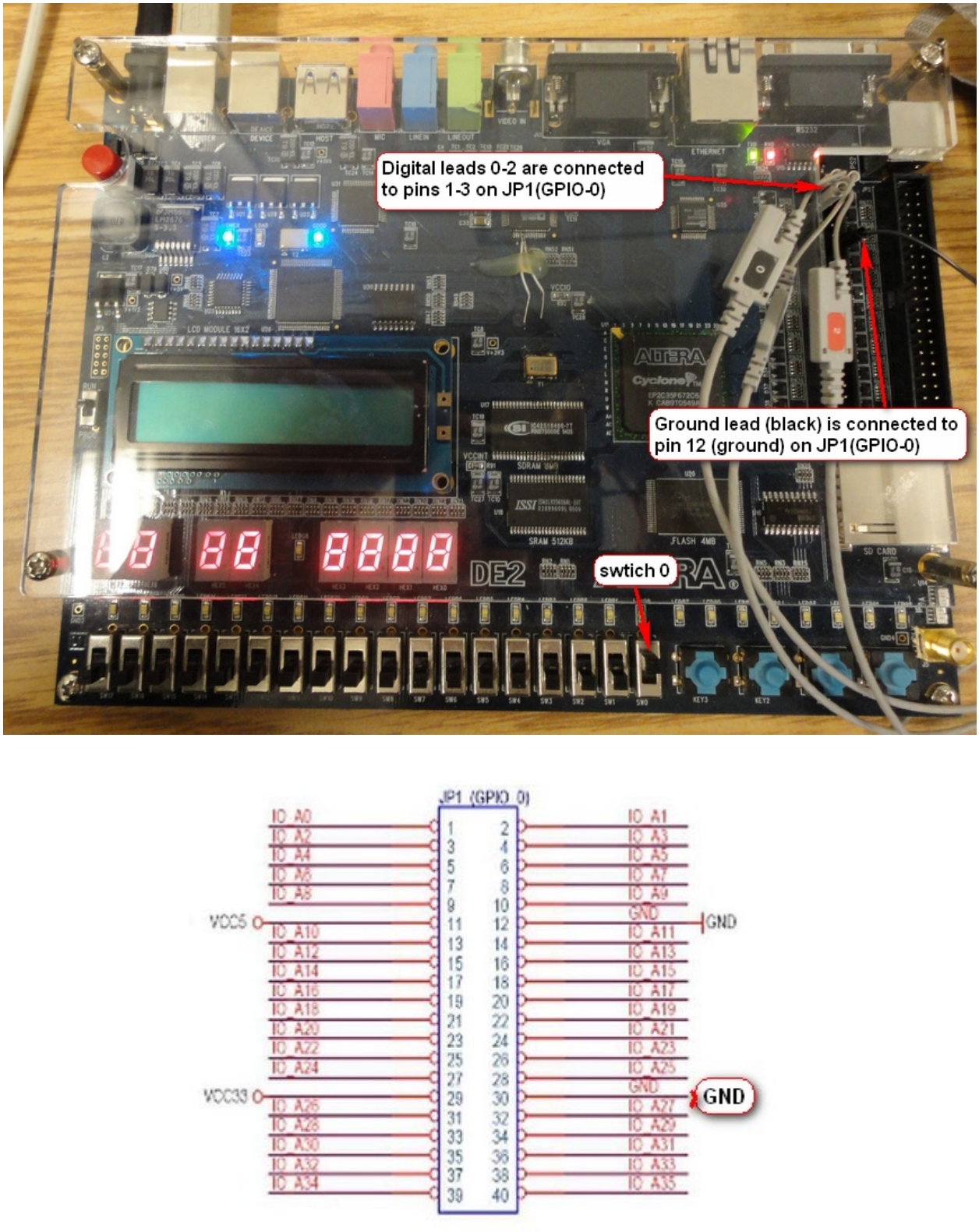


**Figure 12 - connecting digital leads to GPIO 0 on DE2 board.**

**Table 4** describes how the digital leads on the MSO-X-3024A should be connected to the 40 pin header JP1 (GPIO-0) on the DE2 board.

| Name of pin Verilog | Pin location JP1(GPIO-0) | description |
|---|---|---|
| Clock_en | GPIO-1 pin 0 | Enables 27 MHz clock |
| Clk_27 | GPIO-1 pin 1 | 27 MHz clock |
| SCLK | GPIO-1 pin 2 | 40 KHz clock |

Table 4- pin assignments on GPIO-1

Make sure to put switch 0 in the up position. It will enable the 27MHz clock which then will generate the resulting 40 KHz clock through the Verilog code.

- Use **Edge Trigger** mode and SCLK (D2) as the **trigger** source.
- Set **slope** to rising edge (low to high transition).
- Set **delay** is 0.0s
- Set **frequency** to **5.000u/s.**
- Press the **Single** button in the run control area of the MSO-X-3024A.
- Press **cursors**
- Using **X1** and *X2* measure the delta frequency of the SCLK signal. It should be 40.000 KHz.

The resulting trigger captured on the MSO-X-3024A should look like **figure 13**.

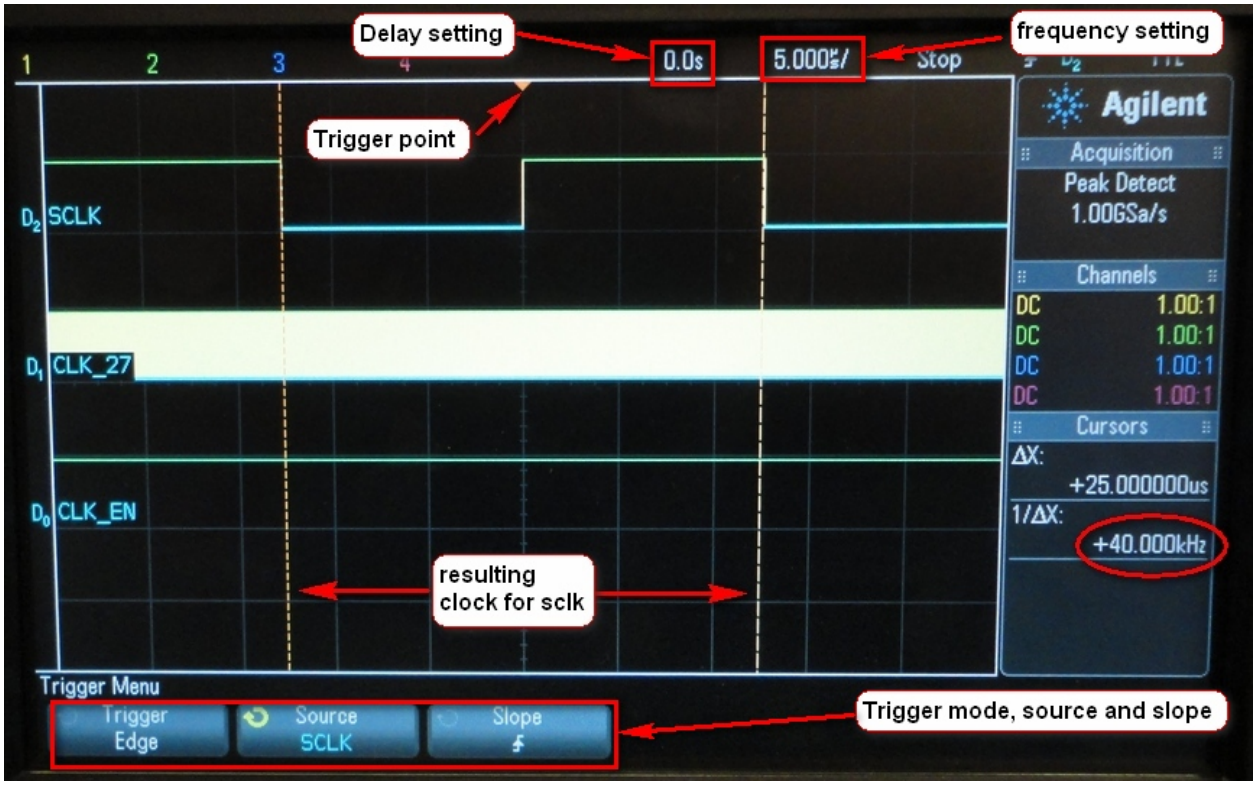


**Figure 13 - using cursors X1 & X2 to evaluate delta clock frequency**

Press the **zoom** button in the horizontal section of the MSO-3024 and set the **delay** to 200 ns you will get the result in **figure 14**. Note that the 40 KHz clock is just multiple divides of the 27 MHz clock.


**Figure 14 - magnified view of 40 KHz clock generation**

Now we have generated the SCLK signal and also used the MSO-3024 to verify that the frequency is correct. This concludes part 1.

## Part 2- Create registers to store the 8 bit command, 8 bit address and 8 bit data.

The following tasks will have to be done;

1. Create an 8 bit register to store HEX data values. There will be three registers one for command, one for address and one for data.
2. Using switches on the DE2 board to generate and select the HEX data values.
3. Using the HEX displays on the DE2 board to give a visual display of what data is stored in the registers created in step 1

Before writing the Verilog code we need to make a few observations;

1. The command value will never change. It will always be HEX 40 for the video decoder. See **table 2**.
2. We will use 4 switches to write a single HEX value (4 bits)
3. We will use 3 switches to select and enable the location where the HEX value will go. See **figure 15** and **table 5** for further explanation.

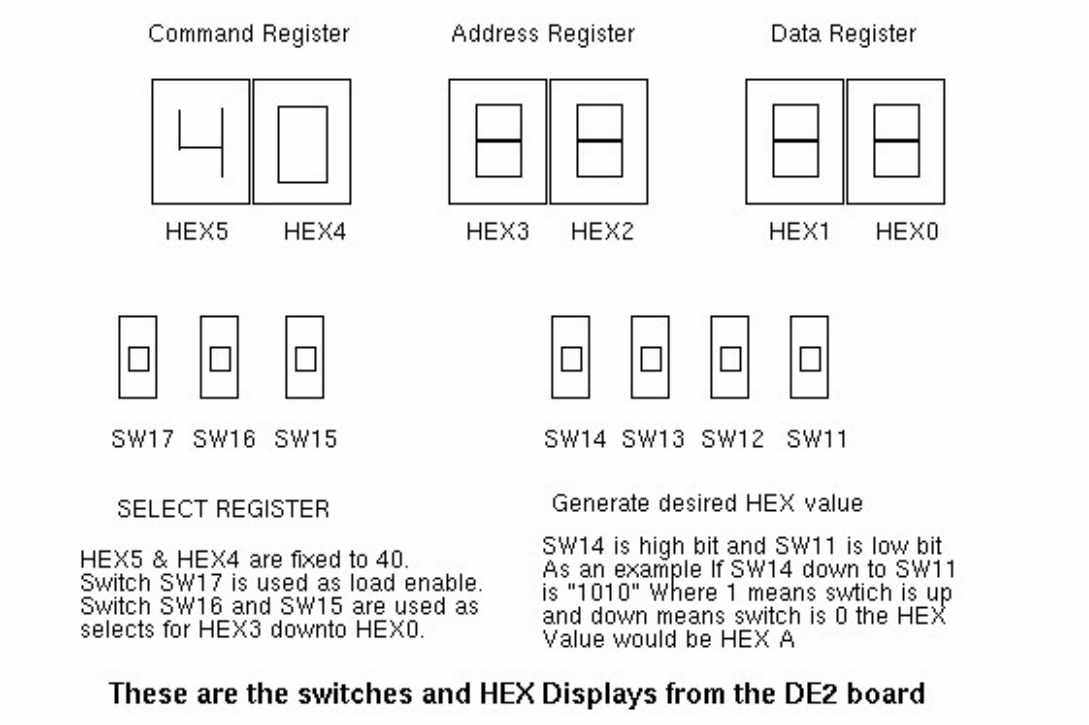


**Figure 15 –Truth table for HEX decoder**

| SW17 up equals enable down disabled | SW16 | SW15 | SW14 down to SW11 | HEX register enabled and displayed |
|---|---|---|---|---|
| Up | Down | Down | HEX value | HEX0 |
| Up | Down | up | HEX value | HEX1 |
| Up | Up | Down | HEX value | HEX2 |
| Up | Up | Up | HEX value | HEX3 |
| Down | Does not matter | Does not matter | N/A | none |

**Table 5 –Switch decoder settings for HEX displays and registers**

With this information we can now write the Verilog code. The Verilog code should look similar to the following link.

http://www-ug.eecg.utoronto.ca/desl

**Select**- DE2>DESL Online Tutorials>HEX_decoder.v

For pin assignments **Select**- DE2>DESL Online Tutorials >HEX_decoder.qsf

Create a new project using the **new project wizard** in Quartus. Compile the Verilog code and downloaded it to the DE2 board.

**Figure 16** shows an example of what the output looks like. Note that HEX7 and HEX6 are blank and HEX5 and HEX4 are fixed to 40. The rest HEX3 down to HEX0 can have their HEX values changed and stored according to the switch settings SW17 down to SW11. SW0 is used as a circuit enable.

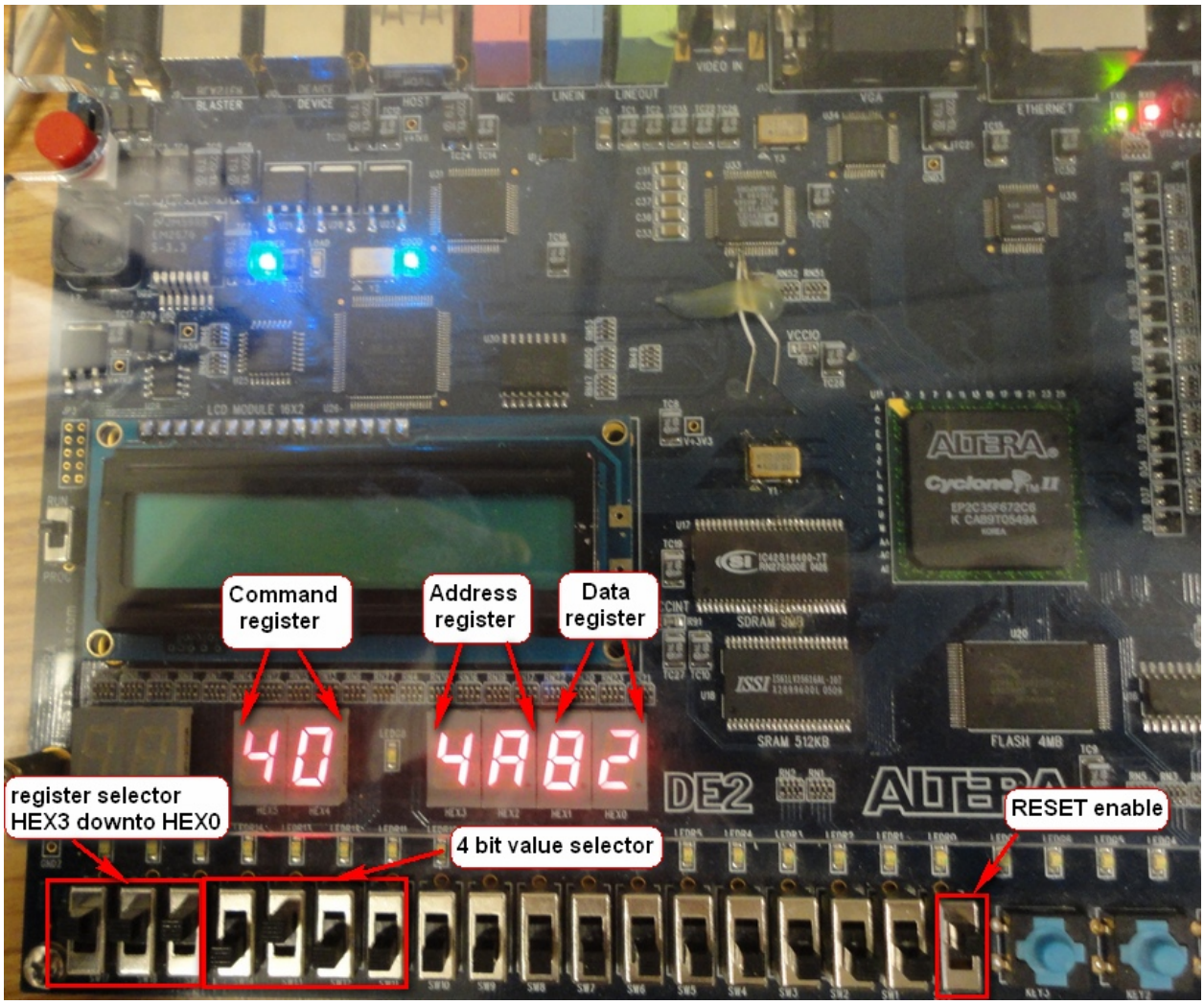


**Figure 16 – HEX data values displayed on DE2 board**

This concludes part 2.

## Part 3 - we will generate the SCLK and SDATA signals so we can serially shift clocked data into the video decoder.

### The key points for part 3 are as follows;

1. Generate a start pulse
2. Generate an 8 bit values for command, address and data.
3. Generate SCLK pulses.
4. Continue until data is sent
5. Wait for acknowledge from video decoder
6. Once command address and data have been sent generate stop pulse.

**Figure 17** give us a pictorial description of the points just mentioned.

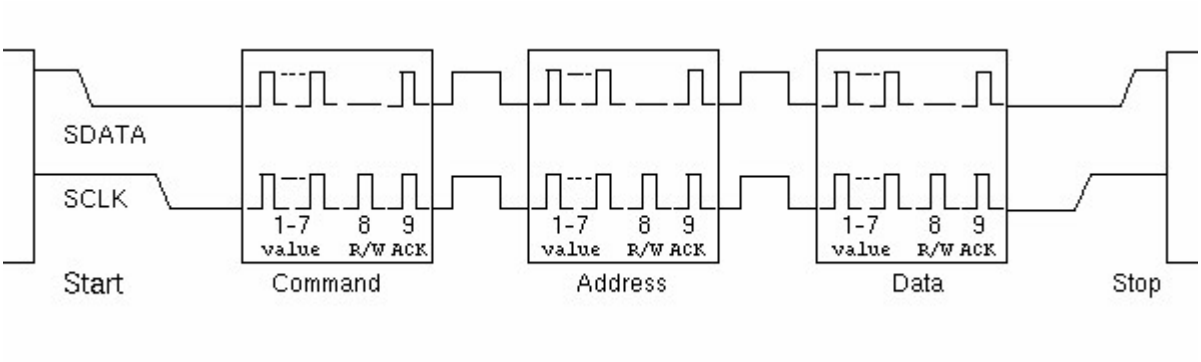


**Figure 17 – block diagram of serial I2C data transfer**

Now that we have all of the information we can write the Verilog code. The code should look similar to one found at the following link;

http://www-ug.eecg.utoronto.ca/desl

**Select**-DE2>DESL Online Tutorials>sdata_sclk.zip. Unzip the files in a new directory.

This contains three Verilog files (**HEX_decoder.v** , **clock_generator.v and sdata_sclk.v**) and the pin assignment file (**sdata.sclk.qsf** ).

Create a new project using the **new project wizard** in Quartus. Once the Verilog code has been compiled, downloaded it to the DE2 board,

Connect the digital probe leads from the MSO-3024 to the GPIO-0 40 pin header. Use column 3 and 4 in **table 6** and **figure 19** as reference. Connect ground pins (black lead) to pins 12 and 30 on 40 pin header as shown on **figure 19**.

| Name of GPIO-0 pin | Location on FPGA | Location on GPIO -0 40 pin header | Probe or Digital lead connection on MSO |
|---|---|---|---|
| GPIO[0] | D25 | 1 (IO A0) | Digital Lead 0 |
| GPIO[1] | J22 | 2 (IO A1) | Digital Lead 1 |
| GPIO[2] | E26 | 3 (IO A2) | Digital Lead 2 |
| GPIO[3] | E25 | 4 (IO A3) | Digital Lead 3 |
| GPIO[4] | F24 | 5 (IO A4) | Digital Lead 4 |
| GPIO[35] | L19 | 40 (IO A35) | Analog Probe 1 |

**Table 6- pin assignments for GPIO 0 Tutorial 1 part 3**

We will use the MS0-X-3024 to check that the serial data (command, address and data) are being transferred correctly.

## Connect analog probe from MSO-X-3024A to DE2 GPIO-0 40 pin header.

Get 2 **probe pin leads** found in a plastic bag inside the pouch at the top of the MS0-X-3024A.

Connect one to the ground alligator clip and the other to the analog probe. See **figure 18**;

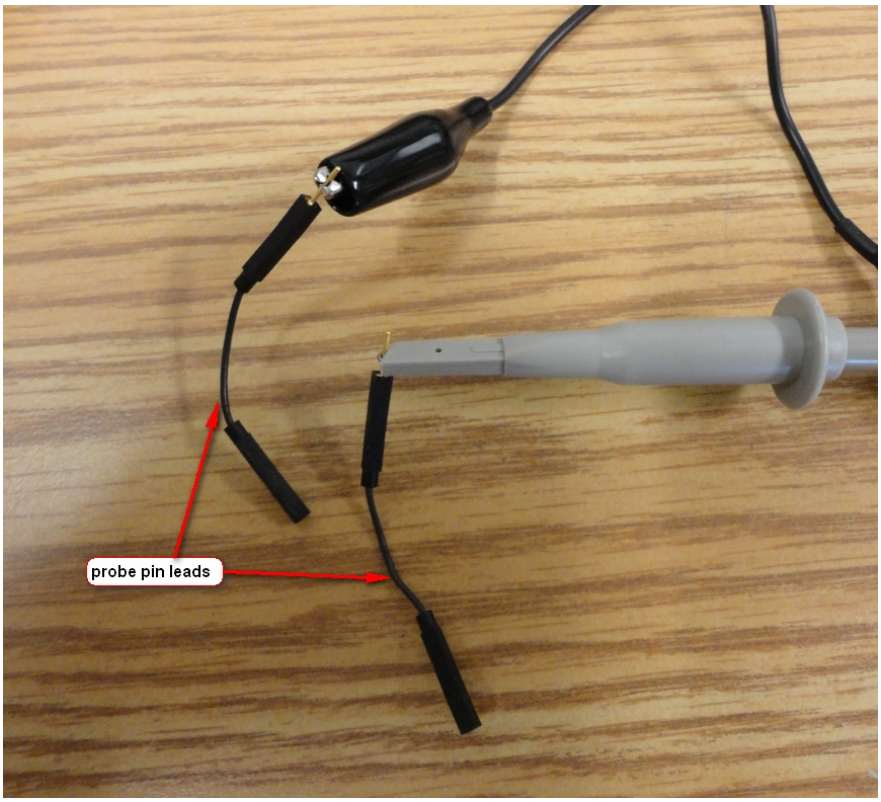


**Figure 18 – description of how to connect probe pin leads to analog probe**

Now connect analog probe as described in **Table 6** at the bottom. Also use **figure 19** as a further reference**.**

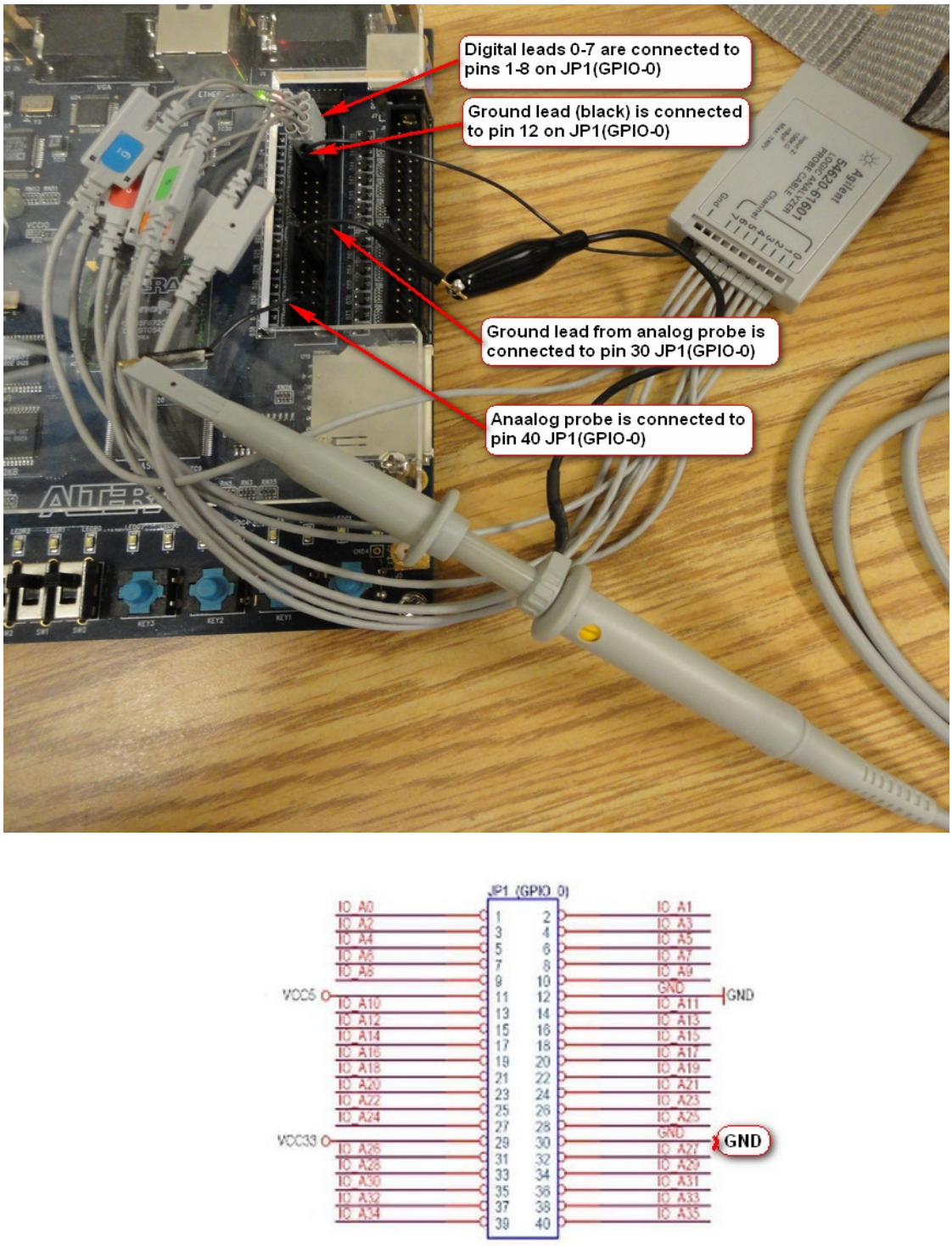


**Figure 19 – digital and analog connections to GPIO 0 header**

- **Press label** and Label the **D0, D1** and **D4** digital leads as in **table 7** below

| Digital lead default name | Rename label |
|---|---|
| D0 | GO |
| D1 | CLK_27 |
| D2 | SCLK |
| D3 | SDA |
| D4 | CLOCK |

**Table 7- Renamed digital leads.**

- Press **Trigger**
- Set trigger type to **serial 1(I2C)**
- Press **Serial**
- Press **Signals**
- Change **SCL** to **D2**
- Change **SDA** to **D3**
- Press **back**
- Press **Addr Size**
- Change to **8 bit**
- Press **Trigger**
- Change **Trigger on:** to **Start Condition**.
- Press **Digital**
- Set **scale** size to Maximum.
- Set **delay** from trigger point to **1.500** m/s.
- Set **frequency** to **500.0** u/s**.**
- Make sure **switch 1** is in the down position and **switch 0** is in the up position.
- **Switch 0** is the clock enable for the 27 MHz clock.
- **Switch 1** is **GO (D0)**.
- Press **Single** in the run section of the MSO-X-3024A.
- Toggle **switch 1** from down to up position.

The result should look like **figure 20**.

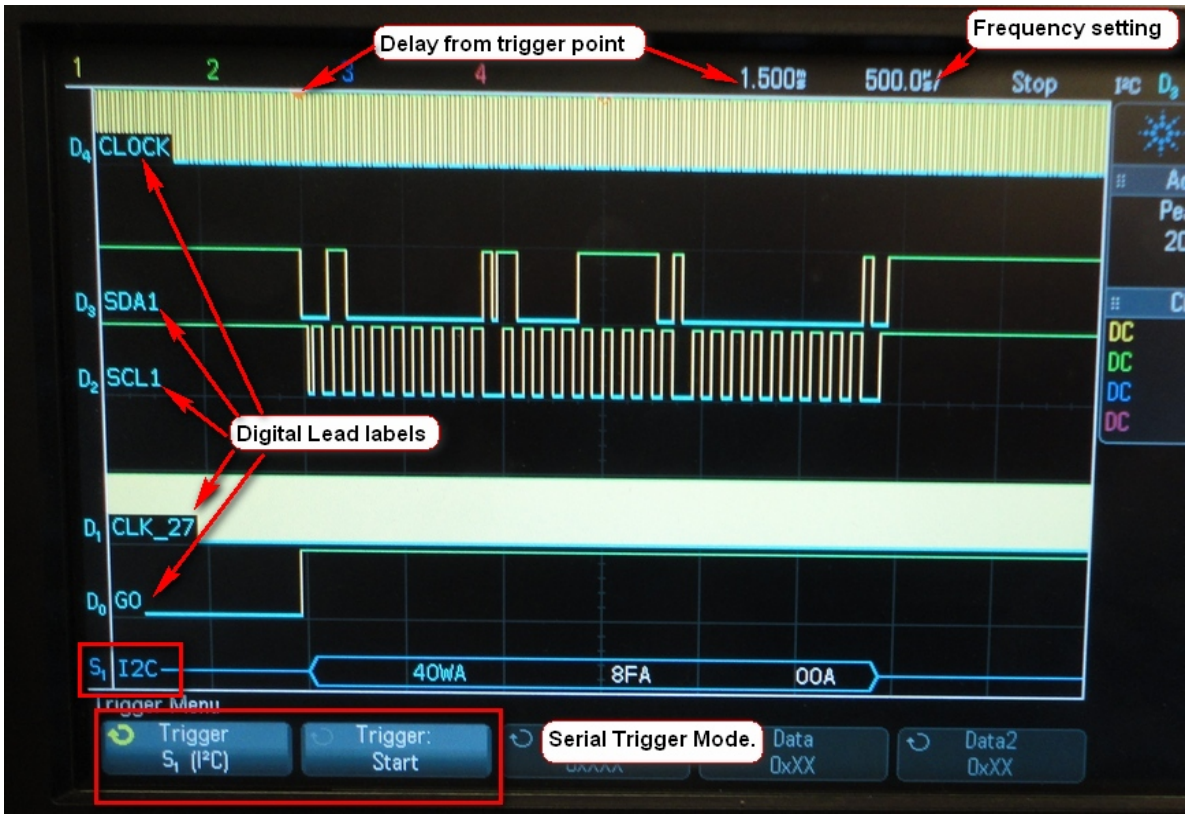


**Figure 20 – label location delay trigger and frequency setting**

*****Note******-The next part of this tutorial can only be done with DE2 boards that have serial numbers starting with **0**. Otherwise go to the next section; **Capturing video from a composite camcorder and saving it to memory** page 26**.**

From part 2 (HEX decoder) Switches 17 down to 11 are used to select and set the value to be transferred. Set the value to be serial shifted as shown in **table 8**.

| Register names | HEX Value | Description |
|---|---|---|
| Command | 40 | This is the **write** value as described on Page 62 ADV7181 data sheet |
| Address | 8F | Address location that we want to change data value |
| Data | 00 | HEX data value that we want to write to the address |

**Table 8 Value to be transferred (LLC frequency control)**

**Page 86** of the **ADV7181** data sheet gives a description of the address value and the different data option available. This address location HEX **8F** controls the frequency of the LLC1 pin (This is the 27 MHz clock pin). By changing the values of bits 6-4 we can change the frequency of the LLC between 13.5 MHz to 27. 0 MHz and vise versa. If the data value is HEX 05 then the LLC output pin will be 13.5 MHz. If the value is HEX 00 then the output is 27 MHZ. This happens to be the default value at reset.

- Make sure **switch 1** is up and **switch 2** is down.
- Press **digital** button and deselect digital channel **D0** and **D1**.

- Move **D2** (SCL1) , **D3** (SDA1) and **D4** (CLOCK) down on the scope. See **figure 21**.
- Connect analog probe to 1 input on the MSO-3024.
- Select analog probe **1** (Yellow channel) and move it up to the top of the scope. See **figure 21**.
- Press Label and rename analog probe 1 **analog_27.**
- Set the voltage level for analog channel 1 to **5.00** V.
- Set **delay** to **1.630** m/s**.**
- Set **frequency** to **500.0** u/s.
- Press **Single** on run section of MSO-X-3024A.
- Toggle **switch 2** (GO) from a down to up position.

The result should look like **figure 21**. **Analog_27** represents the 27 MHz clock. By adjusting the frequency and using the cursors you can verify if this is true or not. This is optional.

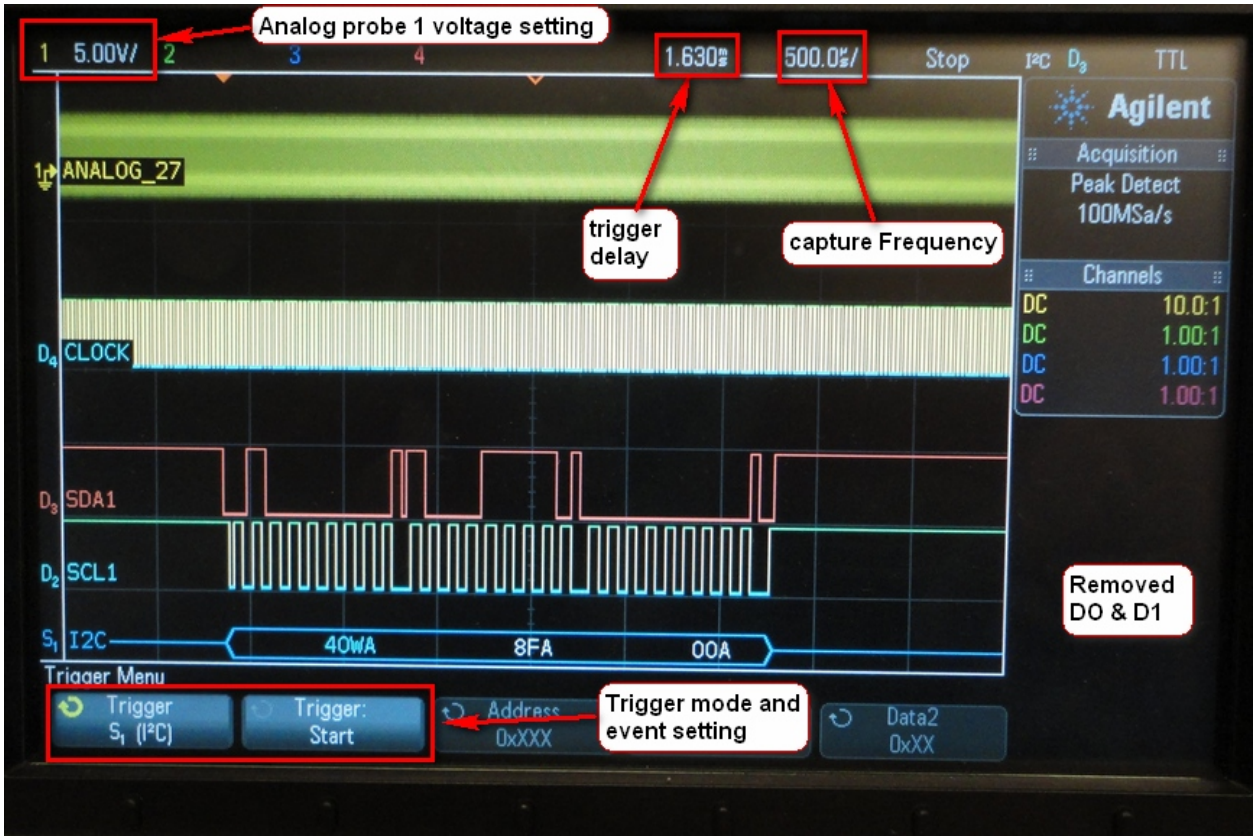


**Figure 21 – add analog probe 1 input and remove D0 and D1**

Change the data value from HEX 00 to HEX 50 using switch 17 down to 11. With this data value change the LLC frequency will change from 27 MHz to 13.5 MHz. To verify this we will use the MSO-X-3024A to not only see this but verify it as well.

- Change the **delay** to **3.320** m/s.
- Change the **frequency** to **500** u/s.
- Make sure **switch 2** is in the down position.
- Press the **single** button in the run section.
- Toggle the **GO** (switch2) from a low to high position.

This should trigger a new event. The result should look like **figure 22**.

Note that at the point where the data value HEX 50 happens the frequency changes.

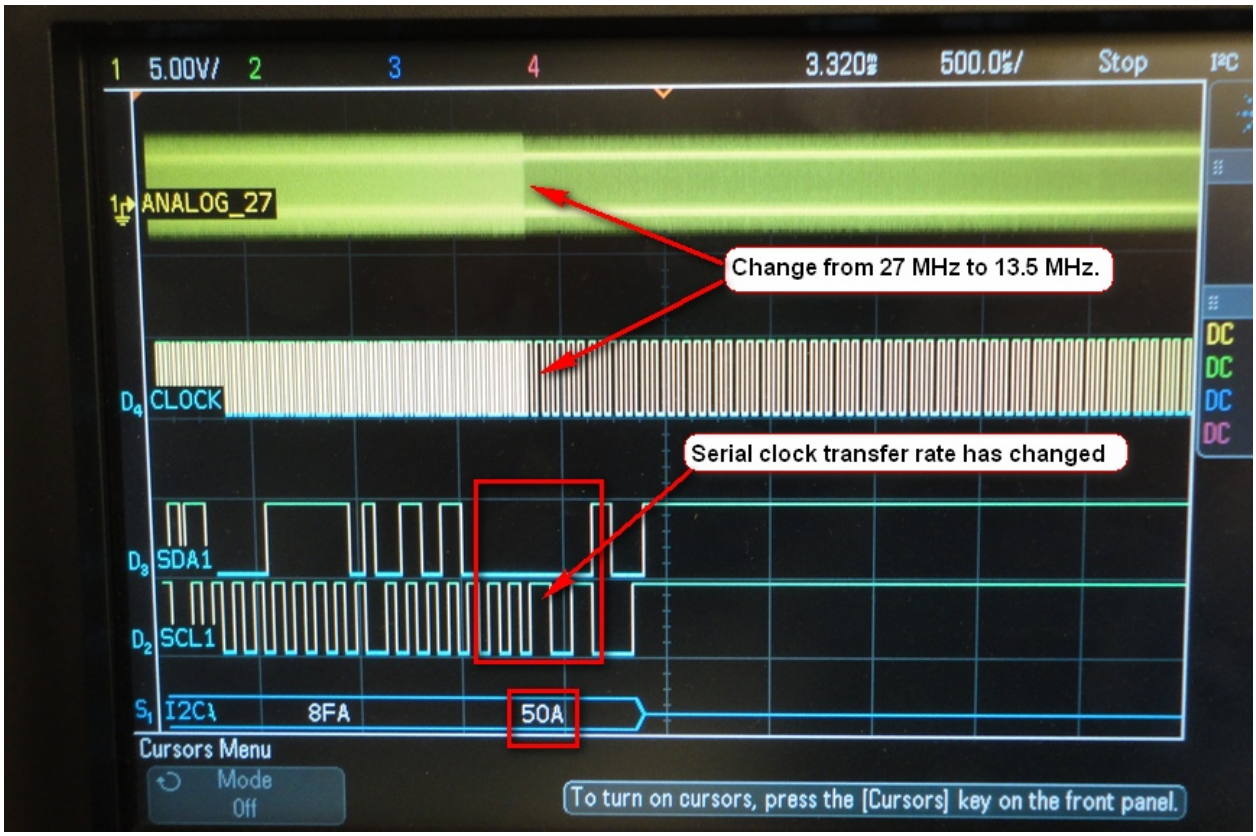


**Figure 22 – clk_27 frequency changed from 27 MHz to 13.5 MHx**

Also note that the serial transfer rate changed too. See **Figure 22**. The SCL1 and SDA1 signals are slower because the CLOCK frequency has been divided in half.

Now we can verify that the clock rate has changed by using the cursors (**X1** and **X2**) and measuring the delta value. Before making the measurement;

- Change the delay value to **5.310** m/s
- Change horizontal frequency to 50.00 u/s
- Make sure **switch 2** is in the down position.
- Press the **single** button
- Toggling **Go (switch2)** from a low to high position.

- Press **cursors**.
- Using X1 and **X2** measure the delta time of **D4 CLOCK**.

The result should be that the (1/delta X) value is **20.000 KHZ**. See **figure 23**.


**Figure 23 – using cursors to check delta frequency**

Previously it was 40 KHz.

Now change the data value from HEX 50 to HEX 00 using switch 17 down to 11.

- Make sure **switch 2** is in the down position.
- Change **horizontal** frequency 500 u/s
- Change **Delay** to 3.320 m/s.
- Press the **single** button.
- Toggling **Go (switch2)** from a low to high position.

The result should be as in **figure 24**


**Figure 24 – change back to default frequency 13.5 MHz to 27 MHz**

- Change **Delay** to **2.628** m/s**.**
- Change **Frequency** to **20 .0** u/s.
- Make sure **switch 2** is in the down position.
- Press the **single** button.
- Toggling **Go (switch2)** from a low to high position.
- Press **cursors**.
- Using X1 and **X2** measure the delta time of **D4 CLOCK**.

The result should be that the (1/delta X value) is **40.000 KHZ**. See **figure 25**.

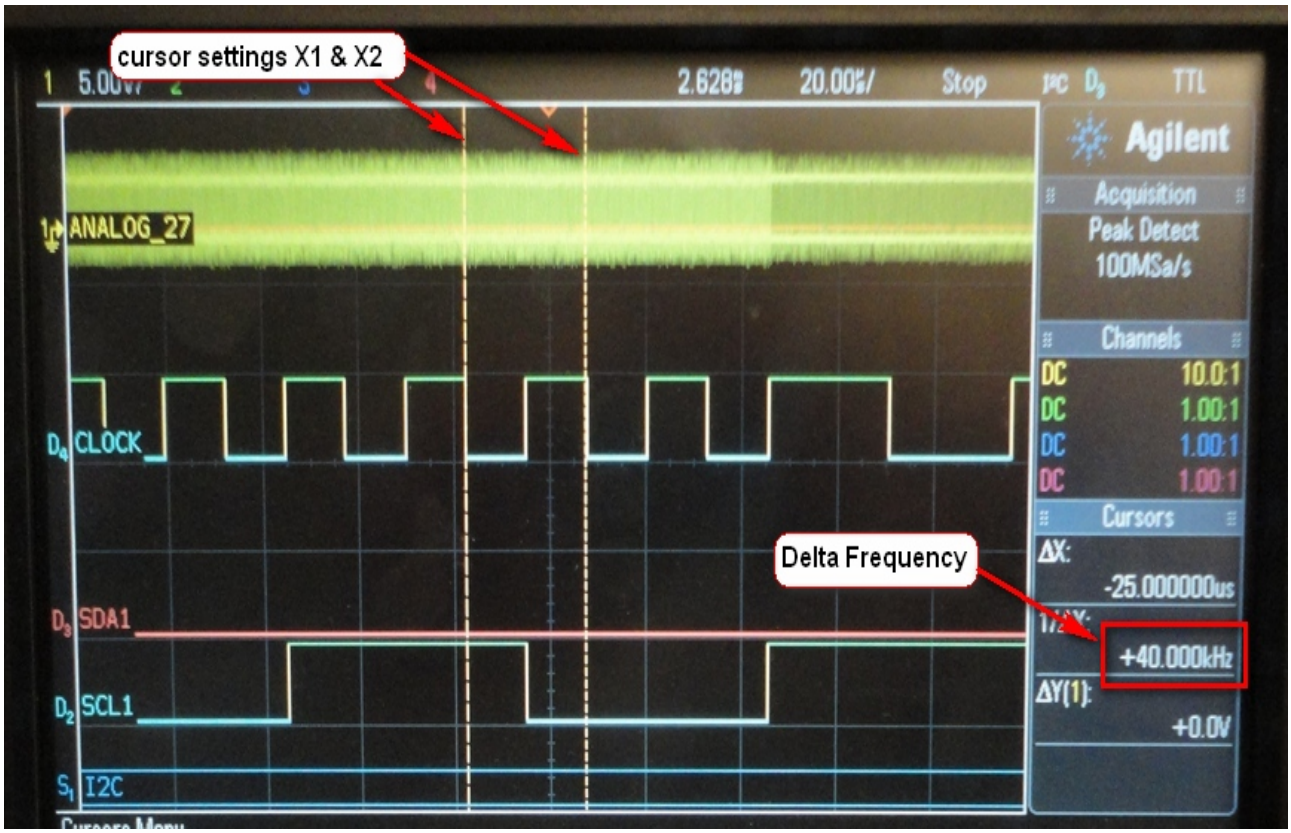


**Figure 25 – checking delta Frequency for default CLK_27 output**

This concludes tutorial 1. You should now be familiar with how I2C serial protocol works and also how to verify if the transfer properly occurred with the aid of a logic analyser.

## Capturing video from a composite camcorder and saving it to memory

Earlier we described the manner in which video is transmitted. Using that principal our next step is to write Verilog code to capture a video frame and store the data on the DE2 board. The DE2 board has several types of memory. For our purpose we will use the 512Kbyte SRAM. See **figure 26** for a block diagram and **table 9** shows the FPGA pin assignments for the SRAM chip.

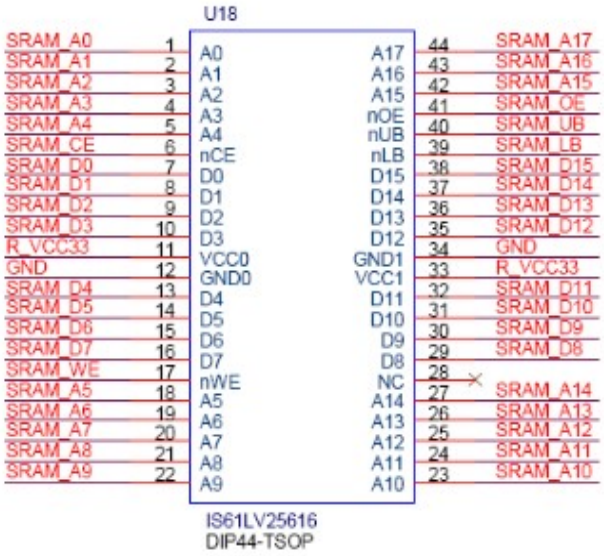


**Figure 26 – SRAM on DE2 board 512k X 8 Mbyte = 4 Mbits**

| Signal Name | FPGA Pin No. | Description |
|---|---|---|
| SRAM_ADDR[0] | PIN_AE4 | SRAM Address[0] |
| SRAM_ADDR[1] | PIN_AF4 | SRAM Address[1] |
| SRAM_ADDR[2] | PIN_AC5 | SRAM Address[2] |
| SRAM_ADDR[3] | PIN_AC6 | SRAM Address[3] |
| SRAM_ADDR[4] | PIN_AD4 | SRAM Address[4] |
| SRAM_ADDR[5] | PIN_AD5 | SRAM Address[5] |
| SRAM_ADDR[6] | PIN_AE5 | SRAM Address[6] |
| SRAM_ADDR[7] | PIN_AF5 | SRAM Address[7] |
| SRAM_ADDR[8] | PIN_AD6 | SRAM Address[8] |
| SRAM_ADDR[9] | PIN_AD7 | SRAM Address[9] |
| SRAM_ADDR[10] | PIN_V10 | SRAM Address[10] |
| SRAM_ADDR[11] | PIN_V9 | SRAM Address[11] |
| SRAM_ADDR[12] | PIN_AC7 | SRAM Address[12] |
| SRAM_ADDR[13] | PIN_W8 | SRAM Address[13] |
| SRAM_ADDR[14] | PIN_W10 | SRAM Address[14] |
| SRAM_ADDR[15] | PIN_Y10 | SRAM Address[15] |

| | | |
|---|---|---|
| SRAM_ADDR[16] | PIN_AB8 | SRAM Address[16] |
| SRAM_ADDR[17] | PIN_AC8 | SRAM Address[17] |
| SRAM_DQ[0] | PIN_AD8 | SRAM Data[0] |
| SRAM_DQ[1] | PIN_AE6 | SRAM Data[1] |
| SRAM_DQ[2] | PIN_AF6 | SRAM Data[2] |
| SRAM_DQ[3] | PIN_AA9 | SRAM Data[3] |
| SRAM_DQ[4] | PIN_AA10 | SRAM Data[4] |
| SRAM_DQ[5] | PIN_AB10 | SRAM Data[5] |
| SRAM_DQ[6] | PIN_AA11 | SRAM Data[6] |
| SRAM_DQ[7] | PIN_Y11 | SRAM Data[7] |
| SRAM_DQ[8] | PIN_AE7 | SRAM Data[8] |
| SRAM_DQ[9] | PIN_AF7 | SRAM Data[9] |
| SRAM_DQ[10] | PIN_AE8 | SRAM Data[10] |
| SRAM_DQ[11] | PIN_AF8 | SRAM Data[11] |
| SRAM_DQ[12] | PIN_W11 | SRAM Data[12] |
| SRAM_DQ[13] | PIN_W12 | SRAM Data[13] |
| SRAM_DQ[14] | PIN_AC9 | SRAM Data[14] |
| SRAM_DQ[15] | PIN_AC10 | SRAM Data[15] |
| SRAM_WE_N | PIN_AE10 | SRAM Write Enable |
| SRAM_OE_N | PIN_AD10 | SRAM Output Enable |
| SRAM_UB_N | PIN_AF9 | SRAM High-byte Data Mask |
| SRAM_LB_N | PIN_AE9 | SRAM Low-byte Data Mask |
| SRAM_CE_N | PIN_AC11 | SRAM Chip Enable |

**Table 9 – pin out of SRAM on DE2 board**

If we go back to the definition of a frame, it is "X" number of colour data pixels per row time "Y" number of rows [XxY = frame]. There are two possible ways that we can store the pixel data in memory.

**Method 1**- is known as substitution. Here as you write data from memory to the monitor and then update that location with a new value from the video source. The advantage to this method is you only need enough memory to cover 1 frame. This method is very timing critical.

**Method 2**- is known as split memory. Here the memory is split into two equal halves. One half is used to update a pixel frame from the video source while the other half is used to display video pixel data to the monitor. At the end of the frame update the roles are reversed. See **figure 27** for further explanation. The advantage of this method, it is less time restrictive but you need at least two times the frame memory size.

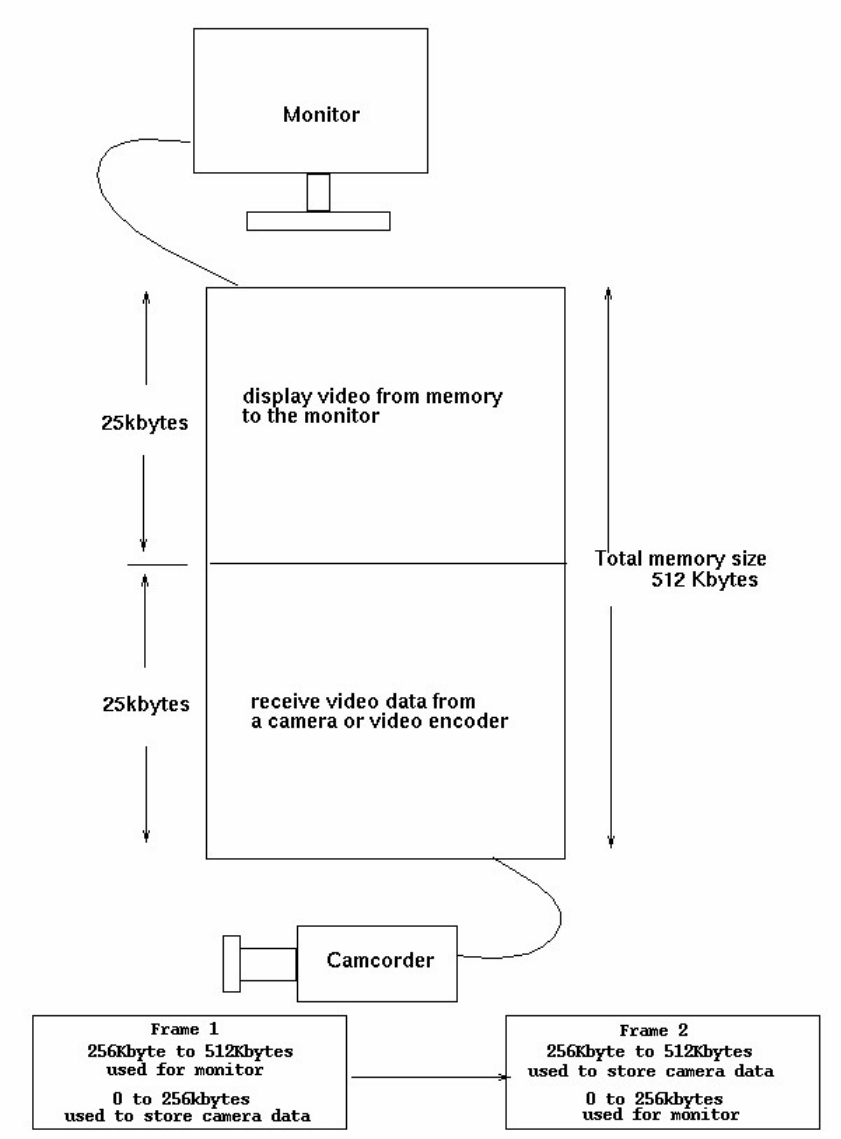


**Figure 27 – block diagram of memory connected to camcorder and monitor**

## Tutorial 2-Capturing video data from camcorder.

For the purpose of this tutorial we will use method 2 and split the memory into two equal halves. Each transfer will store 8 bits of pixel data. The colour format will be 4:2:2 RGB (4 Red, 2 Green and 2 Blue). This is the default mode for the video decoder (ADV7181).

The video decoder chip (ADV7181) generates 4 signals;

1. Video data (8 bits)
2. Clock ( 27 MHz enabled if RESET is active high)
3. Hsync- video in horizontal sync

4. Vsync- video in vertical sync

**Figure 28** shows an example of what the signals look like.


**Figure 28 Video in signals from camera source**

Our objective is to use these signals to capture a video frame and store them in memory. We will make the frame size **624X420**. Each pixel is **8** bits. If we recall from earlier video data from a camcorder is sent interlaced, odd video line in one frame and even video line in the second frame. So we require two frames to make a full frame to be displayed to a monitor. 2X ( 624X210X8).

Now we can make the following observations.

1. With each 27 MHz clock pulse an 8 bit video pixel will be stored into SRAM memory.
2. Hsync will represent each line of video data. So each line is 624 X8 that means we need 4992 memory location per horizontal line.
3. Vsync represents the number of lines (rows) for each frame. From above there will be 210 rows per frame 4992X210 = 1,048,320 (~1 Mbit)
4. So to do one full frame requires 2 Mbits. This is within the memory capacity of the SRAM memory chip on the DE2 board which is 4 Mbits.

Now that we have this information we can use Verilog to create the address and data counter to capture video pixels from a camcorder and store it in the DE2 onboard SRAM memory. The following counters, latches and enables will need to be created;

- Address counter for the odd and even frames

- Address enables for **ud**, **ld, oe**, **we** and **cs** signals on the SRAM memory chip
- Video line counter for each line (horizontal)
- Video frame counter to keep track of each row (vertical)
- Data latches for pixel data both for upper data strobe **ud** and lower data strobe **ld**.

A working example of what the verilog code should look like can be found at the following link;

http://www-ug.eecg.utoronto.ca/desl

If you have an older version of the DE2 board where the 27 Mhz clock is connected to D13 use the following:

**Select -**DESL Online Tutorials>Tutorial2_Clk_D13.zip.

Otherwise for the newer versions of the DE2 board where the 27 Mhz clock is connected to C16 use the following;

**Select**-DE2>DESL Online Tutorials>Tutorial2_Clk_C16.zip.

Create a directory and unzip the file. Using Quartus **new project wizard** create a new project called **camera**. Compile the project. Open the Verilog file called **camera**. Scroll to the bottom of the file and you should see the follow code. See **figure 29.**

```
271   ////////////////////////////////////////
272   /// logic analyser test bit locations ///
273   /// on GPIO JP0 40 in header          ///
274   ////////////////////////////////////////
275
276      always gpio[0] <= clk_27;
277      always gpio[1] <= we;
278      always gpio[2] <= ld;
279      always gpio[3] <= ud;
280      always gpio[4] <= vert;
281      always gpio[5] <= horiz;
282      always gpio[6] <= frame;
283      always gpio[7] <= vid_hs;
284
285      always gpio[15:8] <= address[7:0];
```

**Figure 29 test pins connected to GPIO 0 (JP0)**

As in the previous tutorial we will be using the GPIO-0 40 pin header to look at signals from the FPGA and display them on the Agilent MSO-3024 analyser.

Open up the assignment editor. If you scroll down you will see the follow pin assignments as seen in **figure 30**.

| | | | | | |
|---|---|---|---|---|---|
| 36 | ✓ | | gpio[15] | Location | PIN_G25 |
| 37 | ✓ | | gpio[14] | Location | PIN_K22 |
| 38 | ✓ | | gpio[13] | Location | PIN_G24 |
| 39 | ✓ | | gpio[12] | Location | PIN_G23 |
| 40 | ✓ | | gpio[11] | Location | PIN_P18 |
| 41 | ✓ | | gpio[10] | Location | PIN_N18 |
| 42 | ✓ | | gpio[9] | Location | PIN_F26 |
| 43 | ✓ | | gpio[8] | Location | PIN_F25 |
| 44 | ✓ | | gpio[7] | Location | PIN_J20 |
| 45 | ✓ | | gpio[6] | Location | PIN_J21 |
| 46 | ✓ | | gpio[5] | Location | PIN_F23 |
| 47 | ✓ | | gpio[4] | Location | PIN_F24 |
| 48 | ✓ | | gpio[3] | Location | PIN_E25 |
| 49 | ✓ | | gpio[2] | Location | PIN_E26 |
| 50 | ✓ | | gpio[1] | Location | PIN_J22 |
| 51 | ✓ | | gpio[0] | Location | PIN_D25 |

**Figure 30 Pin assignments GPIO Port JP0**

If we use the information from **figure 29** and **figure 30** we get the result in **table 10**

| Name of GPIO 0 pin | Location on FPGA | Name Assigned to the location from Camera.v project | Probe or Digital lead connection on MSO |
|---|---|---|---|
| GPIO[0] | D25 | CLK_27 | Digital Lead 0 |
| GPIO[1] | J22 | WE | Digital Lead 1 |
| GPIO[2] | E26 | LD | Digital Lead 2 |
| GPIO[3] | E25 | UD | Digital Lead 3 |
| GPIO[4] | F24 | VERT | Digital Lead 4 |
| GPIO[5] | F23 | HORIZ | Digital Lead 5 |
| GPIO[6] | J21 | FRAME | Digital Lead 6 |
| GPIO[7] | J20 | VID_HS | Digital Lead 7 |
| GPIO[8] | F25 | Address[0] | Digital Lead 8 |
| GPIO[9] | F26 | Address[1] | Digital Lead 9 |
| GPIO[10] | N18 | Address[2] | Digital Lead 10 |
| GPIO[11] | P18 | Address[3] | Digital Lead 11 |
| GPIO[12] | G23 | Address[4] | Digital Lead 12 |
| GPIO[13] | G24 | Address[5] | Digital Lead 13 |
| GPIO[14] | K22 | Address[6] | Digital Lead 14 |
| GPIO[15] | G25 | Address[7] | Digital Lead 15 |

**Table 10 connections from the DE2 GPIO header to the Agilent 3000 logic analyzer**

The assignments in column 3 of **table 10** are the ones we will be examining with the logic analyzer. Before we can do this we will have to connect the digital logic pins to the DE2 board. Using column 1 and 4 of **table 10** connect the digital logic probe pins up. The result should look like **figure 31**

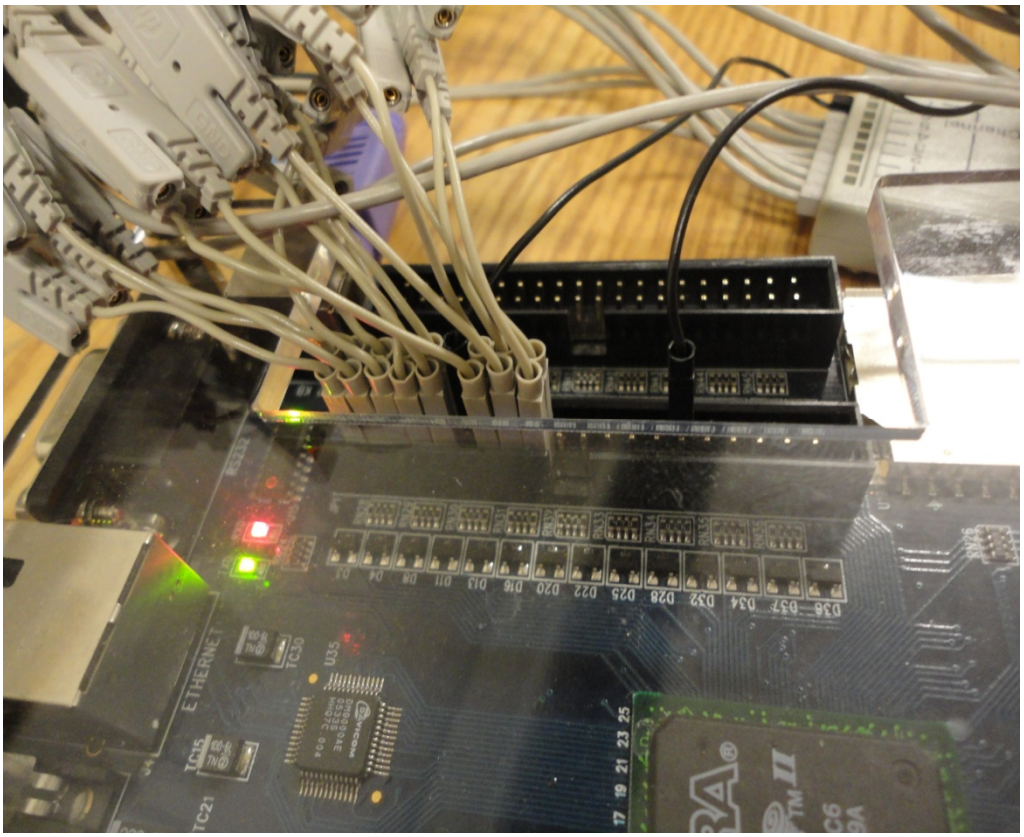


Figure 31 connecting digital logic probe pins from analyzer to DE2 40 pin header JP0

Now power up the DE2 board and download camera.sof which should have been created when the project was compiled.

**Switch 0** and **switch 1** on the DE2 board must be set in the up position. The rest of the switches must be in the down position. See **figure 32.**

Connect a camcorder with a composite video output to the composite video input on the DE2 board. See **figure 32**.

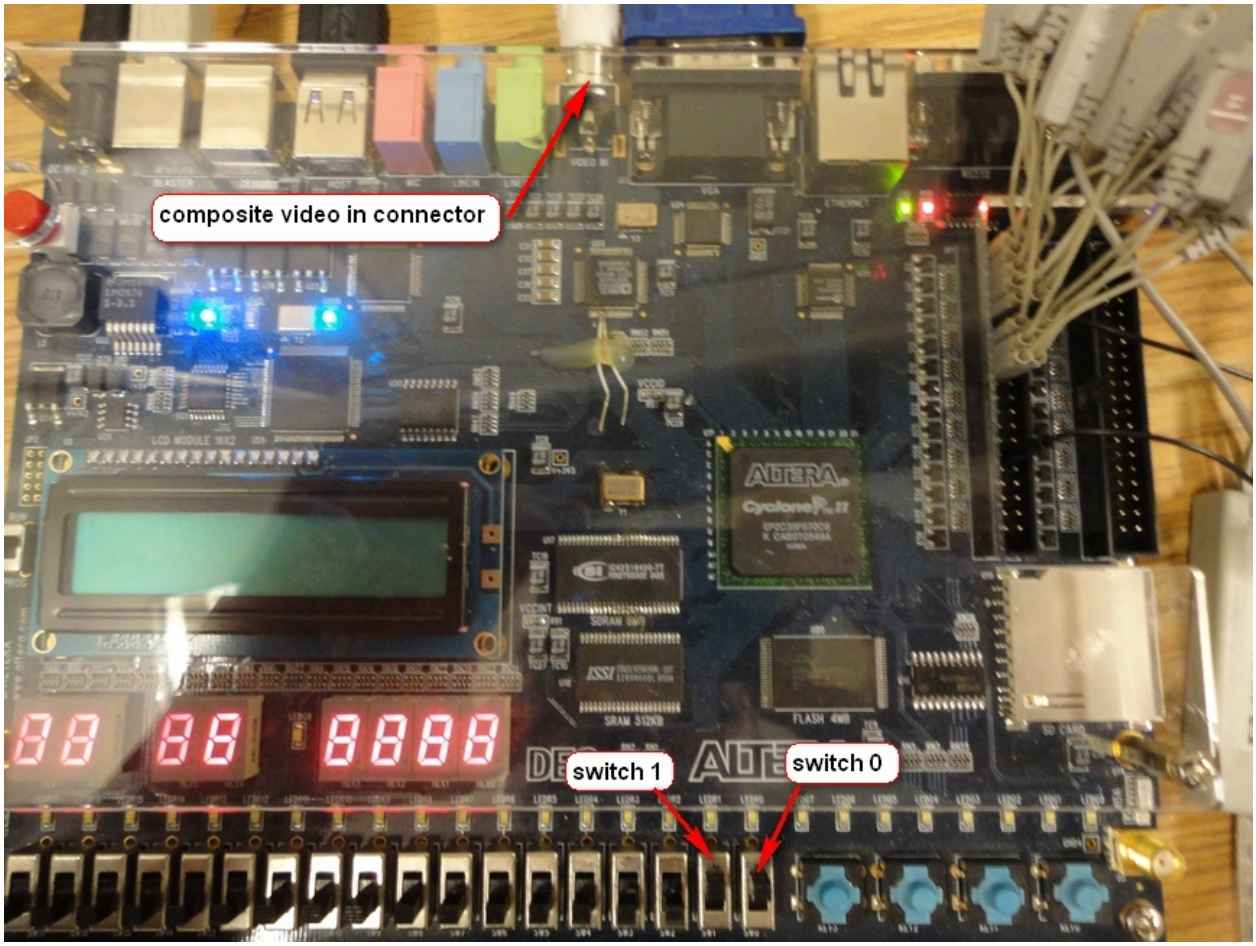


Figure 32 switch setting DE2 board Tutorial 2

Next we need to set up the logic analyzer.

- Press **Default Setting** on logic analyzer.
- Press **Digital.**
- Turn **D15-D8** off and Turn **D7-D0** on.
- Press **Label.**
- Label D0 to D7 as in **table 11**

| Digital lead default name | Rename label |
|---|---|
| D0 | CLK_27 |
| D1 | WE |
| D2 | UD |
| D3 | LD |
| D4 | VERT |
| D5 | HORIZ |
| D6 | FRAME |
| D7 | VID_HS |

Table 11 -label digital logic pin on analyzer

- The result should look like "**New label names**" in **figure 33**.
- Press **Digital.**
- Press **Bus** and enable Bus 1.
- Select Channel D8 to D15 and add to Bus 1.
- Press **Label** and rename Bus 1 to **address.**
- Set delay to **0.0s.** See **figure 33**.
- Set **horizontal** frequency to **5.000** m/s**.** See **figure 33**.
- Press **Digital.**
- Set **scale** to medium. See **figure 33**.

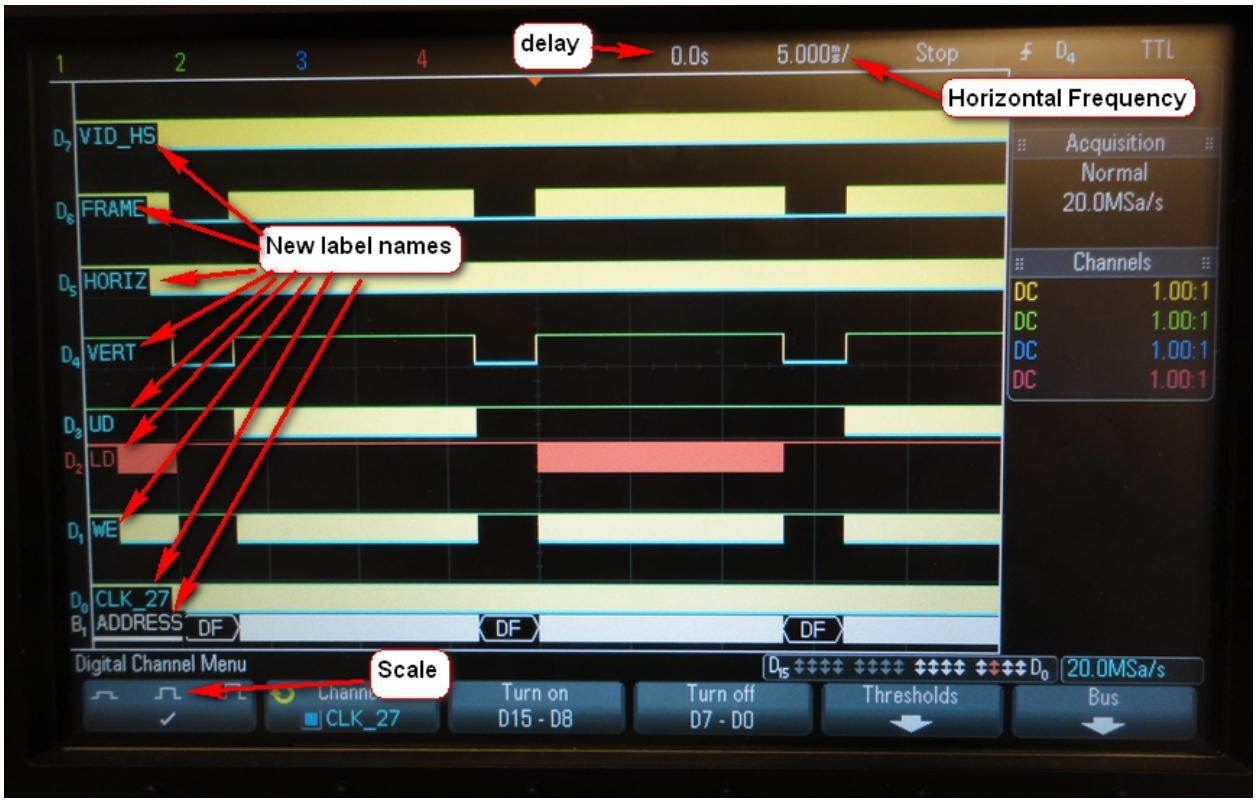


**Figure 33 -newly set up setting for labels, delay and frequency**

- Press **Trigger.**
- Set **mode** to Edge trigger
- Set **source** to VERT
- Set **slope** to rising edge
- Press **Single.**

The result should look like **figure 34**


**Figure 34- trigger result tutorial 2 multiple**

Go back to Quartus and scroll to line 105 of the camera.v verilog code. See **Figure 35**

```
104
105   wire vid_ldll = (frame & !timer2[0] & timer2[1] & !vid_address_low[0] ) ? 1'b0 : 1'b1; // low odd address enable
106   wire vid_udll = (frame & timer2[0] & timer2[1] & !vid_address_low[0] )  ? 1'b0 : 1'b1; // low even address enable
107   wire vid_ldlh = (frame & !timer2[0] & !timer2[1] & !vid_address_high[0] ) ? 1'b0 : 1'b1; // high odd address enable
108   wire vid_udlh = (frame & timer2[0] & !timer2[1] & !vid_address_high[0] )  ? 1'b0 : 1'b1; // high even address enable
109
110   wire we = (frame & !vid_address_low[0] ) ? 0 : 1; // write enable memory
111   wire ud = (frame & timer2[0] & !vid_address_low[0]) ? 0 : 1; // even byte enable
112   wire ld = (frame & !timer2[0] & !vid_address_low[0]) ? 0 : 1; // odd byte enable
113
114   wire vert = ( ( timer1 > 30 &  timer1 <= 240 ) ) ? 1 : 0; // adjust vertical sync
115   wire horiz = ( ( horizontal > 300 & horizontal <= 1548) ) ? 1 : 0; // adjust horizontal sync
116   wire frame = ( horiz  & vert) ? 1'b1 : 1'b0 ; // address range enable
```

**Figure 35 timing verilog code video in**

- From line 114 (figure 35) **VERT** represents the vertical time pulse for each frame. When there is an active low to active high transition that is the beginning of a new frame **Note** that we made an adjustment as to when we start saving vertical data. We are waiting till **30** vertical lines

have been counted before starting to save data. Then we stop when we have reached **240** lines. So in total we are saving **210** vertical lines of video data (240-30= 210).

- From line 115(figure 35) **horiz** represents the number of video bytes per line (horizontal line). Here we are waiting till 300 data pixels have been counted before we start saving to memory. Once the counter has reached 1548 it will stop saving to memory. So we are storing 1248 bytes per line (1548-300=1248). **Note** video pixel data is sent as 16 bits by the video in chip.(More will be explained about this later).Since we are only storing 8 bits we will have to divide 1248 by 2 which is 624 bytes per line.
- From line 116 (figure 35) **frame** represents 1 frame of video data. So in this case 624 horizontal bytes times 210 vertical lines. (624 X 210= 131040)
- From line 111 (figure 35) **ud** represents even lines.
- From line 112 (figure 35) **ld** represents odd lines
- From line 110 (figure 35) **we** represents write enable for both **ud** and **ld**.
- **Note** that from (figure 35) **ud** and **ld** are not enabled at the same time. This distinguishes where in memory the odd lines and even lines are located. This will become very helpful for retrieving video data and displaying video pixels on a monitor.
- From line 105 and 106 (figure 35) **vid_ldll** and **vid_udll** are enables for memory locations from 0 to 256K. If you scroll to line 219 you will see how address is set to **vid_address_low** which starts at memory location 0 and updates values of **video_in** to **data_low** or **data_high** depending on whether **vid_ldll** or **vid_udll** are active low. See **figure 36** for verification of the above.
- From line 107 and 108 (figure35) vid_ldlh and vid_udlh are enables for memory locations from 256K to 512K. . If you scroll to line 242 you will see how address is set to **vid_address_high** which starts at memory location 256k and updates values of **video_in** to **data_low** or **data_high** depending on whether **vid_ldlh** or **vid_udlh** are active low. See **figure 36** for verification of the above.

```
219  ////////////////////////////////////////
220     if ( !vid_ldll )
221
222           begin
223           address <= vid_address_low[18:1]; // odd rows
224           data_low <= video_in;
225           end
226
227        else
228
229           if ( !vid_udll )
230
231           begin
232           address <= vid_address_low[18:1]; // even rows
233           data_high <= video_in;
234           end
235
236        else
237
238  ////////////////////////////////////////
239  // high byte memory frame load        ///
240  ////////////////////////////////////////
241
242           if ( !vid_ldlh )
243
244           begin
245           address <= vid_address_high[18:1]; // odd rows
246           data_low <= video_in;
247           end
248
249        else
250
251
252           if ( !vid_udlh )
253
254           begin
255           address <= vid_address_high[18:1]; // even rows
256           data_high <= video_in;
257           end
258
```

**Figure 36** video data update to memory upper and lower memory

**Figure 37** gives a zoomed view of how each single cycle looks like. A few key observations can be made.

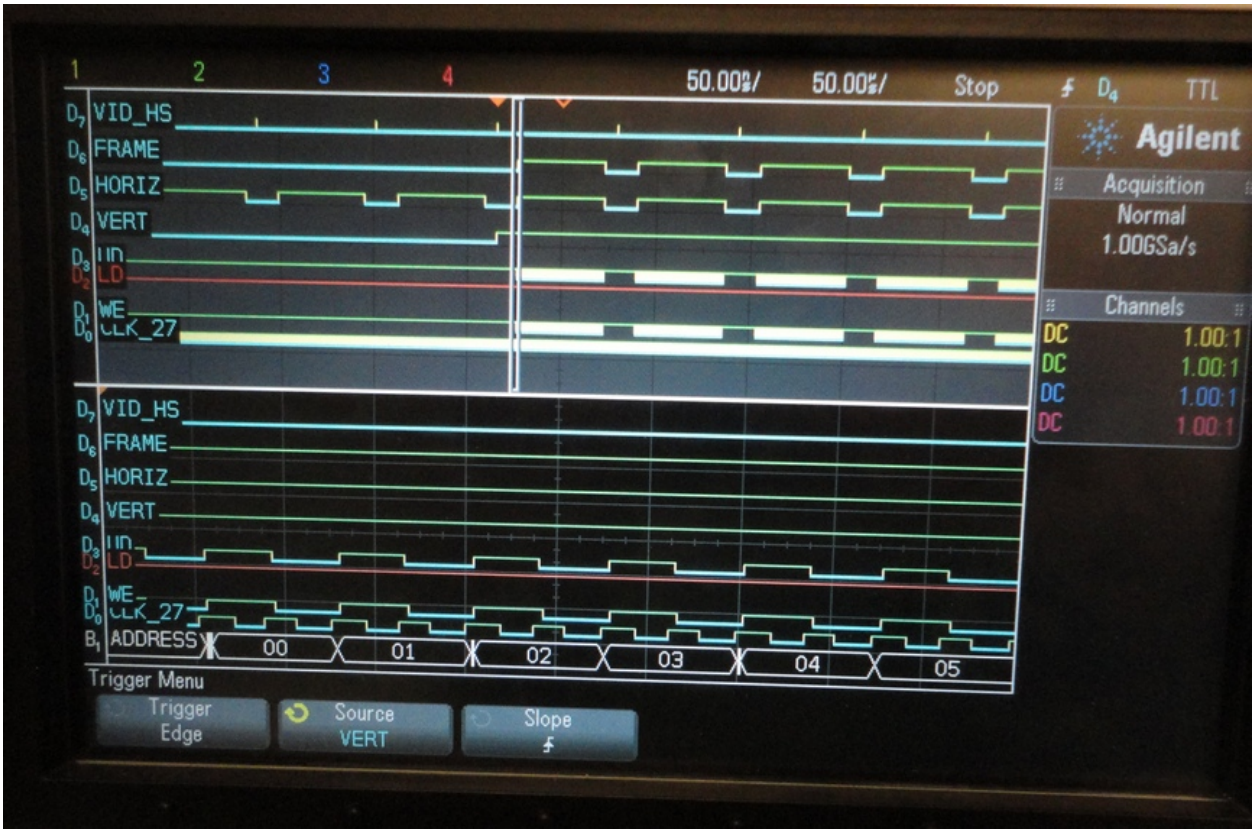


**Figure 37- zoom view interlace**

- The **address** counter increments only after every two clock cycles (27MHz). This keeps in line with what we mentioned earlier that video is sent as 16 bits but we are only capturing 8 bits for this tutorial.
- **ud** and **we** are active low at the same time to enable capture of even lines of video data to be stored at the address location specified. However ld is active high. This is keeping in line with what was said earlier that only **ld** or **ud** is enables during any active high **vert** cycle.

This concludes this tutorial. You should now be familiar on how to create counters and enables to stored video data from a camcorder and store it in memory. The next tutorial will focus on retrieving that data from memory and making sure it is synced correctly.

## VGA interface

The DE2 board uses an Analog Devices ADV7123 triple 10 bit high speed video DAC (digital to Analog converter). The resulting conversion generates red green and blue analog values which are sent to a 15 pin D-SUB connector. For more information on the chip follow the link;

http://www-ug.eecg.utoronto.ca/desl/manuals/ADV7123.pdf

A block diagram of how it is connected can be found in **figure 38**.

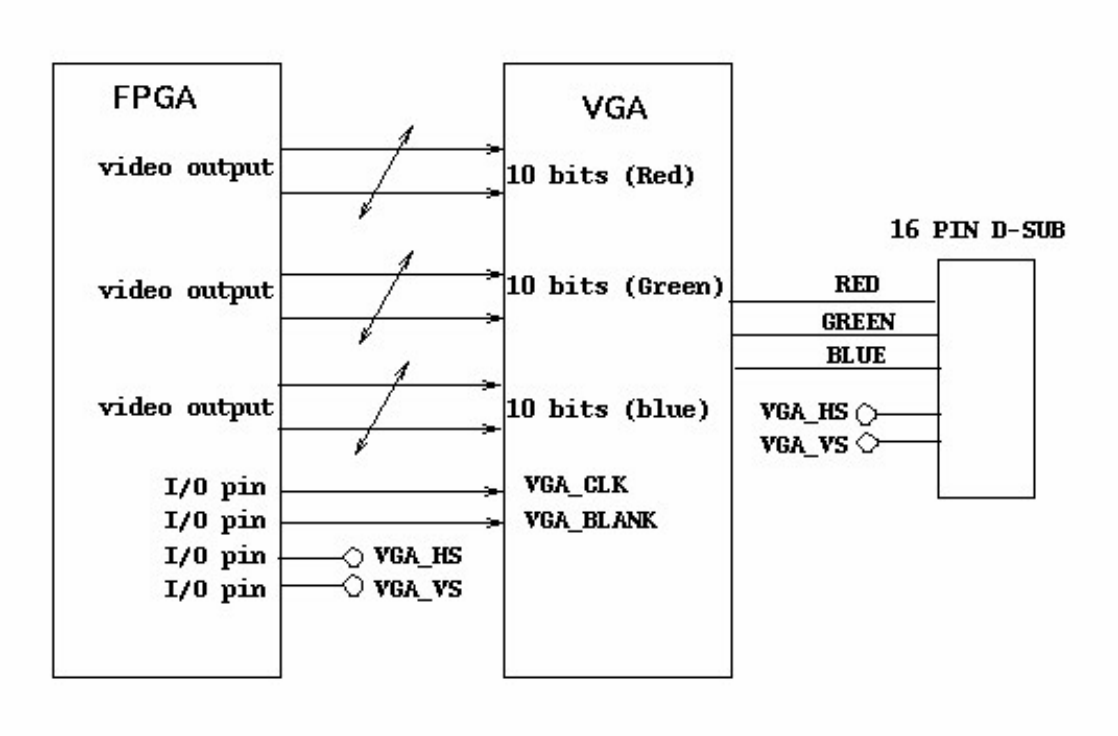


**Figure 38-VGA to FPGA interface**

The pin assignment for the FPGA can be found in **table 12**

| Signal Name | FPGA Pin No. | Description |
|---|---|---|
| VGA_R[0] | PIN_C8 | VGA Red[0] |
| VGA_R[1] | PIN_F10 | VGA Red[1] |
| VGA_R[2] | PIN_G10 | VGA Red[2] |
| VGA_R[3] | PIN_D9 | VGA Red[3] |
| VGA_R[4] | PIN_C9 | VGA Red[4] |
| VGA_R[5] | PIN_A8 | VGA Red[5] |
| VGA_R[6] | PIN_H11 | VGA Red[6] |
| VGA_R[7] | PIN_H12 | VGA Red[7] |
| VGA_R[8] | PIN_F11 | VGA Red[8] |
| VGA_R[9] | PIN_E10 | VGA Red[9] |
| VGA_G[0] | PIN_B9 | VGA Green[0] |
| VGA_G[1] | PIN_A9 | VGA Green[1] |
| VGA_G[2] | PIN_C10 | VGA Green[2] |
| VGA_G[3] | PIN_D10 | VGA Green[3] |
| VGA_G[4] | PIN_B10 | VGA Green[4] |
| VGA_G[5] | PIN_A10 | VGA Green[5] |
| VGA_G[6] | PIN_G11 | VGA Green[6] |
| VGA_G[7] | PIN_D11 | VGA Green[7] |
| VGA_G[8] | PIN_E12 | VGA Green[8] |
| VGA_G[9] | PIN_D12 | VGA Green[9] |
| VGA_B[0] | PIN_J13 | VGA Blue[0] |
| VGA_B[1] | PIN_J14 | VGA Blue[1] |
| VGA_B[2] | PIN_F12 | VGA Blue[2] |
| VGA_B[3] | PIN_G12 | VGA Blue[3] |
| VGA_B[4] | PIN_J10 | VGA Blue[4] |
| VGA_B[5] | PIN_J11 | VGA Blue[5] |
| VGA_B[6] | PIN_C11 | VGA Blue[6] |
| VGA_B[7] | PIN_B11 | VGA Blue[7] |
| VGA_B[8] | PIN_C12 | VGA Blue[8] |
| VGA_B[9] | PIN_B12 | VGA Blue[9] |
| VGA_CLK | PIN_B8 | VGA Clock |
| VGA_BLANK | PIN_D6 | VGA BLANK |
| VGA_HS | PIN_A7 | VGA H_SYNC |
| VGA_VS | PIN_D8 | VGA V_SYNC |

Table 12- pin assignments VGA on FPGA

- VGA_R[0-9] –These are the 10 digital bits that represent the analog RED colour output to the 15 pin D_SUB interface. Depending on the number of red pins used will determine the RED colour values. As an example if you use 4 bits of red R[0-3]you will get $2^4$ red values.
- VGA_G[0-9] –These are the 10 digital bits that represent the analog GREEN colour output on the 15 pin D_SUB interface. Depending on the number of green pins used will determine the GREEN colour values. As an example if you use 6 bits of green G[0-5]you will get $2^6$ green values.
- VGA_B[0-9] –These are the 10 digital bits that represent the analog BLUE colour output on the 15 pin D_SUB interface. Depending on the number of blue pins used will determine the BLUE colour values. As an example if you use 2 bits of blue B[0-1]you will get $2^2$ blue values.
- VGA_clock – The monitor frequency is 25 MHz. This is standard frequency for most VGA monitors.
- VGA_Blank –This signal is used to black out areas on the monitor that you do not want to display video. When active high video display is enabled. This signal must be generated using Verilog code.
- VGA_Hs- This is the horizontal signal that goes to the 15 pin D-SUB connector. This determines the width of each video frame. An active low to high transition on this signal starts a new line of video. This signal must be created using Verilog code.
- VGA_Vs- This is the vertical signal that goes to the 15 pin D-SUB connector. This determines the length of each frame. An active low to high transition on this signal starts a new vertical row. This signal must be created using Verilog code.

Based on the above information we can now start tutorial 3. We will create a video driver to display video on a monitor. **Note** in order to do **tutorial 3** you must have done and understood **tutorial 2.**

## Tutorial 3-Displaying data from memory to a monitor (VGA)

## Part 1- creating counters and generating vsync, hsync and vid_blank

Unlike the video in chip ADV7181 we will need to generate;

- vsync, hsync and vid_blank signals.
- First using the 50 Mhz onboard clock we will divide it to generate the needed 25 MHz monitor clock
- Using this clock create a counter and generate the needed sync pulses.

A working example of what the verilog code can be found at the following link;

http://www-ug.eecg.utoronto.ca/desl

**Select -**DESL Online Tutorials>Tutorial3_monitor.zip.

Create a directory and unzip the file. Using Quartus **new project wizard** create a new project called **monitor** Compile the project. Open the verilog file called **monitor**. Scroll down to line 51. The code should look the same as **figure 39.** If you look at line 63 you will notice that vid_clk <= clkcount[0]. This represents the 50 Mhz having been divide by 2 which equals the need 25 Mhz clock.

```
47  ////////////////////////////////
48  /// general clock divider  ///
49  ////////////////////////////////
50
51   always @ (posedge clk )
52
53  begin
54
55        clkcount <= clkcount + 1;
56
57  end
58
59  ////////////////////////////
60  ///  25 Mhz clock    //////
61  ////////////////////////////
62
63  always vid_clk <= clkcount[0];
64
```

**Figure 39- clock divider 25 Mhz**

Next we will generate both the horizontal and vertical counters. Keep in mind that the standard frame size for VGA video is 640 X480. So our counters have to be greater or equal to these values.

- Clrvidh is used to reset the horizontal counter to zero and then counter up to a value. In this case we have chosen 800. See line 44 in **figure 40**. The counter is generated in **figure 41** starting at line 81
- Clrvidv is used to reset the vertical counter to zero and then counter up to a value. In this case we have chosen 525. See line 45 in **figure 40**. The counter is generated **in figure 41** starting at line 100.

```
36  ////////////////////////////
37  ////    control values  /////
38  ////////////////////////////
39  reg         vid_clk;
40
41  wire        vsync = ((contvidv >= 491) & (contvidv < 493))? 1'b0 : 1'b1;
42  wire        hsync = ((contvidh >= 664) & (contvidh < 760))? 1'b0 : 1'b1;
43  wire        vid_blank = ((contvidv >= 8) & (contvidv <  420) &(contvidh >= 20) & (contvidh < 624))? 1'b1 : 1'b0;
44  wire        clrvidh = (contvidh <= 800) ? 1'b0 : 1'b1;
45  wire        clrvidv = (contvidv <= 525) ? 1'b0 : 1'b1;
```

**Figure 40- counters and sync pulses**

```
79    ////////////////////////////////
80
81    always @ (posedge vid_clk )
82
83    begin
84
85          if(clrvidh)
86          begin
87          contvidh <= 0;
88          end
89
90          else
91          begin
92          contvidh <= contvidh + 1;
93          end
94    end
95
96    ////////////////////////////////////////
97    //vertical counter when clrvidv is low /
98    ////////////////////////////////////////
99
100   always @ (posedge vid_clk)
101
102   begin
103
104         if (clrvidv)
105         begin
106         contvidv <= 0;
107         end
108
109         else
110         begin
111            if
112            (contvidh == 798)
113            begin
114            contvidv <= contvidv + 1;
115            end
116         end
117   end
118
```

**Figure 41- counters generated**

- Vsync pulse - active low when contvidv is greater than or equal to 491 and less than 493 else it is active high. See **figure 40**.
- hsync pulse - active low when contvidh is greater than or equal to 664 and less than 760 else it is active high. See **figure 40**.
- vid_blank- active high if contvidv is great than or equal to 8 and less than 420 and contvidh is greater than and equal to 20 and less than 624. See **figure 40**.

Let's examine the signals and verify the above hardware code. Connect the digital logic pins of the logic analyzer to the 40 pin header GPIO-0 as in **table 13** column 1 and 3. Label the pin names according to column 4 of **table 13**.

| 40 pin HeaderJP0 | Pin assignment | Digital lead default name | Rename label |
|---|---|---|---|
| GPIO[0] | D25 | D0 | VID_CLK |
| GPIO[1] | J22 | D1 | VSYNC |

| | | | |
|---|---|---|---|
| GPIO[2] | E26 | D2 | HSYNC |
| GPIO[3] | E25 | D3 | VID_BLANK |
| GPIO[4] | F24 | D4 | CLRVIDH |
| GPIO[5] | F23 | D5 | CLRVIDL |

Table 13- pin assignments clk and sync pulses

Do the following procedures on the logic analyser;

- Press **Digital.**
- Press **Bus** and disable all the Buses.
- Press **back**
- De-select Channel D8 to D15 and select D5-D0.
- Press **Label** and rename D0-D5 according to **table 13** column 4.
- Set delay to **0.0s.**
- Set **horizontal** frequency to **42.000n/s.**
- Press **Digital.**
- Set **scale** to high.
- Press **Trigger**.
- Set **Edge** trigger.
- Set **VSYNC** as source.
- Set **rising Edge** as slope.
- Press **Single** to trigger an event.

The result should look similar to **figure 42**.

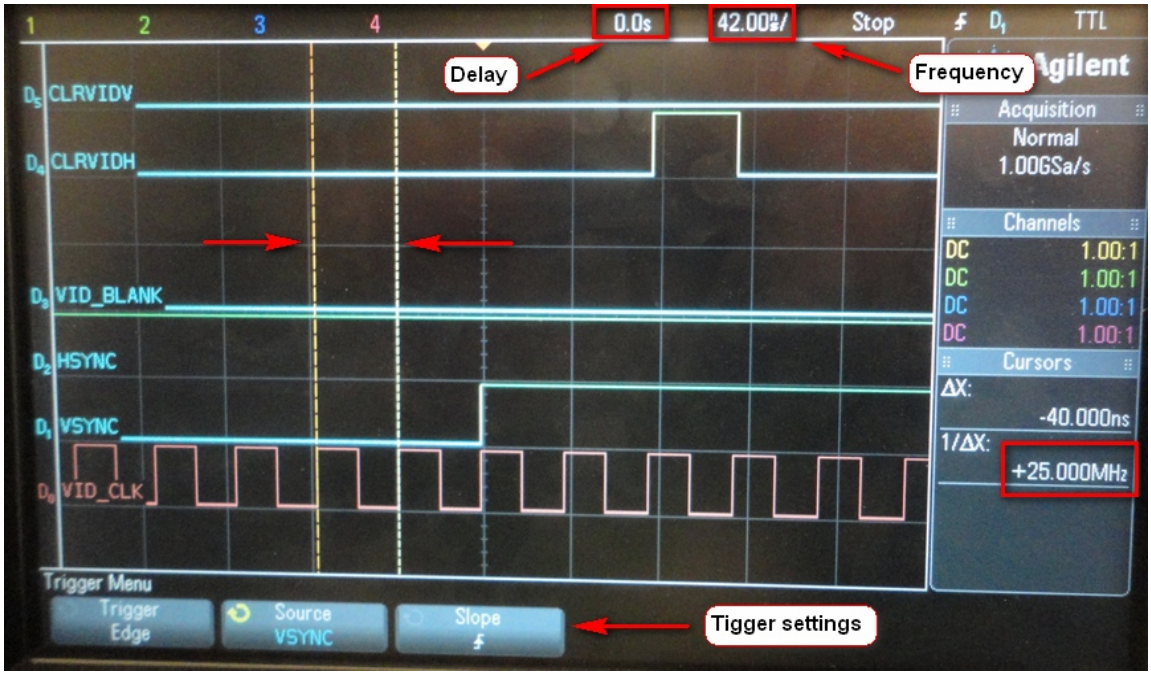


**Figure 42-clock frequency measurement**

- Press **Cursers** and measure the (1/DELTA) frequency of Vid_clk.
- The result should be around 25 MHz, depending on how the cursors are set. See **figure 42.**
- Change frequency to **2.200** m/s.
- Press **Single** to retrigger event.
- Press **zoom.**
- Set zoom frequency to **86.00 u/s**

The result should look **figure 43**


**Figure 43**- sync pulses

**Note** in **figure 43** where the label **Restart** is. This is where the counters are reset to zero and new count begins. Vid_blank becomes active high after 8 hsync cycles. This is also the beginning of a new video frame.

We have now generated all the sync pulses. Now we need to add the address counter and memory enables before the code will be complete.

## Part 2- Creating and examining address counters and enables

If we recall from **tutorial 2** we created a driver that took composite video from a camcorder and stored the digital pixel values in SRAM memory on the DE2 board.

- The memory was divided into two halves.
- One half of memory was used to store the previous frame from the camcorder.
- The other half was used to update the present frame.
- See **Figure 27** for a pictorial view and a review of the previous 3 bullet points.
- Two frames were required to create a full video frame to display.
- **Ud** was used to store all the even lines of video data.
- **Ld** was used to store all the odd lines of video data.
- There were a total of 210 vertical lines. Also note we started after 30 vertical lines and end at 240 vertical lines (240-30= 210). So s full frames was 210x2= 420
- There were a total of 624 horizontal video pixels (8 bits per pixel). Again note we started the counter at 150 pixels in and stopped at 774 pixels (774-150=624)

Also recall each video pixel is 8 bits. This video format is called 4:2:2 (4 red bits, 2 green bits, and 2 blue bits) for more information on this video format and others go to the following link.

http://en.wikipedia.org/wiki/Chroma_subsampling

With this information we can now create our address counters. A total of 4 address counters will be required;

- Lower address counter odd lines.
- Lower address counter even lines.
- Upper address counter odd lines.
- Upper address counter even lines.

A working example of what the verilog code should look like can be found at the following link;

http://www-ug.eecg.utoronto.ca/desl

**Select**-DE2>DESL Online Tutorials>Tutorial3_monitor_full.zip.

Create a directory and unzip the file. Using Quartus **new project wizard** create a new project called **monitor**. Compile the project. Open the Verilog file called **monitor**.

```
73  ////////////////////////////////////////////////////////
74  //////// memory enables
75  ////////////////////////////////////////////////////////
76
77  wire        ramvidv = ( ( (contvidv <= 420) ) ? 1'b0 : 1'b1);
78  wire        adden = ( (( contvidh < 624) & (contvidv <= 420) ) ? 1'b1 : 1'b0); // address enable
79  wire        readh = (!vid_clk & adden & oddeven) ? 0 : 1; // even byte memory enable
80  wire        readl = (!vid_clk & adden & !oddeven)  ? 0 : 1; // odd byte memory enable
81  wire        read = (!vid_clk & adden) ? 0 : 1; // oe to memory enable
82  wire        read_lowl = (!framem & adden & !oddeven ) ? 0 : 1; // low odd address enable
83  wire        read_highl = (!framem & adden & oddeven ) ? 0 : 1; // low even address enable
84  wire        read_lowh = (framem & adden & !oddeven ) ? 0 : 1; // high odd address enable
85  wire        read_highh = (framem & adden & oddeven ) ? 0 : 1; // high even address enable
86
87  parameter address_high = 18'h20000; //top address start 256 meg
88  parameter address_low = 18'h00000;  // lower address start at 0 meg
89
90  always framem <= framev[0];
91  always oddeven <= contvidv[0];
```

**Figure 44- address enables odd and even**

Scroll to line 75. It should look like **figure 44**. These are all the enables for the address counters.

- Ramvidv - is used to rest the address counters. See **figure 45** line 175 and 195 or **figure 46** line 217 and line 239. Depending on if it is upper memory or lower memory it will be zero or 256K. This is determined by **parameter**.
- Line 88 parameter **address_low** is set to HEX zero. See **figure 45** line 177 and line 197.
- Line 87 parameter **address_high** is set to HEX 256K. See **figure 46** line 219 and line 241.
- Adden – is the address enable. If convidv is less than 624 and contvidv is less than or equal to 420 **adden** is active high otherwise it is active low.
- Readh- is even memory enable (**ud** is enabled on SRAM memory chip) active low.
- Readl- is odd memory enable (**ld** is enabled on SRAM memory chip) active low.
- Read- is used to enable **oe** on SRAM memory chip, active low.

```
167   ////////////////////////////////////////////////////////
168   // address counter out to monitor odd lines low memory
169   ////////////////////////////////////////////////////////
170
171   always @ (posedge vid_clk )
172
173   begin
174
175         if(ramvidv)
176         begin
177         ramaddressl_odd <= address_low;// memory reset to "0"
178         end
179
180         else
181         begin
182         if (adden & !oddeven & !framem)
183            begin
184            ramaddressl_odd <= ramaddressl_odd + 1;
185            end
186         end
187   end
188   ////////////////////////////////////////////////////////
189   // address counter out to monitor even lines low memory
190   ////////////////////////////////////////////////////////
191   always @ (posedge vid_clk)
192
193   begin
194
195         if (ramvidv)
196         begin
197         ramaddressl_even <= address_low; // memory reset to "0"
198         end
199
200         else
201         begin
202            if
203            (adden & oddeven & !framem)
204            begin
205            ramaddressl_even <= ramaddressl_even + 1;
206            end
207         end
208   end
```

**Figure 45**- lower memory counter odd and even

```
210   ////////////////////////////////////////////////////////////
211   // address counter out to monitor odd lines high memory
212   ////////////////////////////////////////////////////////////
213   always @ (posedge vid_clk )
214
215   begin
216
217         if(ramvidv)
218         begin
219         ramaddressh_odd <= address_high; // memory reset to 256K
220         end
221
222         else
223         begin
224         if (adden & !oddeven & framem)
225            begin
226            ramaddressh_odd <= ramaddressh_odd + 1;
227            end
228         end
229   end
232   ////////////////////////////////////////////////////////////
233   // address counter out to monitor even lines high memory
234   ////////////////////////////////////////////////////////////
235   always @ (posedge vid_clk)
236
237   begin
238
239         if (ramvidv)
240         begin
241         ramaddressh_even <= address_high; // memory reset to 256K
242         end
243
244         else
245         begin
246            if
247            (adden & oddeven & framem)
248            begin
249            ramaddressh_even <= ramaddressh_even + 1;
250            end
251         end
252   end
```

**Figure 46-Upper address counter odd and even**

- Read_lowl- enables lower odd address counter and latches data from SRAM memory, active low. See **figure47** line 260.
- Read_highl- enables lower even address counter and latches data from SRAM memory , active low. See **figure47** line 270.
- Read_lowh- enables upper odd address counter and latches data from SRAM memory , active low. **See figure47** line 280.

- Read_highh- enables upper even address counter and latches data from SRAM memory , active low. See **figure47** line 290.

```
254   ////////////////////////////////////////////////////////////////////
255   //////// latch address and data from memory
256   ////////////////////////////////////////////////////////////////////
257   always @ (posedge vid_clk)
258
259   begin
260         if (!read_lowl )
261         begin
262
263         video_mem <= data_low;
264         address_mem <= ramaddressl_odd; // low memory odd byte
265         end
266
267         else
268
269
270         if (!read_highl )
271         begin
272
273         video_mem <= data_high;
274         address_mem <= ramaddressl_even; // low memory even byte
275         end
276
277         else
278
279
280         if (!read_lowh )
281         begin
282
283         video_mem <= data_low;
284         address_mem <= ramaddressh_odd; // high memory odd byte
285         end
286
287         else
288
289
290         if (!read_highh )
291         begin
292
```

```
293         video_mem <= data_high;
294         address_mem <= ramaddressh_even; // high memory even byte
295         end
296
297         else
298
299         begin
300         address_mem <= 18'bz; // if not true memory is tri state
301
302         end
303 end
```

**Figure 47- latch enables upper and lower memory**

Let's examine the signals and verify the above hardware code. Connect the digital logic pins of the logic analyzer to the 40 pin header GPIO-0 as in **table 14** and re-label the pin names according to column 4.

| 40 pin HeaderJP0 | Pin assignment | Digital lead default name | Rename label |
|---|---|---|---|
| GPIO[0] | D25 | D0 | FRAMEM |
| GPIO[1] | J22 | D1 | READL |
| GPIO[2] | E26 | D2 | READH |
| GPIO[3] | E25 | D3 | READ |
| GPIO[4] | F24 | D4 | READ_LL |
| GPIO[5] | F23 | D5 | READ_LH |
| GPIO[6] | J21 | D6 | READ_HL |
| GPIO[7] | J20 | D7 | READ_HH |
| GPIO[8] | F25 | D8 | Address (bus) |
| GPIO[9] | F26 | D9 | Address (bus) |
| GPIO[10] | N18 | D10 | Address (bus) |
| GPIO[11] | P18 | D11 | Address (bus) |
| GPIO[12] | G23 | D12 | Address (bus) |
| GPIO[13] | G24 | D13 | Address (bus) |
| GPIO[14] | K22 | D14 | Address (bus) |
| GPIO[15] | G25 | D15 | Address (bus) |

**Table 14 -Tutorial 3 Part 2**

Do the following on logic analyser;

- Press **Digital.**
- Press **Bus** and enable **Bus 1**.
- Set D15-D8 to represent that bus
- Press **back.**
- Press **Digital.**
- Set scale to be **medium.**
- D5-D0 should still be selected from part 1. Select D7 and D6
- Press **Label** and rename D0-D7 according to table 14 column 4.

- Set delay to **0.0s.**
- Set **horizontal** frequency to **10.00**m/s**.**
- Press **Trigger**.
- Set **Edge** trigger.
- Set **FRAMEM** as source.
- Set **rising Edge** as slope.
- Press **Single** to trigger a new event.

The result should look like **figure48.**


**Figure 48- address and data enable multiple frames**

Note that when FRAMEM is active low READ_LL and READ_LH are active and READ_HL and READ_HH remain active high. This means that only the low half of memory is being updated and the upper memory remains constant. The opposite is true when FRAMEM is active high.

- Press **zoom.**
- Set **Delay** to **1.14**m/s.
- Set zoom frequency to **50.00**u/s.

Here we can see that READ_HL and READ_HH are never active at the same time. See **figure 49** below.

- READ_HL is only active when READL is active (odd lines of video)
- READ_HH is only active when READH is active (even lines of video)

- We see that every other cycle odd lines are being display to the monitor (READ_LL or READ_HL)
- We see that every other cycle even lines are being display to the monitor (READ_LH or READ_HH)


**Figure 49**- frame zoom in on enables

Go back to Verilog code **monitor.v** and change scroll to line 308. Change entry **FRAMEM** to **ADDEN.** This means **GPIO[0]** is now pointing to **ADDEN** as opposed to **FRAMEM**.

Save the project and recompile it. Download it to the DE2 board.

- Press **Zoom** again to turn it off.
- Press **label.** Change name of **D0** from **FRAMEM** to **ADDEN**
- Set frequency to **70.00**n/s**.**
- Set delay to **257.0**ns.
- Press **Trigger**
- Set trigger mode to **Pattern.**
- Set Qualifier to **Entered**
- Set Channel to **ADDEN.**
- Set Pattern to **rising edge.**
- Set Channel to **READ_LL.**
- Set Pattern to **0.**

- Press **Single.**

The result should look like **figure 50**. **Note** only **READ_LL** is active and after each transition the address counter increments to the next memory location. These are the odd lines lower half of memory (0-256k) that is being updated.

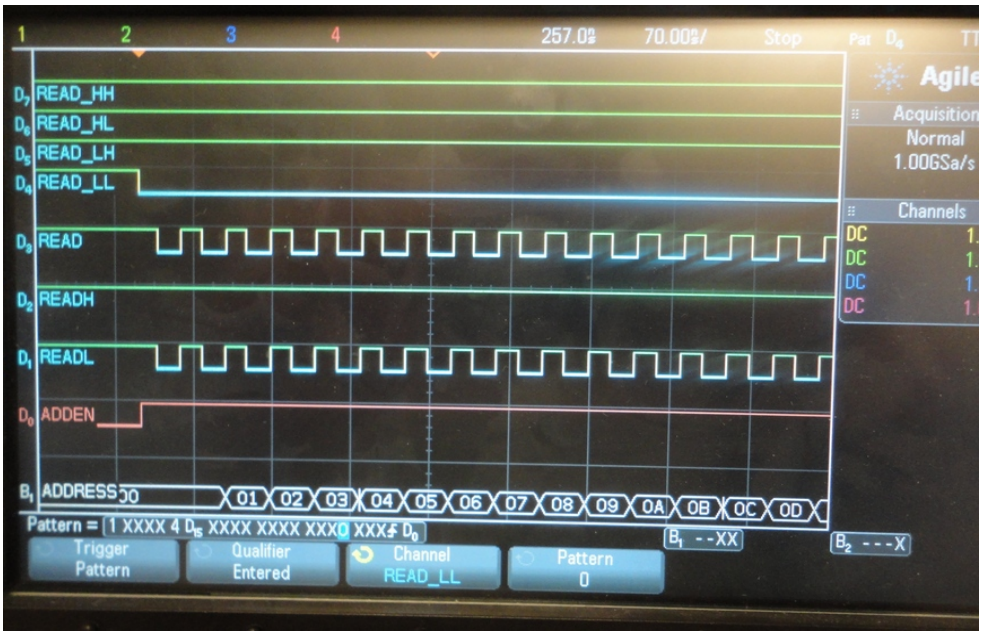


**Figure 50- odd lines lower half of memory**

- Set Channel to **READ_LH.**
- Set Pattern to **0.**
- Press **Single.**

The result should look like **figure 51**. Note only **READ_LH** is active and after each transition the address counter increments to the next memory location. These are the even lines lower half of memory (0-256k) that is being updated.

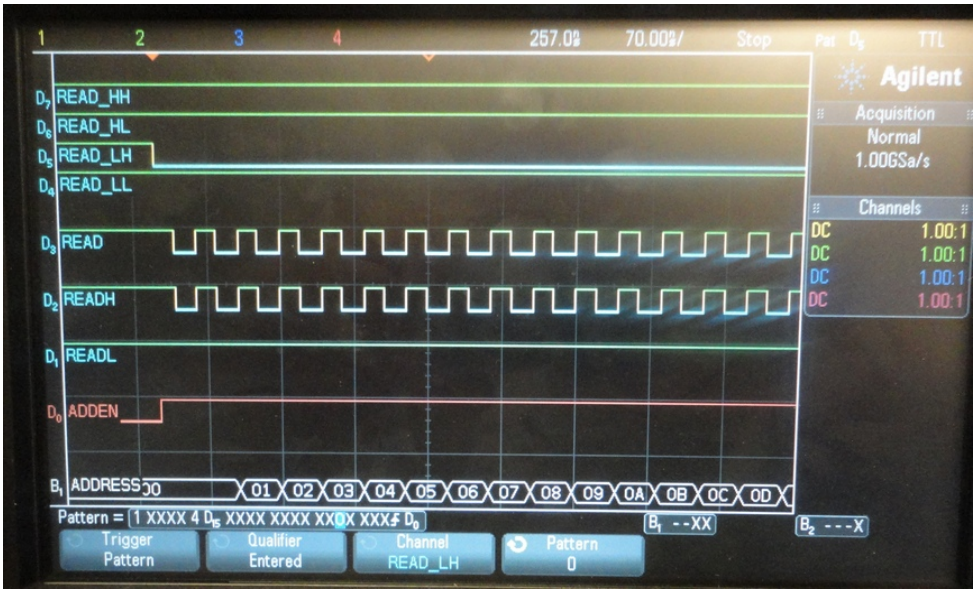


**Figure 51-even lines lower half of memory**

- Set Channel to **READ_HL.**
- Set Pattern to **0.**
- Press **Single.**

The result should look like **figure 52**. Note only **READ_HL** is active and after each transition the address counter increments to the next memory location. These are the odd lines upper half of memory (256-512k) that is being updated.

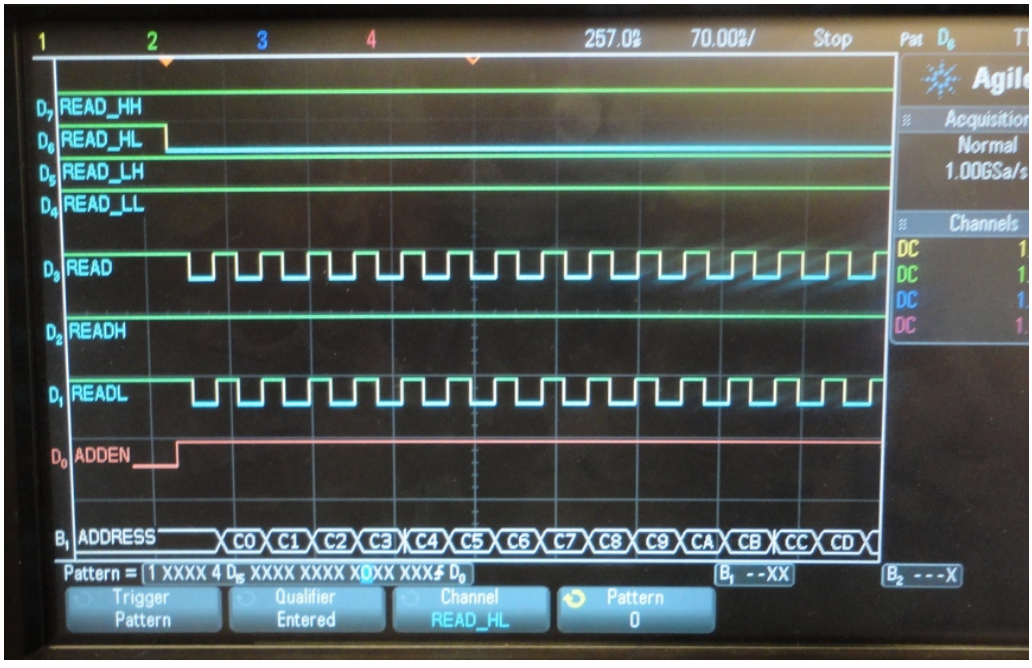


**Figure 52-odd lines upper half of memory**

- Set Channel to **READ_HH.**
- Set Pattern to **0.**
- Press **Single.**

The result should look like **figure 53**. Note only **READ_HH** is active and after each transition the address counter increments to the next memory location. These are the even lines upper half of memory (256-512k) that is being updated.

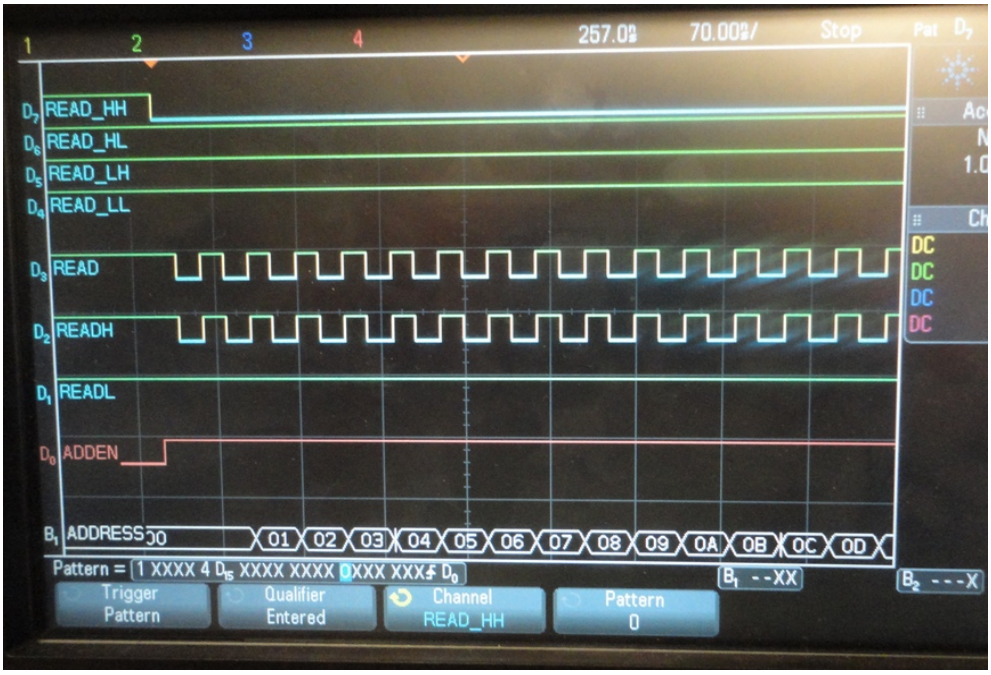


**Figure 53-even lines upper half of memory**

We have now created the address counters and data enables to retrieve data from the upper half of memory and lower half of memory. More importantly we have done it in a manor where by one half of memory is being displayed to the monitor the other half of memory can be updated with new video data values. Our final objective will be to create a state machine that will sync the camera.v program created in part 1 with the monitor.v program created part 2. We will do this in part3. This concludes part 2.

## Part 3- Creating a data memory retrieval and memory update state machine.

In part 1 we created a state machine to capture video data from a video source (a camcorder) and then store it in SRAM memory. Lower data bits (D0 –D7) were used to store odd lines of data and upper data bits (D8-D15) were used to store even lines of data. The SRAM memory on the DE2 board is 512 Kbytes of memory. We split the SRAM memory into two halves 0 to 256Kbytes (lower half) and 265Kbytes to 512Kbytes (upper half). We then built a state machine that would update one half of memory with new

video values and the other half of memory would remain as is. At the end of each frame the roles would be reversed.

In part 2 we create a state machine that displayed video data from memory to a monitor. Using the principles from part 1 we displayed either the upper or lower part of SRAM memory.

In part 3 we will combine both of these state machines carmera.v and monitor.v and synchronize them such that when video data is being updated in one half of memory the other half is displaying to the monitor. At the end of the update the roles are reversed. The key is to find the best way to sync the two pieces of code. Let us make a few observations;

- The video in data state machine is running at 27 Mhz
- The video out data state machine is running at 25 Mhz
- Once video data has been displayed to the monitor it can be updated.
- At the end of each frame a smooth update transition needs to take place.
- **Address**, **data_low**, **data_high** , **ud** and **ld** will be required for reading and writing to the SRAM memory.
- **Oe** (output enable) will need to be active low when reading from SRAM memory. **Data _low** and **Data_high** will have to be tri-stated and buffered so you can read the data values from memory.
- **We (**write**)** will need to be active low when writing to SRAM memory
- **Cs** (chip select) is active low for the SRAM to be enabled.
- We need to make a decision as to whether to sync the state machine to video in (camera.v) or video out (monitor.v) **.**

Let us expand the last point. If we use video out (read from memory) as the synchronizing video that means we would asynchronously have to latch the new video during times when video is not being read from memory. This means we need to make a temporary location to store values coming from the video in source. Then update memory during non display enable times. Look at **figure 54.**

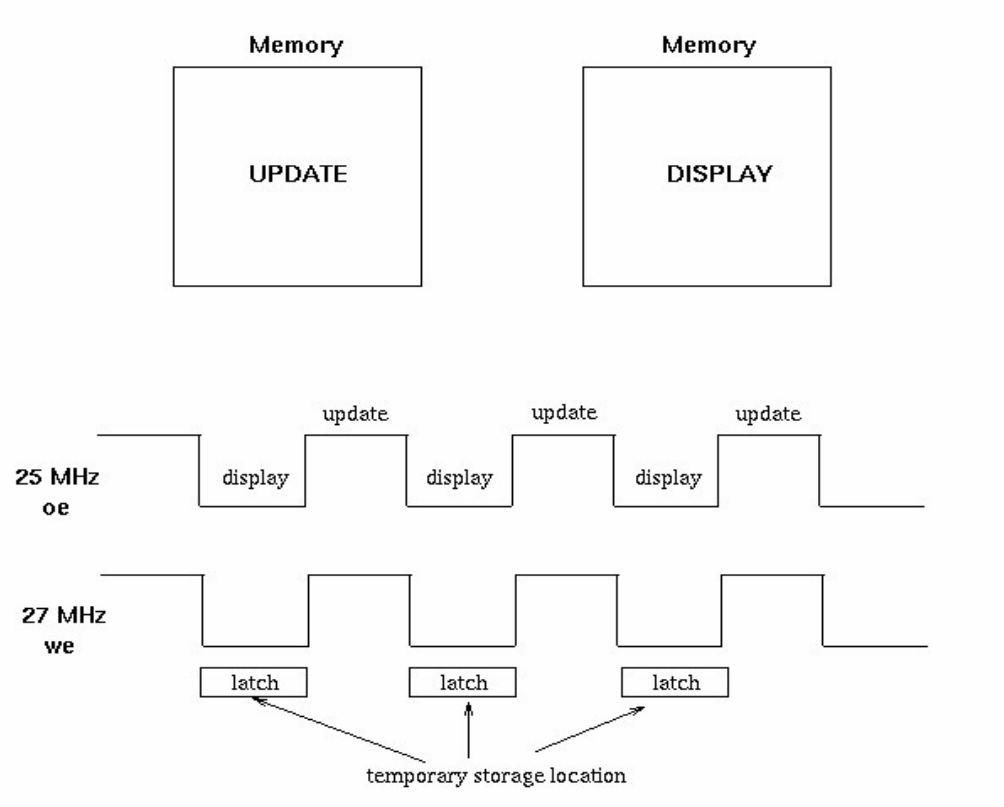


**Figure 54- sync video in with video out**

- When **oe** is active low video is displayed to memory
- That means during **oe** active high periods video_in (latched) can be updated
- At the end of each video display frame the memory would be swapped and the process would start over again.

A working example of what the Verilog code should look like can be found at the following link;

http://www-ug.eecg.utoronto.ca/desl

If you have an older version of the DE2 board where the serial number starts with 0 use the following:

**Select -**DESL Online Tutorials>Tutorial3_video_D13.zip.

The newer versions of the DE2 board where the serial number starts with 1 and has the ADV7181B video decoder use the following;

**Select**-DE2>DESL Online Tutorials>Tutorial3_video_C16.zip.

The newer versions of the DE2 board where the serial number starts with 1 and has the ADV7180 video decoder use the following;

**Select**-DE2>DESL Online Tutorials>Tutorial3_video_v3.zip.

Create a directory and unzip the file. Using Quartus **new project wizard** create a new project called **video**. Compile the project. Open Verilog file called video. The following is for the DE2 board starting with 0 as the serial number. The line numbers may be off for the other two versions of the DE2.

Scroll to line 159 of the file and you should see the follow code. See **figure 55.**

```
157
158
159   wire[7:0] videorgb = ((!read ) ? video_mot : 8'b0);
160
161   wire[7:0] data_high = ((!readh  ) ? 8'bz : data_camh);
162   wire[7:0] data_low = ((!readl  ) ? 8'bz : data_caml);
163
164   wire[17:0] address = ((!read ) ? address_mot : address_cam);
165
166   wire ld = ((!read ) ? readl : ld_cam);
167   wire ud = ((!read ) ? readh : ud_cam);
168   wire oe = ((!read ) ? 0 : 1);
169   wire we = ((!read  ) ? 1 : we_cam);
170
```

**Figure 55- sync verilog code**

When **read** is active low, data from the SRAM is displayed at a pixel location on the display. According to **figure 55** the following can be concluded when **read** is active low;

- Address <= address_mot (address counter keeps track of where each pixel should be located)
- Videorgb <= video_mot ( this is the 8 input video bits to the ADV7123 video decoder)
- data low <= tri state (read lower 8 bits of data from memory)
- data high <= tri state(read higher 8 bits of data from memory)
- Ld <= readl
- Ud <= readh
- Oe <= active low
- We <= active high

Open monitor.v scroll to line 233, you will notice that when read_lowl or read_highl or read_lowh or read_highh is active low **video_mot** and address_**mot** are updated, see **figure 56**. These updated values are then updated during the next active low of **read, s**ee **figure 55**.

```
233  begin
234        if (!read_lowl )
235        begin
236
237        video_mot <= data_low;
238        address_mot <= ramaddressl_odd; // low memory odd byte
239        end
240
241        else
242
243
244        if (!read_highl )
245        begin
246
247        video_mot <= data_high;
248        address_mot <= ramaddressl_even; // low memory even byte
249        end
250
251        else
252
253
254        if (!read_lowh )
255        begin
256
257        video_mot <= data_low;
258        address_mot <= ramaddressh_odd; // high memory odd byte
259        end
260
261        else
262
263
264        if (!read_highh )
265        begin
266
267        video_mot <= data_high;
268        address_mot <= ramaddressh_even; // high memory even byte
269        end
270
```

**Figure 56- monitor update enable signals syncs**

When **read** is active high updated video data from the camcorder is loaded into the SRAM memory. According to **figure 55**;

- Address <= address_cam (address counter keeps track of where updated video data should be stored)
- Ld <= ld_cam
- Ud <= ud_cam
- Data_high <= data_camh (write odd 8 bits of data to memory)
- Data_low <= data_caml (write even 8 bits of data to memory)
- Oe <= active high
- We <= we_cam

Open camera.v and scroll to line178 you will notice that when vid_ldll or vid_udll or vid_ldlh or vid_udlh are active low data caml, data_camh and address_cam are updated, see **figure 57**. These values will be stored into SRAM memory when **read** is active high, see **figure 55**.

```
176   // low byte memory frame load
177   ////////////////////////////////////////
178      if ( !vid_ldll )
179
180            begin
181            address_cam <= vid_address_low[18:1]; // odd rows
182            data_caml <= video_in;
183            end
184
185         else
186
187            if ( !vid_udll )
188
189            begin
190            address_cam <= vid_address_low[18:1]; // even rows
191            data_camh <= video_in;
192            end
193
194         else
195
196   ////////////////////////////////////////
197   // high byte memory frame load
198   ////////////////////////////////////////
199
200            if ( !vid_ldlh )
201
202            begin
203            address_cam <= vid_address_high[18:1]; // odd rows
204            data_caml <= video_in;
205            end
206
207         else
208
209
210            if ( !vid_udlh )
211
212            begin
213            address_cam <= vid_address_high[18:1]; // even rows
214            data_camh <= video_in;
215            end
```

**Figure 57-camcorder update temporary storage**

Note that since updates to the monitor are running at 25 Mhz and video data from camcorder is being updated at 27 MHz there will be the odd missing pixel but it is not noticeable to the human eye.

Connect up your DE2 as follow;

- Monitor cable to monitor input (15 pin video cable)
- Composite connector to camcorder (RCA Jack)
- Digital connectors from logic analyser should still be connected the same as in part 2

The result should look like **figure 58**

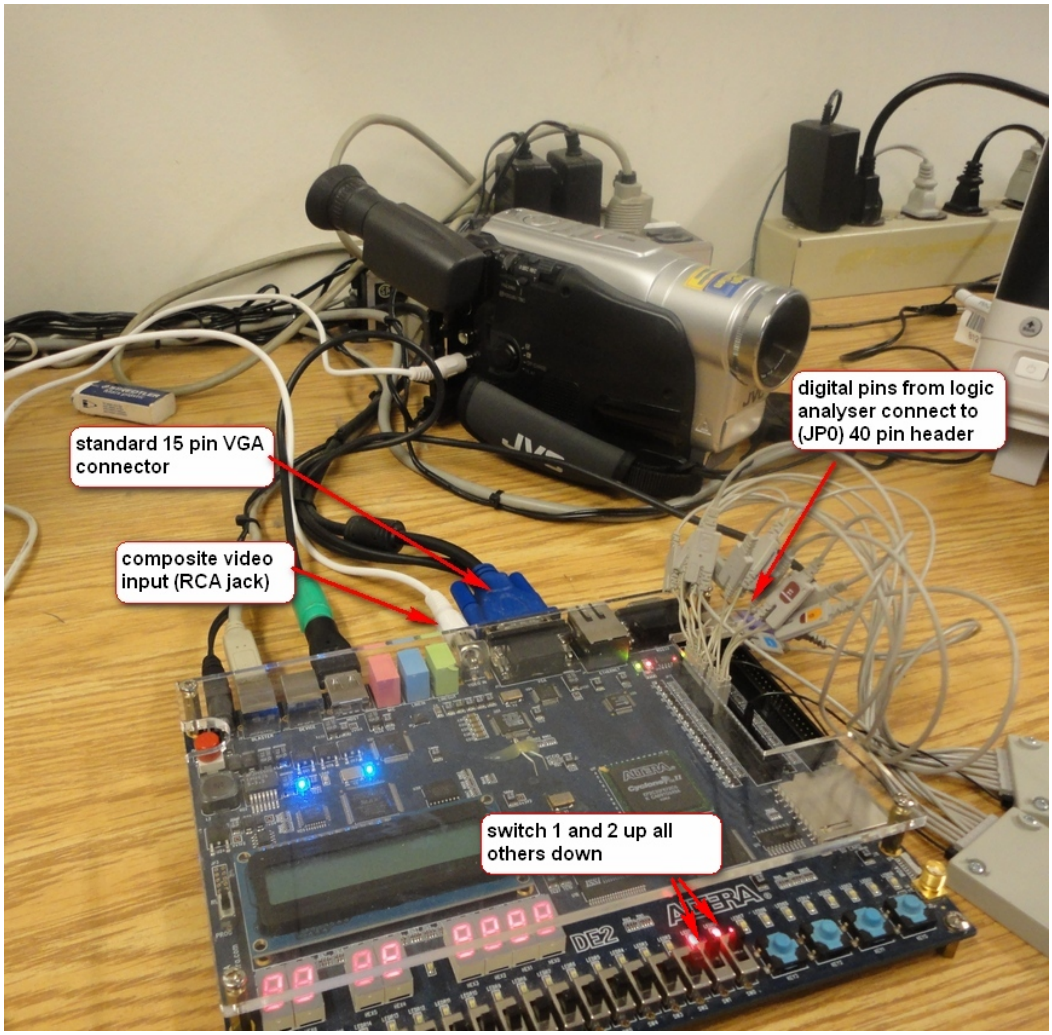


Figure 58-layout video connectors

Down load **the video.sof** file to the DE2 board. Make sure switch **1** and **2** are up and all other switches are down. Video from the camcorder should appear on the monitor connected to the VGA connector. **Figure 59** will give you an example of what the video looks like.


**Figure 59-video sample monitor**

If video does not appear likely you did not download the correct version of the code. In this case try the other zip file. Note that this is a very simple video so there is no depth to the video it is only displaying colours according to shading. We will learn more about how to improve this video later.

Let's examine some of the signals using the MSO scope. Do the following on the MSO scope;

- Press **Digital.**
- Press **Bus** and enable **Bus 1**.
- Set D15-D8 to represent that bus
- Press **back**
- Set scale to be **medium**
- Press **Label** and rename D0-D15 according to **table 15** column 4.

| 40 pin HeaderJP0 | Pin assignment | Digital lead default name | Rename label |
|---|---|---|---|
| GPIO[0] | D25 | D0 | WE |
| GPIO[1] | J22 | D1 | OE |
| GPIO[2] | E26 | D2 | LD |
| GPIO[3] | E25 | D3 | UD |
| GPIO[4] | F24 | D4 | READ |

| | | | |
|---|---|---|---|
| GPIO[5] | F23 | D5 | HSYNC |
| GPIO[6] | J21 | D6 | VSYNC |
| GPIO[7] | J20 | D7 | VID_CLK |
| GPIO[8] | F25 | D8 | Address (bus) |
| GPIO[9] | F26 | D9 | Address (bus) |
| GPIO[10] | N18 | D10 | Address (bus) |
| GPIO[11] | P18 | D11 | Address (bus) |
| GPIO[12] | G23 | D12 | Address (bus) |
| GPIO[13] | G24 | D13 | Address (bus) |
| GPIO[14] | K22 | D14 | Address (bus) |
| GPIO[15] | G25 | D15 | Address (bus) |

**Table 15-part3 pin outs 40 pin header**

- Set delay to **0.0s.**
- Set **horizontal** frequency to **200.0 u/s.**
- Press **Trigger**.
- Select **Edge** trigger.
- Set **VSYNC** as source.
- Set **rising Edge** as slope.
- Press **Single** to trigger event.
- Set delay to **700.0ns.**
- Press **Single** to trigger event again.

The result should look very similar **figure 60**. We can make several observations.

- The **VSYNC** pulse is the end of the previous frame and the beginning of another frame.
- In this trigger event, while **READ** is active high even line video data is being captured from the camcorder and stored into SRAM memory. We can verify this by the fact that **WE** and **UD** are active low for periods of time.
- When **READ** changes to active low data from memory is written to the display. When **READ** returns to active high video data is captured from the camcorder. This can be verified by the fact that both **WE** and **OE** are active.
- Let us zoom in closer to get a better understanding of how the data transfer takes place.


**Figure 60- VSYNC trigger multiple HSYNC cycles**

Do the following:

- Press **Trigger**.
- Change trigger mode from **EDGE** to **EDGE then EDGE**.
- Change **Delay** in **EDGE then EDGE** to **150 ns** (see figure 61).
- Change **Frequency** to **500.0 n/s.**
- Press **Source** in **EDGE then EDGE** menu (see figure 61).
- Change **Arm A** to HSYNC**.**
- Change **Slope A** to rising edge.
- Change **Trigger B** to OE
- Change **Slope B** to falling edge.
- Press **Single** to trigger event.

The result may or may not look similar **figure 61.** This is because we are triggering on **HSYNC** and it can be any one of 240 lines of odd or even video data. Let us make some observations of about this trigger capture.

- The row begins on the rising edge of **HSYNC**.
- In this trigger event **UD** is active high for the whole period, so this is strictly an odd row transfer.

- Prior to **READ** going active low, odd line data is being captured from the camcorder and stored into memory when **WE** and **LD** is active low.
- When **READ**, **OE** and **LD** are active low video data from memory is sent to a pixel location on the display.
- Note that **OE** and **WE** cannot happen at the same time.

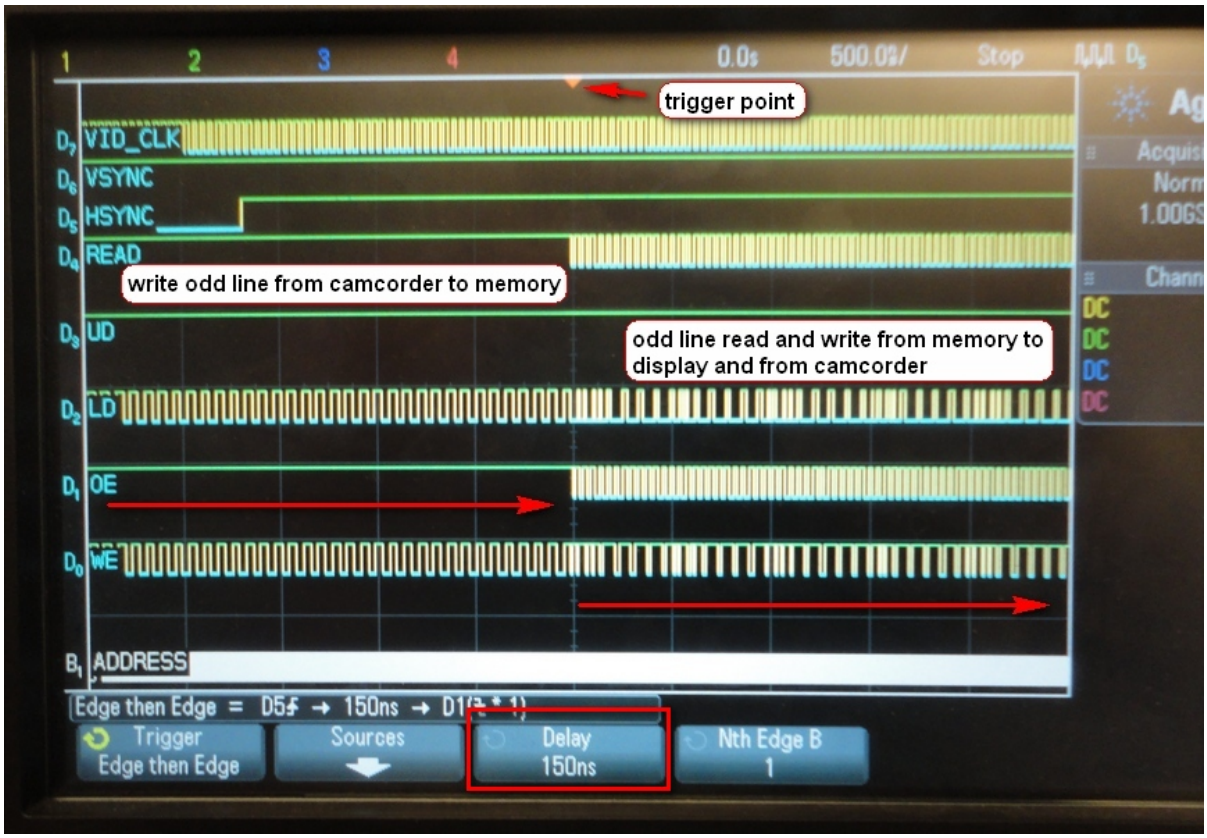


**Figure 61- zoom in OE trigger**

- To verify this we will zoom in further to look at individual cycles.
- Change **Frequency** to **20.00ns**.
- Change **Delay** to **100.0ns**.
- Press **Single** to trigger event.


**Figure 62- read and write cycles**

Here we can see that when **OE** is active low **WE** is active high. They never happen at the same time. Also note that after every **OE** active low transaction the address increments to the next address 60,61,62,63 see **figure 62**. Similarly when **WE** has transitioned it also increments to the next address location, 88, 89,8A see **figure 62**. If in the case of **address** 89 appearing twice, this means that the video data has not been updated since the previous 25 MHz clock cycle. This makes sense since the camera.v state machine is independently updating as is the monitor.v state machine. This concludes this tutorial. You should now be familiar with how to update video in real time.

As was explained earlier the video we just generated is very simply video. To improve on this some important preliminary information needs to be understood. To get a better understanding about video and how video is displayed on a monitor follow these link;

http://en.wikipedia.org/wiki/YCbCr

http://www-ug.eecg.utoronto.ca/msl/manuals/Video.pdf

Read starting at page 6 - background on video.

The DE2 board uses YCrCb 4:4:4 format. See **figure 63** below.

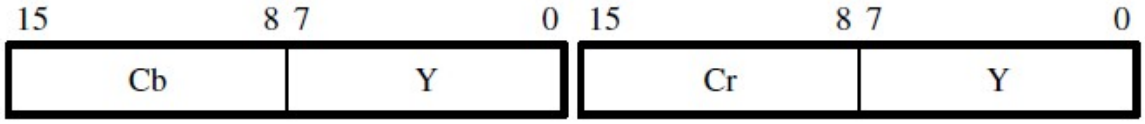


**Figure 63-YCrCb video format**

This means that the Cr and Cb components are updated every pixel. This format is defined as 8 bits per colour and full colour plane. To see how this video improves what is displayed on the monitor download the following zip file.

http://www-ug.eecg.utoronto.ca/desl

(older versions DE2 board, serial number starting with **"0"**)

**Select**-DE2>DESL Online Tutorials> ycrcb_d13.zip.

Or

(new versions of the DE2 board, serial number starting with **"1"** and video decoder is **ADV7181B** )

**Select**-DE2>DESL Online Tutorials>ycrcb_c16.zip.

Or

(new versions of the DE2 board, serial number starting with **"1"** and video decoder is **ADV7180** )

**Select**-DE2>DESL Online Tutorials>ycrcb_c16_v3.zip.

These are fully designed projects from the Terasic website.

As you will notice these designs have a much more realistic full colour video.

This concludes the Tutorial.

All enquiries should be emailed to aulich@ece.utoronto.ca