Interfaces on the DE1-SoC board

 Thanks you to Mike (Mehrdad) Mehramiz for his help developing this manual.

Sept 2016 Version 5.4

The following tutorials are meant to help the user become more familiar with the different interfaces on the Altera DE1-SoC board. We will be concentrating on the FPGA interfaces.

The objective of each tutorial is to learn and understand how each interface works and use 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 DE1-SoC board uses the Cyclone V 5CSEMA5F31C. For more information on this FPGA, go to the following link:

https://www.altera.com/products/fpga/cyclone-series/cyclone-v/overview.html

There are 5 different interfaces on the DE1-SOC board [FPGA portion].

1. Composite Video Input – connects to a camcorder with composite video output
2. VGA Video Output – connects to a VGA monitor
3. PS2 Interface
4. Audio Interface – mic input , line input, speaker output
5. 40 pin general purpose ports – GPIO-0 (JP1), GPIO-1(JP2)

Figure 1 shows where all the interfaces are located.

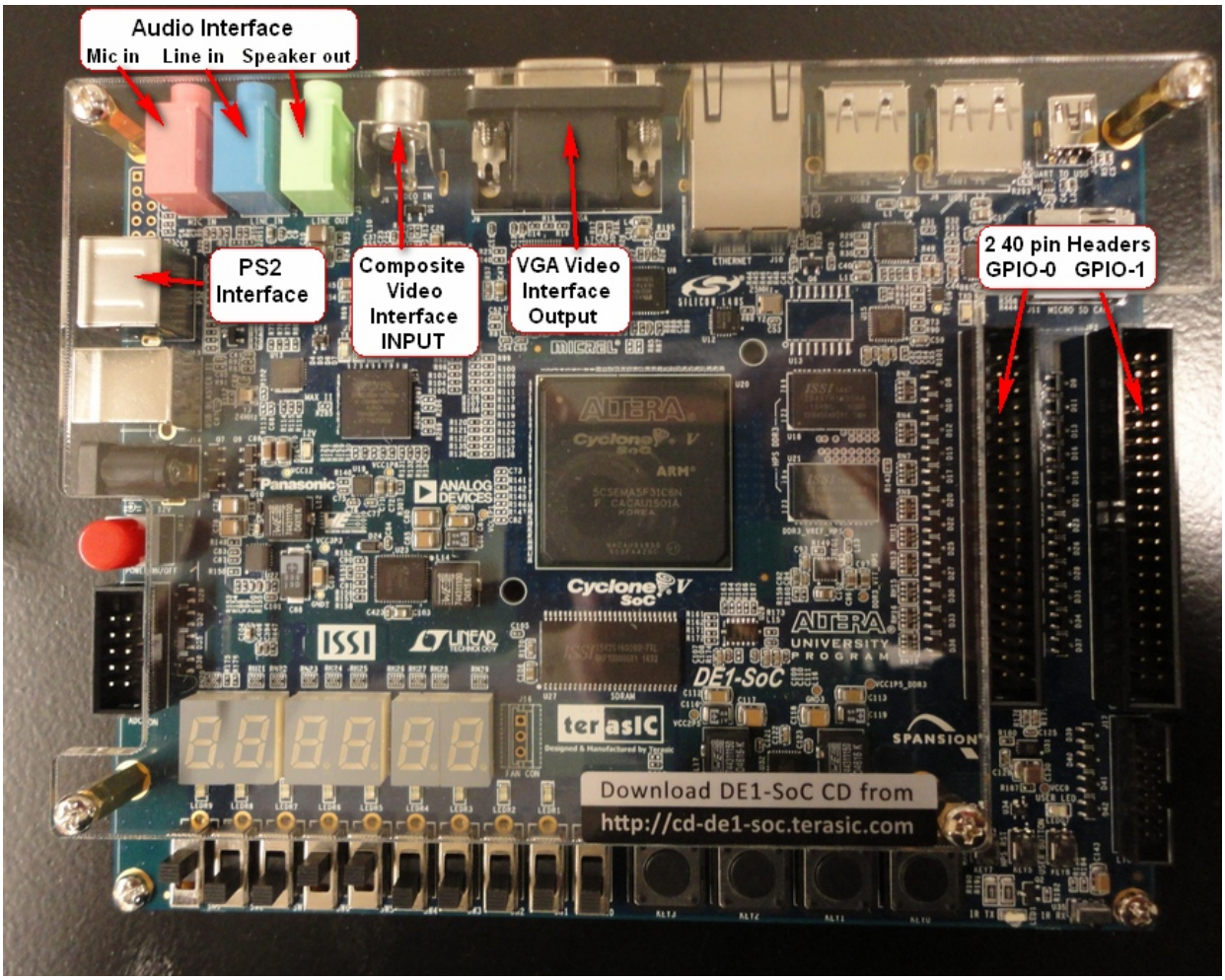


Figure 1- Location of interfaces.

Let us examine each of the interfaces individually.

Composite Video Interface

The chip used on the DE1-SoC board is an Analog Devices ADV7180. For a description of this interface follow the link below:

http://www-ug.eecg.utoronto.ca/desl/manuals/ADV7180.pdf

Figure 2 is a block diagram of how the interface is connected to the FPGA on the DE1-SoC board. Video is collected from a composite video source, such as a camcorder, with a composite video RCA jack output.

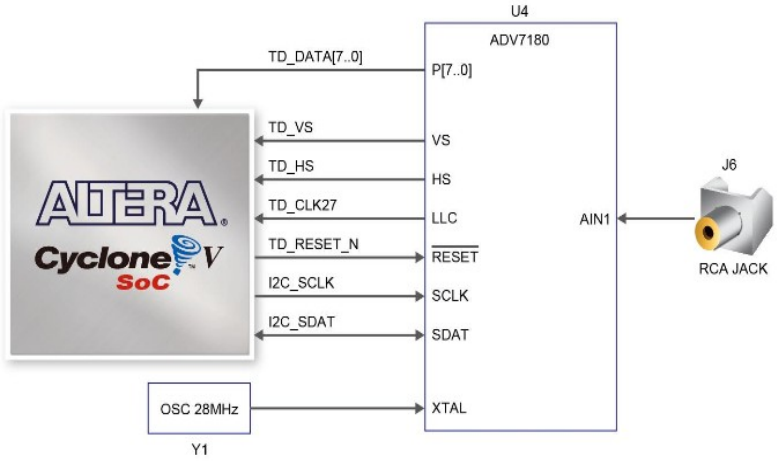


Figure 2-Block diagram of the DE1-SoC FPGA to the video decoder

Table 1 is the pin assignments

Signal Name	FPGA Pin No.	Description
TD_Data[0]	Pin_D2	TV Decoder Data[0]
TD_Data[1]	Pin_B1	TV Decoder Data[1]
TD_Data[2]	Pin_E2	TV Decoder Data[2]
TD_Data[3]	Pin_B2	TV Decoder Data[3]
TD_Data[4]	Pin_P1	TV Decoder Data[4]
TD_Data[5]	Pin_E1	TV Decoder Data[5]
TD_Data[6]	Pin_C2	TV Decoder Data[6]
TD_Data[7]	Pin_B3	TV Decoder Data[7]
TD_HS	Pin_A5	TV Decoder H_SYNC
TD_VS	Pin_A3	TV Decoder V_SYNC
TD_CLK27	Pin_H15	TV Decoder Clock Input
TD_RESET	Pin_F6	TV Decoder Reset
TD_SCIk	Pin_J12	I2C Clock
TD_SDAT	Pin K12	I2C Data

Table 1- Pin assignments on DE1-SoC 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 generated by the ADV7180 video decoder chip. Pin is assigned according to **Table 1**.
- [TV Decoder V_SYNC]- Vertical sync pulse generated by the ADV7180 video decoder chip. Pin is assigned according to **Table 1.**

- [TV Decoder Clock input]- This is a 27 MHZ clock generated by the ADV7180 video chip. In order to enable the clock, the TD_RESET pin must be asserted to an active high logic level.
- I2C Data- This is a bi-directional serial data bus pin, use to program the internal serial register of the ADV7180 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 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 ADV7180 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 pulses and pixel pulses between each vertical pulse.

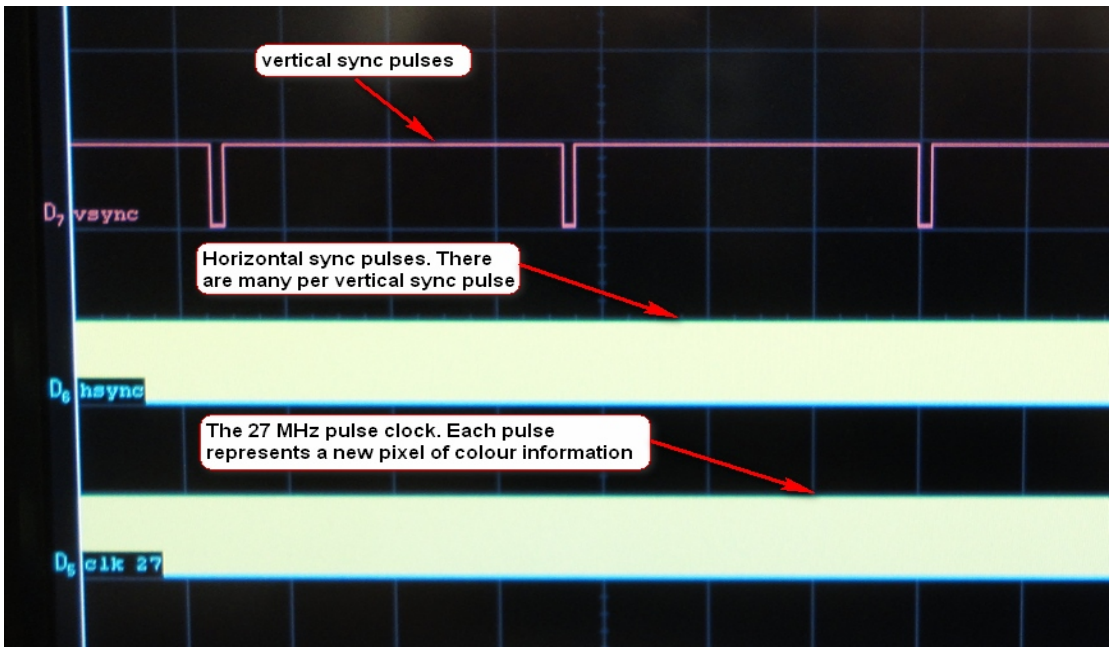


Figure 3-Video decoder output signals (vertical, horizontal and clock pulses)

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

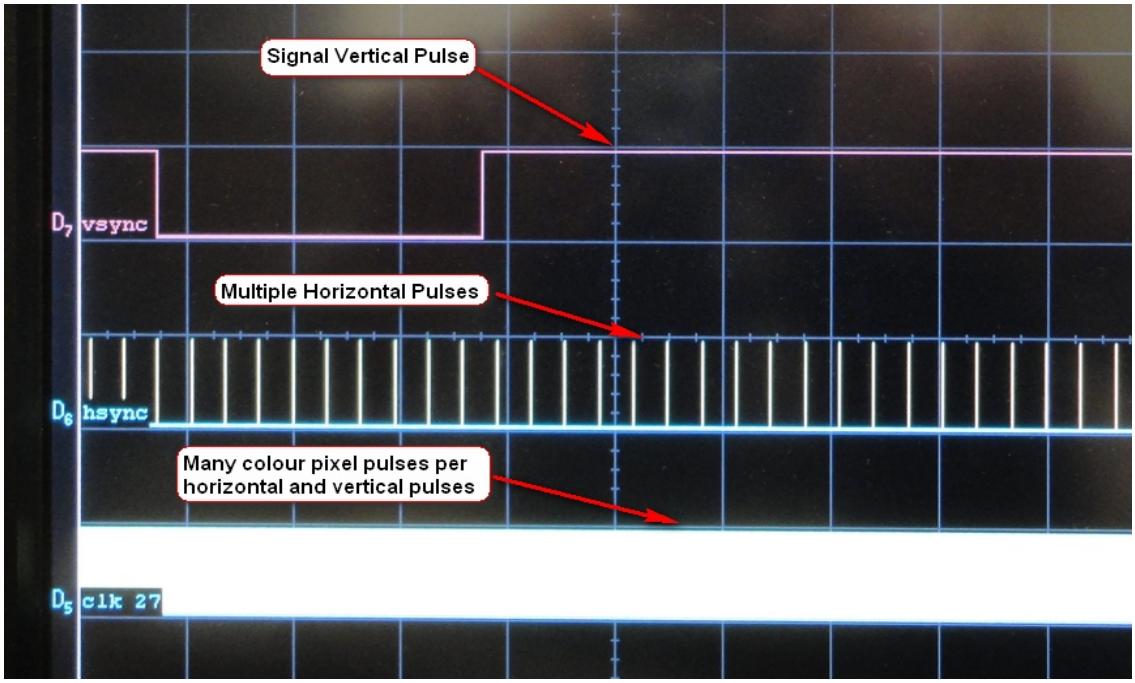


Figure 4-Zoomed in view of horizontal pulses

Figure 5 shows the beginning of a video decoder frame

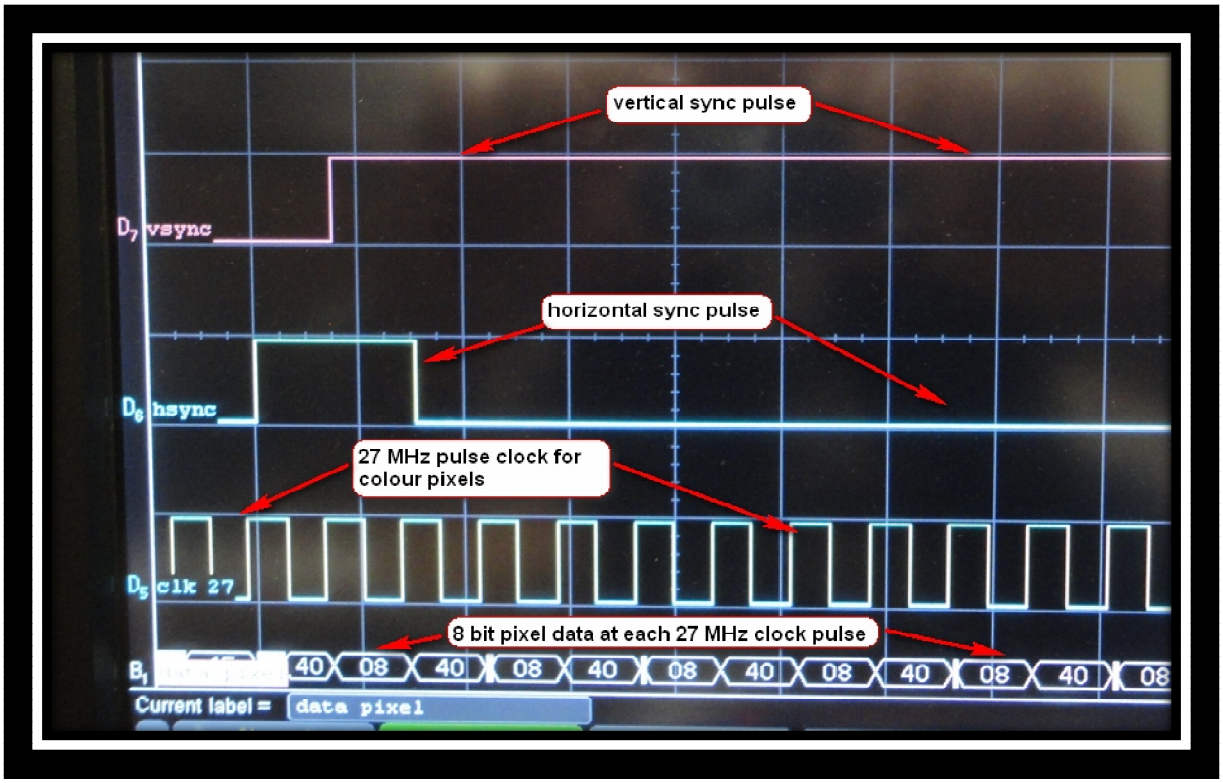


Figure 5-Zoomed in view of pixel pulses

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

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

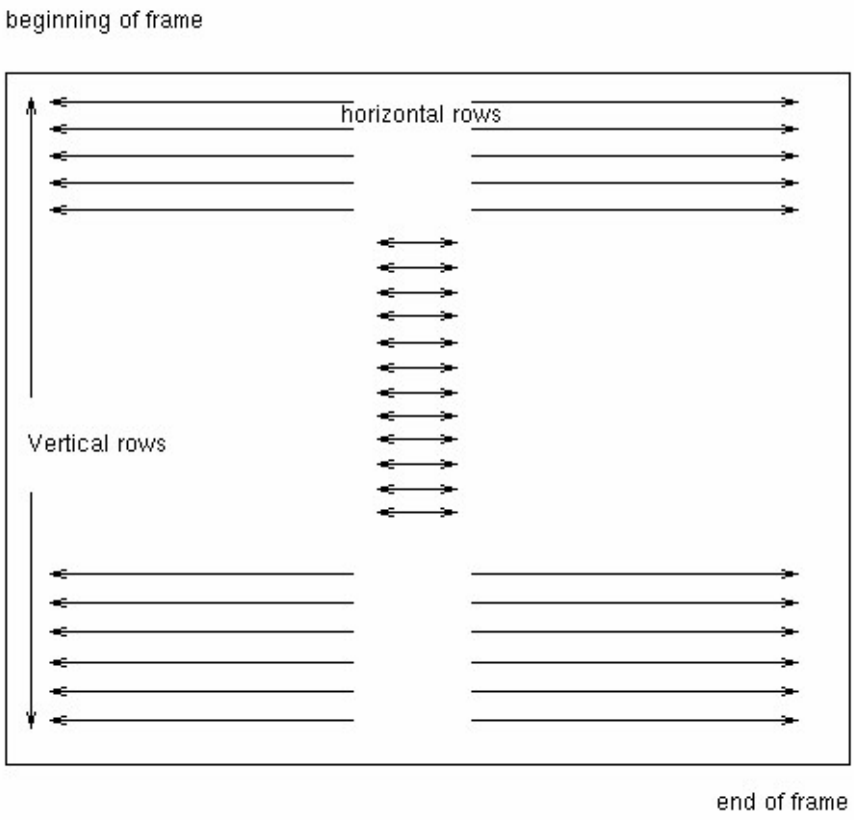


Figure 6- block view of a video frame

As an example say each horizontal row has **640** pixels and there are **480** vertical rows. This means the frame size is **640X480 = 307200**. This is important to know because it tells you how much memory is needed to save one frame of video data. 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 video is received by the ADV7180 decoder is NTSC. This standard was 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

This format is slowing being phased out in place of the more common Digital Video format. However the DE1-SoC still uses the NTSC format. The key features of this format are:

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

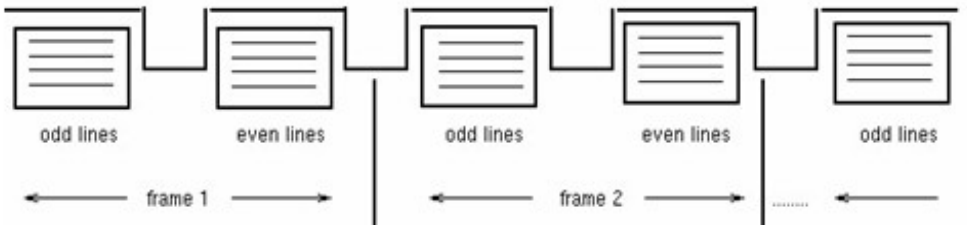


Figure 7- Interlacing frames

I2C- Inter-Integrated Circuit

This is a very common serial protocol used in industry to serially program internal registers. The DE1-SOC board has two devices that use this protocol: the video decoder chip ADV7180 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 ADV7180	Writes data to the internal video registers
HEX 0X41	Video chip ADV7180	Reads data from the internal video registers
HEX 0X34	Audio chip WM8731	Writes data to the internal audio registers
HEX 0X35	Audio chip WM8731	Reads data from the internal audio registers

Table 2 Address values for read and write commands

Figure 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 column 2 of **Table 1**. The serial clock and serial data will have to be generated using Verilog code. The FPGA is generating the clock and data signals, so it is called the master device. The video encoder and the audio CODEC are called the slave devices because they are receiving the serial data and serial clock signals.

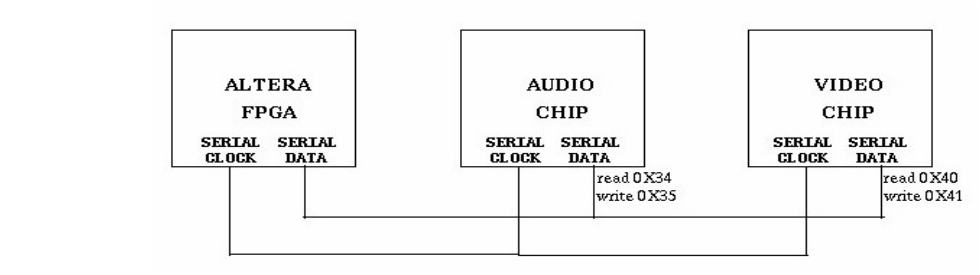


Figure 8-Interface of I2C devices

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 to the slave device properly. At the end of the transfer the slave device sends an acknowledge pulse to indicate that it properly received the serial data.

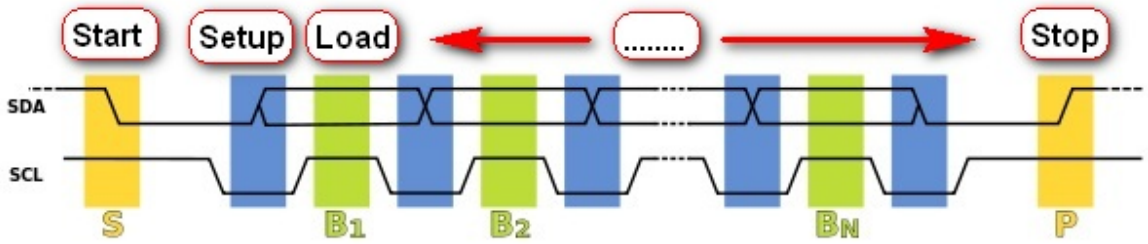


Figure 9- timing for serial data and clock

- **Start**- represents the beginning of the serial load cycle. The data line transitions from high to low and then shortly after, the clock line transitions from high to low.
- **Setup**- represents the period where the data bits become valid.
- **Load** – The serial clock transitions from low to high. At that time the data value on the data bus is accepted by the internal data register of the slave device. This continues 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. First the clock value goes higher and then one cycle after that, the data line goes higher. Once this is complete another load can occur using the same procedure described above.

Figure 10 shows an example of a single load to the video encoder (ADV7180) chip on the DE1-SoC board.

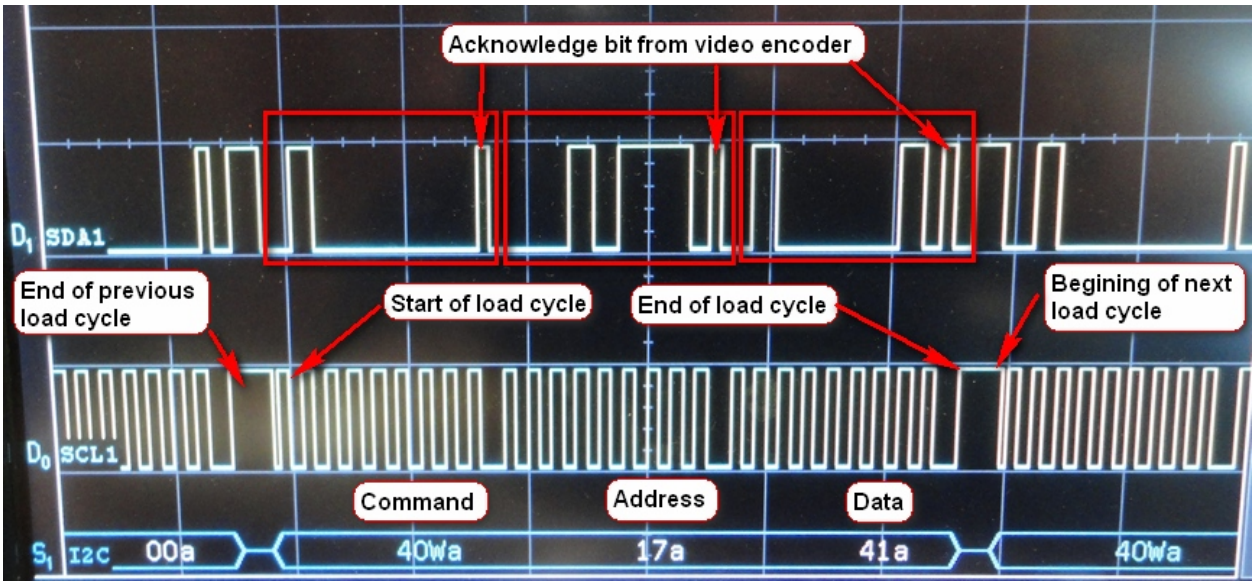


Figure 10 – Single serial load I2C protocol

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].

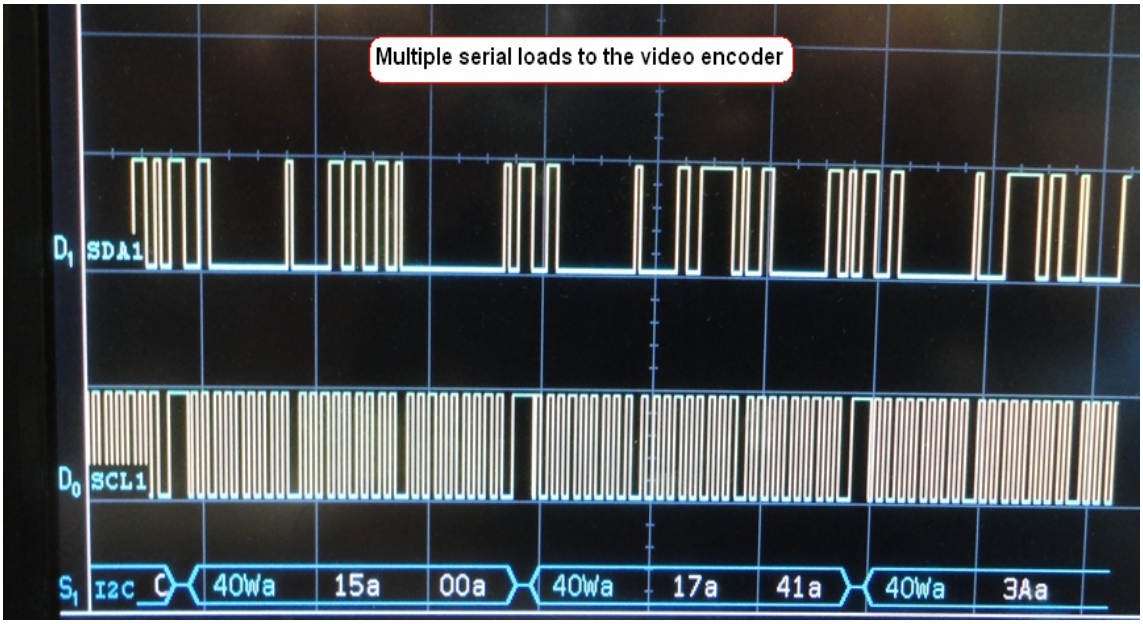


Figure 11- Multiple loads of I2C protocol

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:

- Altera CAD package Quartus. Use version 15.0 or greater.
- Verilog programming language. If you are not familiar with Verilog go to the following link for tutorials and examples:

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

  Select NIOS II (DE1-SoC)>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:

  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 ADV7180 that will be needed for this tutorial.

TD_CLK27	Pin_H15	TD_Decoder Clock Input
TD_Reset	Pin_F6	TD_Decoder Reset
I2C_SCLK	Pin_J12	I2C Clock
I2C_Data	Pin_K12	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 DE1-SOC 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 10[table2] of the ADV7180 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.

   Download the following Verilog code:

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

   Select- DE1-SoC>DESL Online Tutorials>clock.generator.v

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

   Select- DE1-SoC>DESL Online Tutorials>clock_generator.qsf

Create a new project called **clock_generator** using the **new project wizard** in Quartus (version 15 or greater). Once the Verilog code has been compiled and downloaded to the DE1-SoC board, connect the digital leads of the MSO-X-3024A as shown in **Figure 12.**

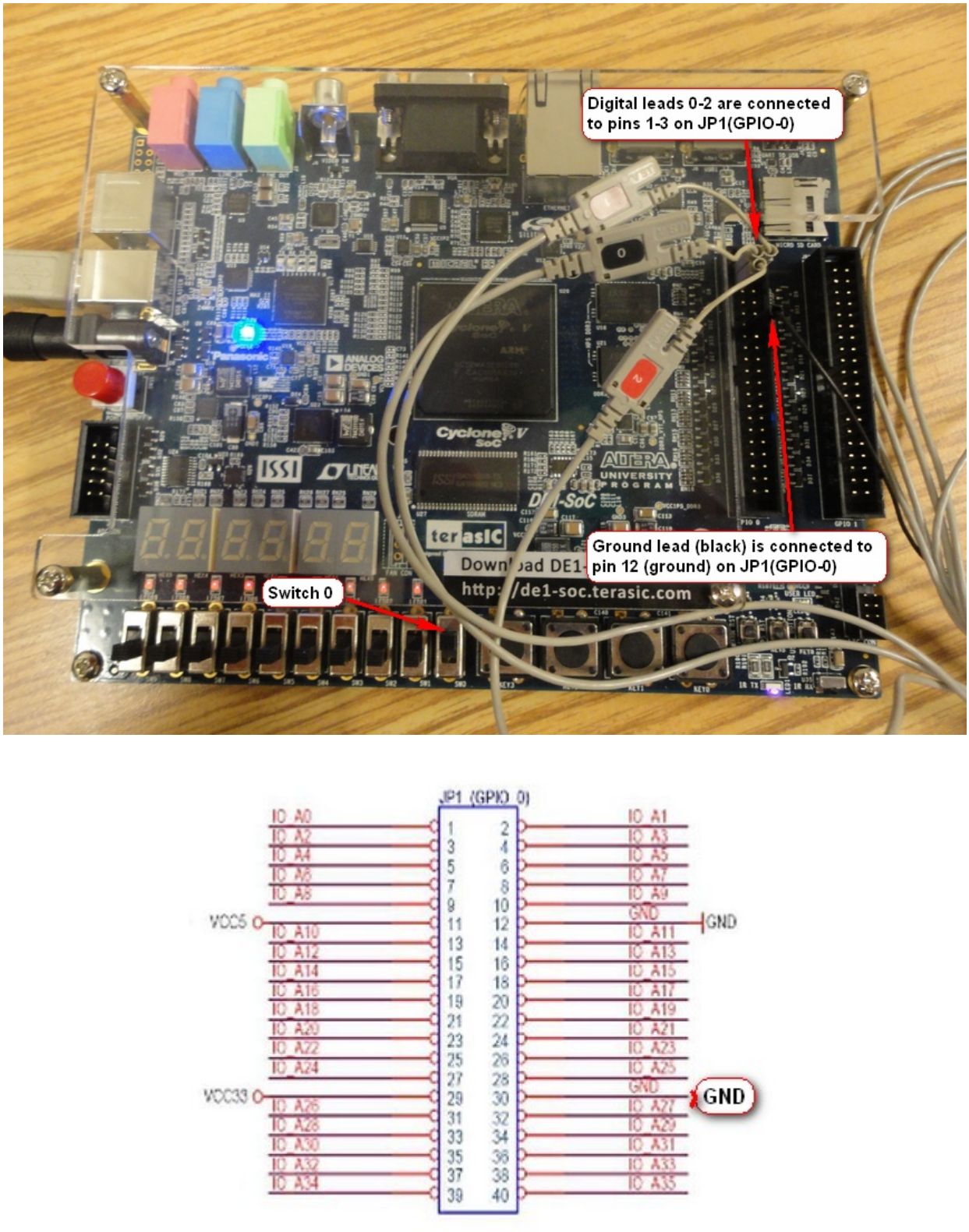


Figure 12 - Connecting digital leads to GPIO-0 on the DE1-SOC board.

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 DE1-SOC 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. This will enable the 27MHz clock, which will generate the 40 KHz clock created by the Verilog code.

- Press **Trigger**.
- Set **SCLK** to trigger source.
- Set **Slope** to rising edge (low to high transition).
- Set **Delay** is **0.0**s
- Set **Frequency** to **5.000**u/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**.

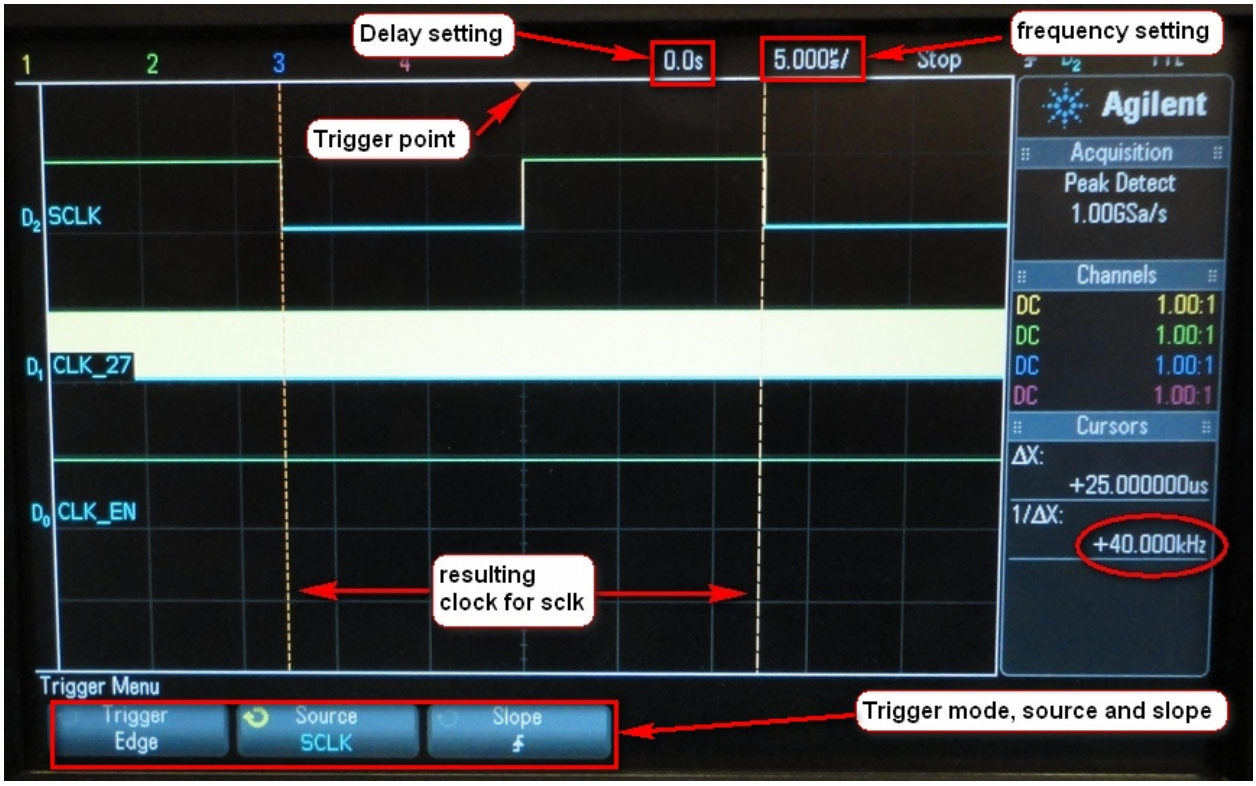


Figure 13 - Using cursors X1 & X2 to evaluate delta clock frequency

- Press the **Zoom** button in the horizontal section of the MSO-X-3024A.
- Set the **Delay** to **200** n/s

The result will look like **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-X-3024A 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 need to be accomplished:

1. Create an 8 bit register to store the HEX data values. There will be three registers:

   One for the command, one for the address and one for the data.

2. Using switches on the DE1-SoC board to generate and select the HEX data values.

3. Use the HEX displays on the DE1-SoC 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.

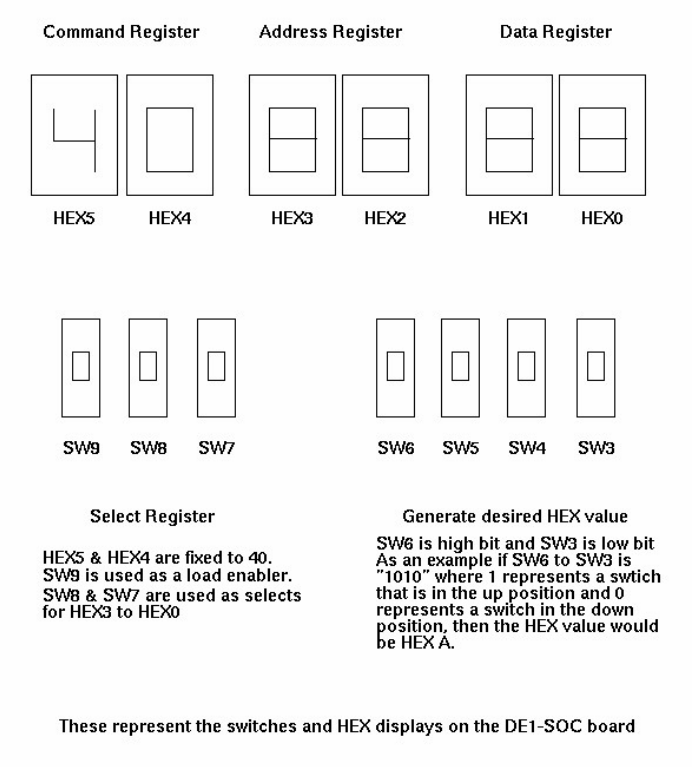


Figure 15 –Truth table for HEX decoder

SW9 up equals load enable and down load disabled	SW8	SW7	SW6 down to SW3	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.

Download the following Verilog code:

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

Select- DE1-SoC>DESL Online Tutorials>HEX_decoder.v

For pin assignments **Select**- DE1-SoC>DESL Online Tutorials >HEX_decoder.qsf

Create a new project called **HEX_decoder** using the **new project wizard** in Quartus. Compile the Verilog code and downloaded it to the DE1-SOC board.

Figure 16 shows an example of what the output looks like. Note HEX5 and HEX4 are fixed to 40. HEX3 to HEX0 can have their HEX values changed and stored according to the settings of switches SW9 to SW7. SW0 is used as a circuit enable.

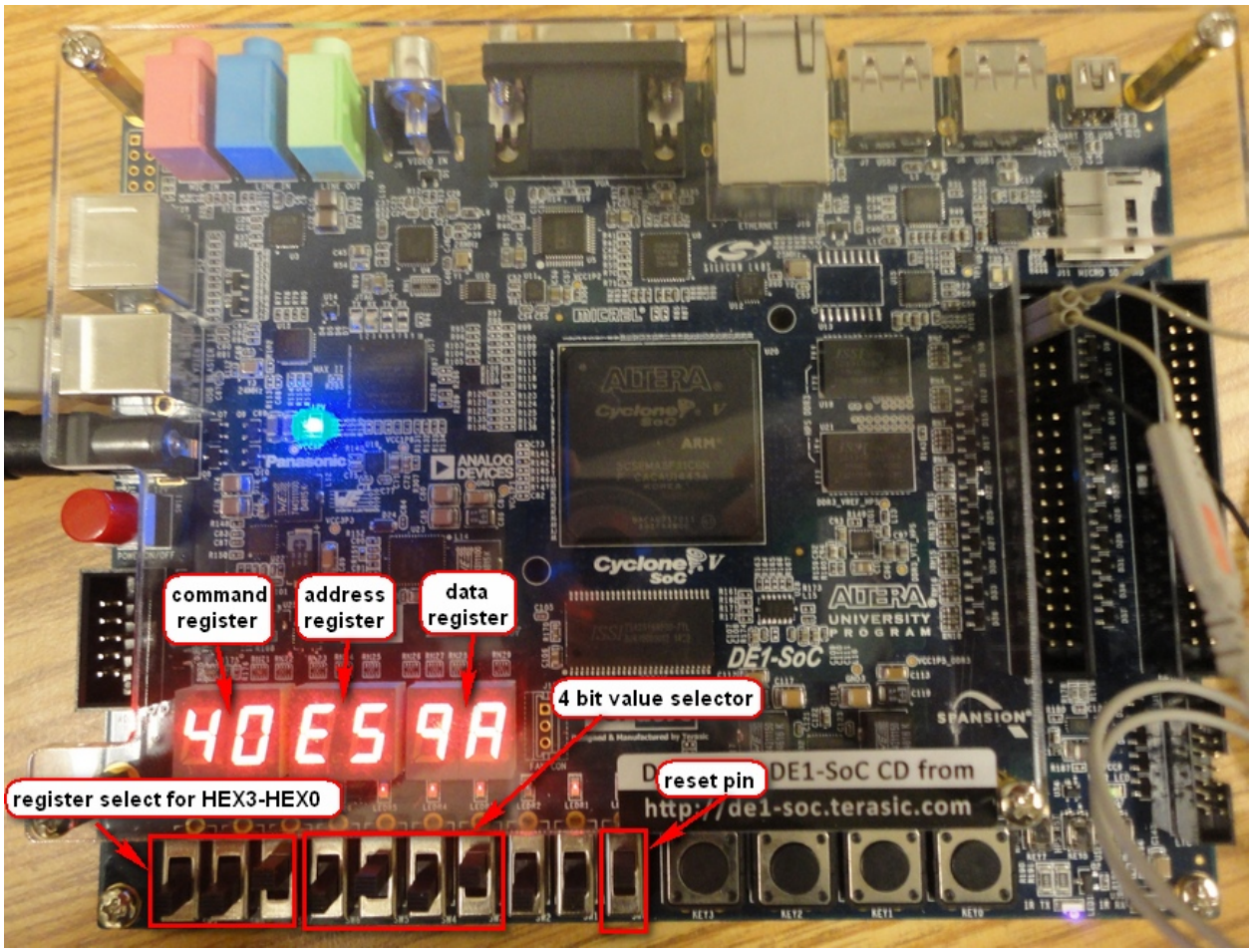


Figure 16 – HEX data values displayed on the DE1-SoC board

This concludes part 2.

Part 3 - Generate the SCLK and SDATA signals to serially shift clocked data into the video decoder.

The key tasks for part 3 are:

1. Generate a start pulse
2. Generate an 8 bit value for command, address and data.
3. Generate SCLK pulses.
4. Continue until the data is sent
5. Wait for acknowledgement from the 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.

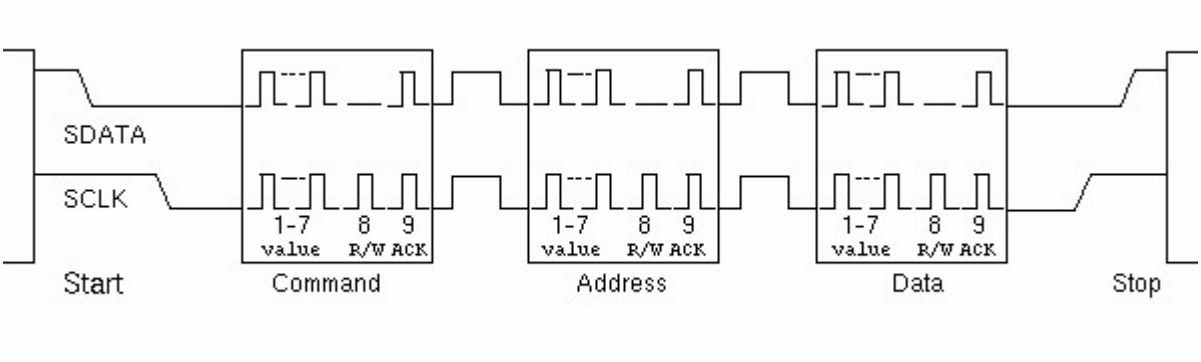


Figure 17 – Block diagram of the serial I2C data transfer

Now that we have all of the information, we can write the Verilog code.

Download the following Verilog code:

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

Select-DE1-SoC>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. Use **sdata_sclk** as the project name. Once the Verilog code has been compiled, download it to the DE1-SOC board,

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

Connect the analog probe from the MSO-X-3024A to the DE1-SOC GPIO-0 40 pin header.

Get 2 **probe pin leads** from the 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**.

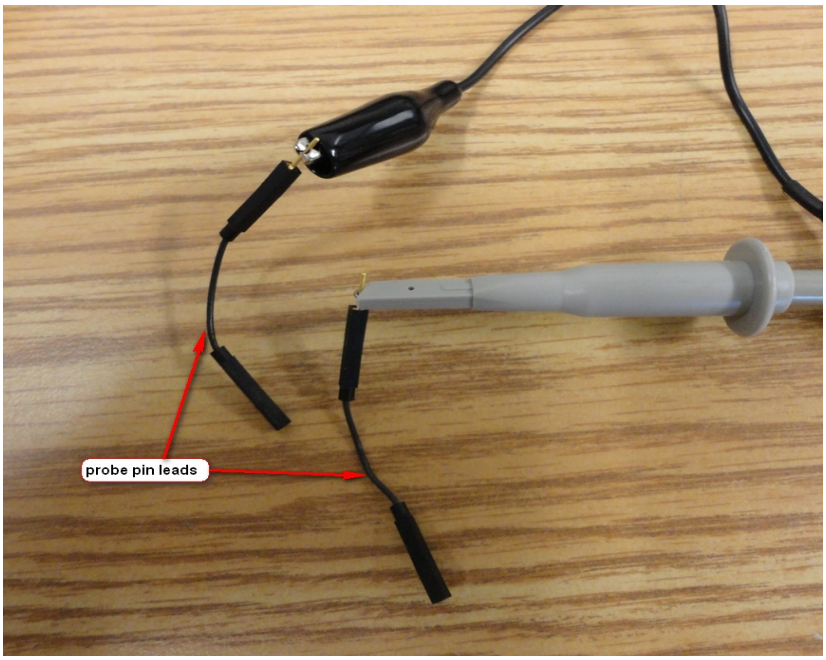


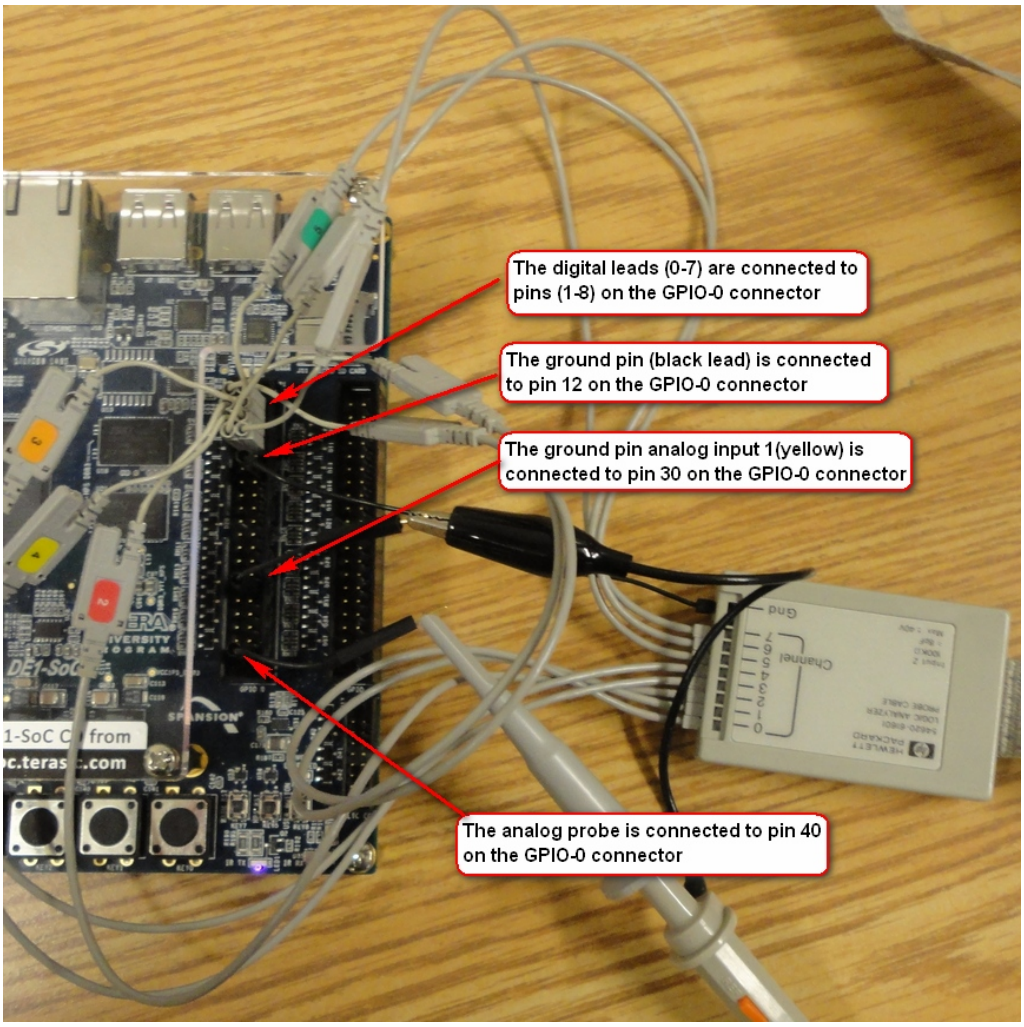
Figure 18 – Description of how to connect the probe pin leads to the analog probe

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

Now connect the analog probe as described in **Table 6**. Use **Figure 19** as a further reference**.**

Name of GPIO-0 pin	Location on FPGA	Location on GPIO -0 40 pin header	Probe or Digital lead connection on MSO
GPIO[0]	AC18	1 (IO A0)	Digital Lead 0
GPIO[1]	Y17	2 (IO A1)	Digital Lead 1
GPIO[2]	AD17	3 (IO A2)	Digital Lead 2
GPIO[3]	Y18	4 (IO A3)	Digital Lead 3
GPIO[4]	AK18	5 (IO A4)	Digital Lead 4
GPIO[35]	AJ21	40 (IO A35)	Analog Probe 1

Table 6- Pin assignments for GPIO 0 Tutorial 1 part 3



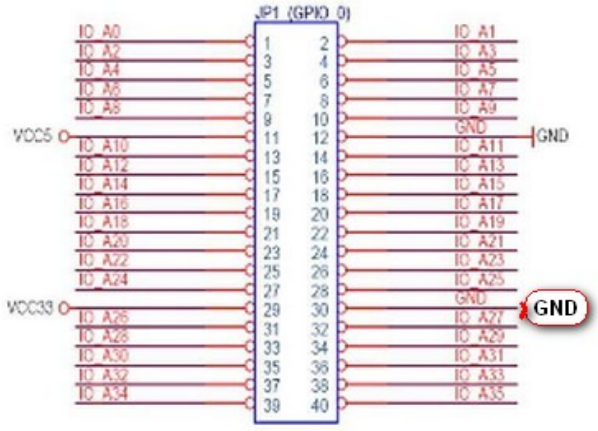


Figure 19 – Digital and analog connections to the GPIO-0 header

- 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 the 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 large.
- Set the delay from the 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 the down to up position.

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

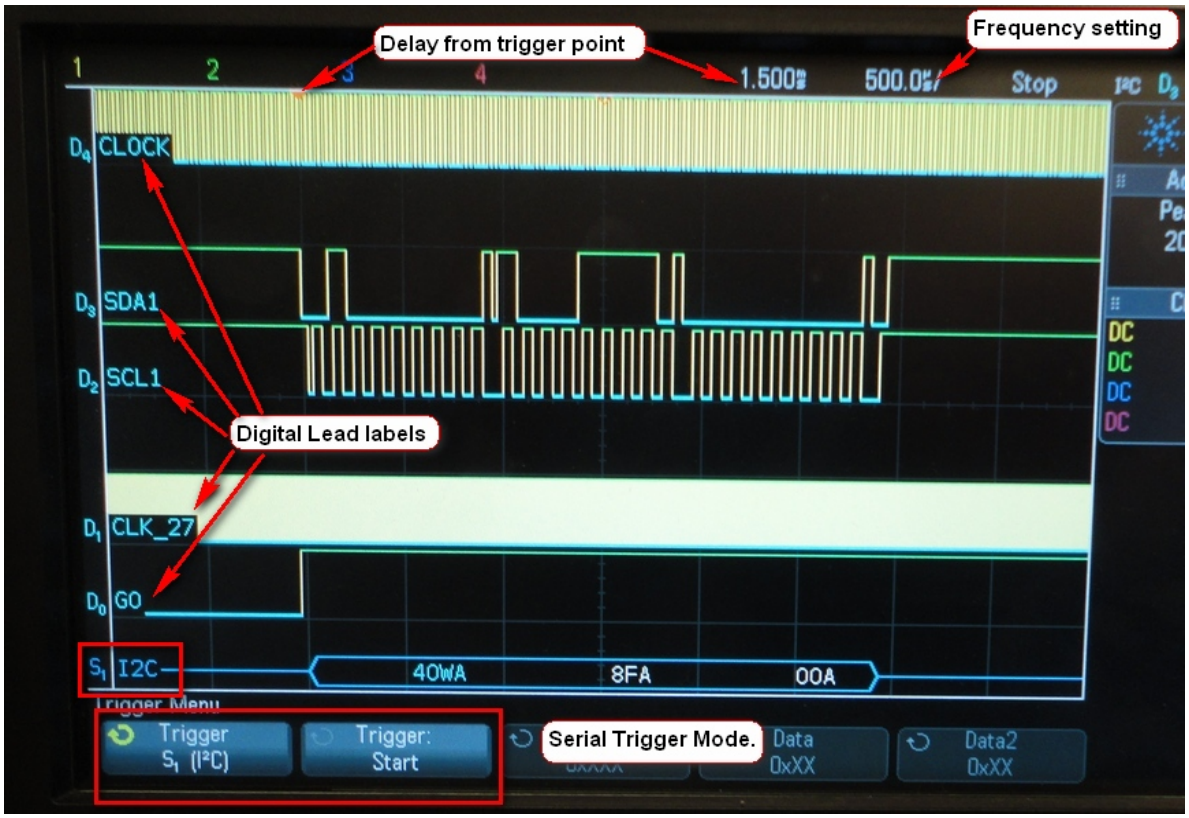


Figure 20 – The locations of the delay trigger and frequency setting

From part 2 (HEX decoder), Switches 9 - 3 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 of the 62 ADV7180 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 **ADV7180** data sheet gives a description of the address value and the different data options available. The 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 from 13.5 MHz to 27. 0 MHz. 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 pin will be 27 MHz. This happens to be the default value at reset.

- Make sure **switch 1** is up and **switch 2** is down.
- Press the **Digital** button and deselect digital channels **D0** and **D1**.
- Move **D2** (SCL1), **D3** (SDA1) and **D4** (CLOCK) down on the scope. See **Figure 21**.
- Connect the analog probe to the input 1 (yellow) on the MSO-X-3024A.
- 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 as **analog_27.**

- Set the voltage level for analog channel 1 to **5.00** V.
- Set the **Delay** to **1.630**m/s**.**
- Set the **Frequency** to **500.0**u/s.
- Press **Single** on the run section of the MSO-X-3024A.
- Toggle **switch 2** (GO) from a down to an up position.

The result should look like F**igure 21**. **Analog_27** represents the 27 MHz clock. By zooming in and using the cursors you can measure the frequency to verify if this is true or not. This is optional.

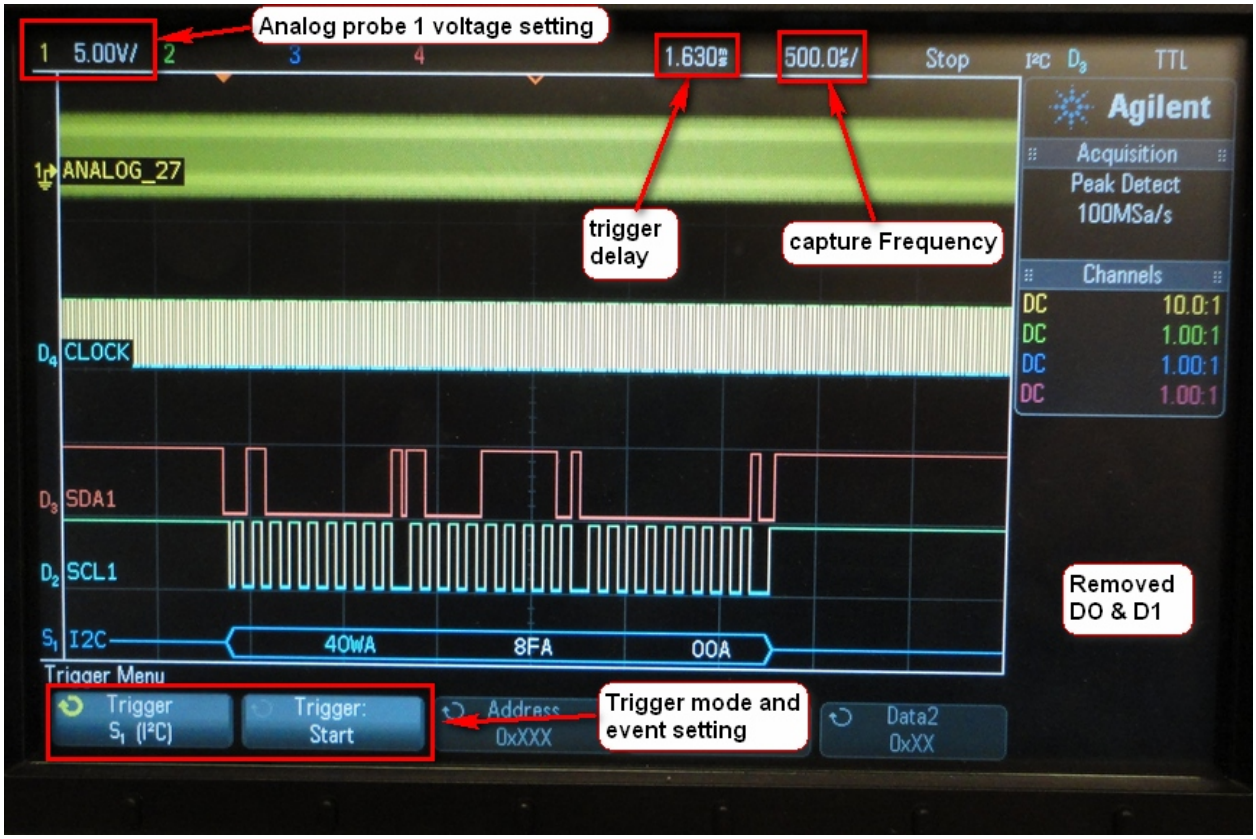


Figure 21 – Analog probe 1 input added and D0 and D1 removed.

Change the data value from HEX 00 to HEX 50 using switches 9 to 3. 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.

- 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 a high position.

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

Note that where the data value HEX 50 occurs, the frequency changes.

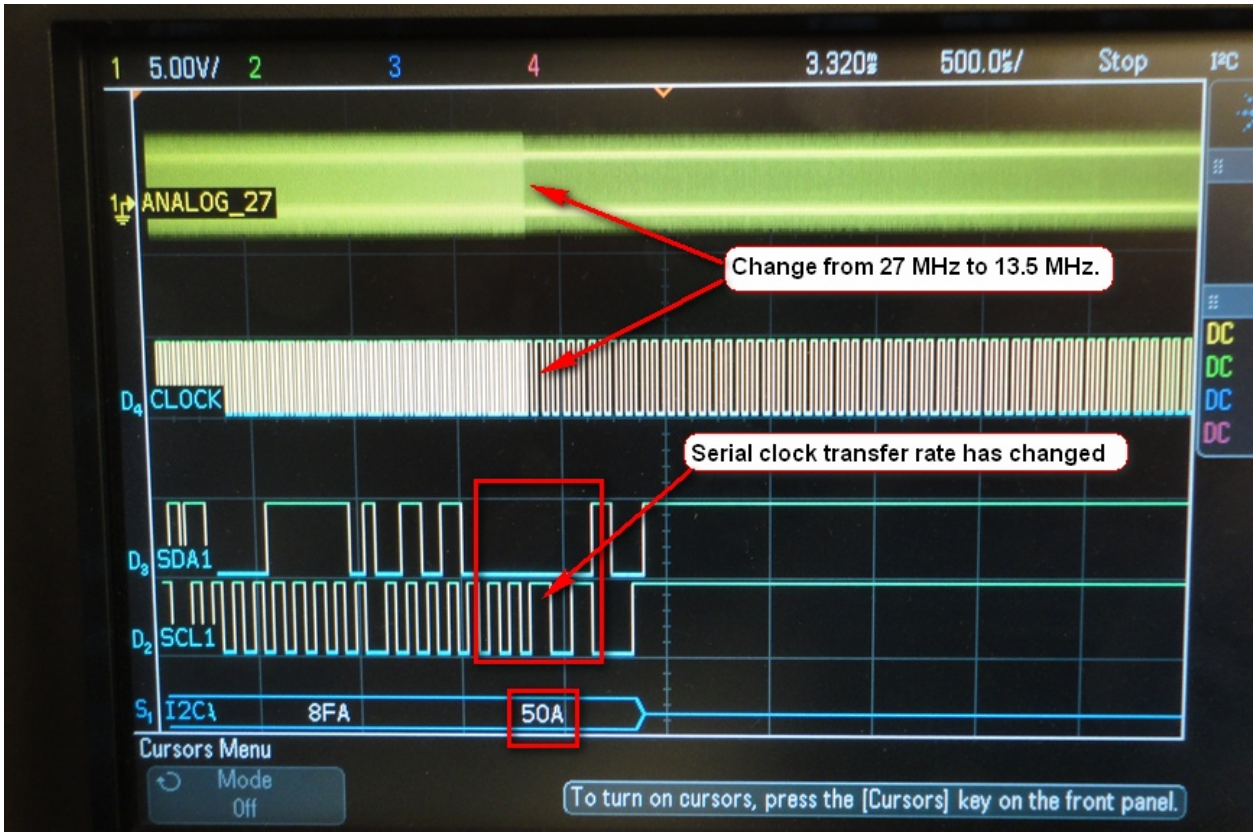


Figure 22 – clk_27 frequency changed from 27 MHz to 13.5 MHz

Also note that the serial transfer rate has 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 the 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 a high position.
- Press **Cursors**.
- Using **X1** and **X2** measure the delta time of the **D4 CLOCK**.

The result should be that the delta value is **20.000 KHz**. See **Figure 23.**

Figure 23 – Using cursors to check the delta frequency

Previously it was 40 KHz.

Now change the data value from HEX 50 to HEX 00 using switch 9 - 3.

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

The result should be the same as **Figure 24.**

Figure 24–The default frequency changes from 13.5 MHz to 27 MHz

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

The result should be a delta value of **40.000 KHZ**. See **Figure 25**.

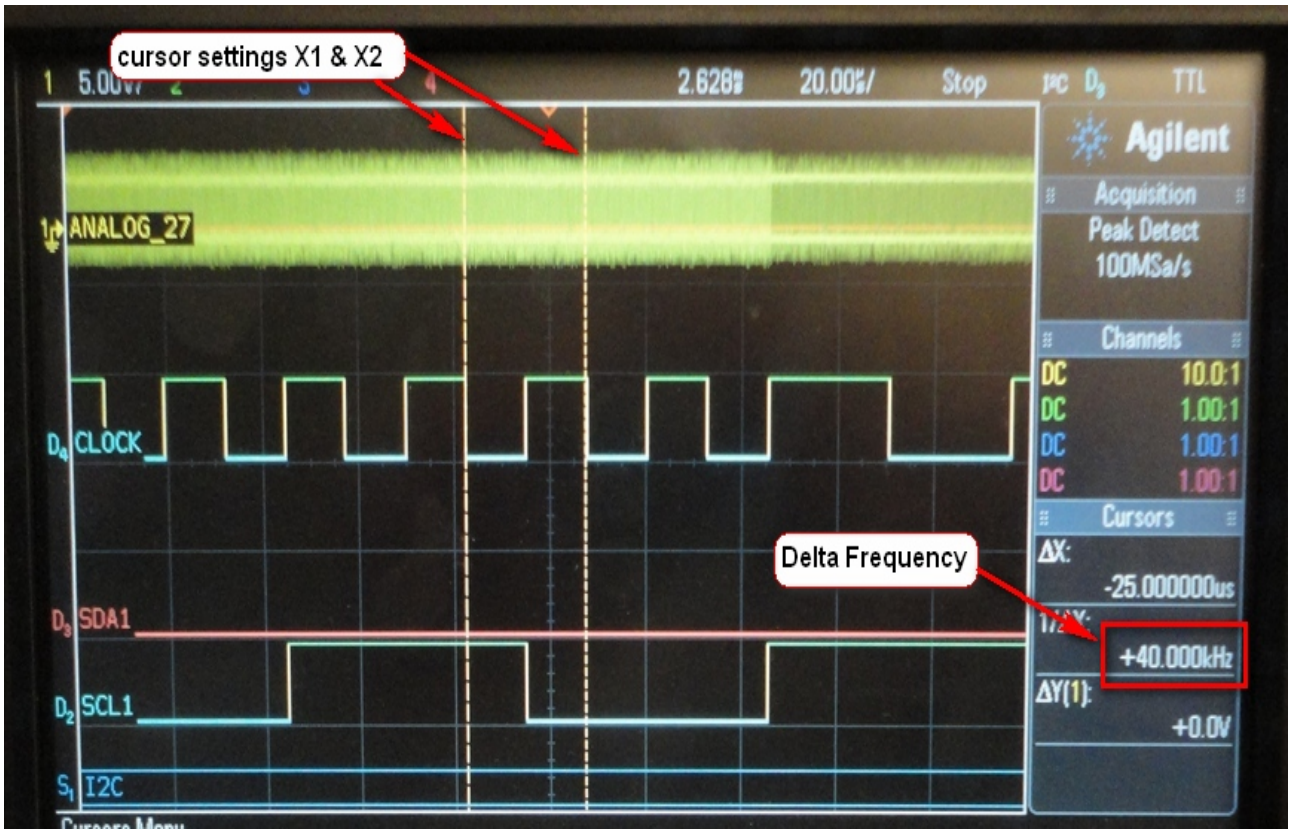


Figure 25 – The delta frequency for the default CLK_27 output

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

Capturing video from a composite camcorder.

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 DE1-SOC board. For this tutorial we will use the internal on chip SRAM. The maximum size of the SRAM chip is 1,400 Kbytes. This will be reduced depending on the size of your design.

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

Method 1- is known as substitution. Here 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. See **Figure26**.

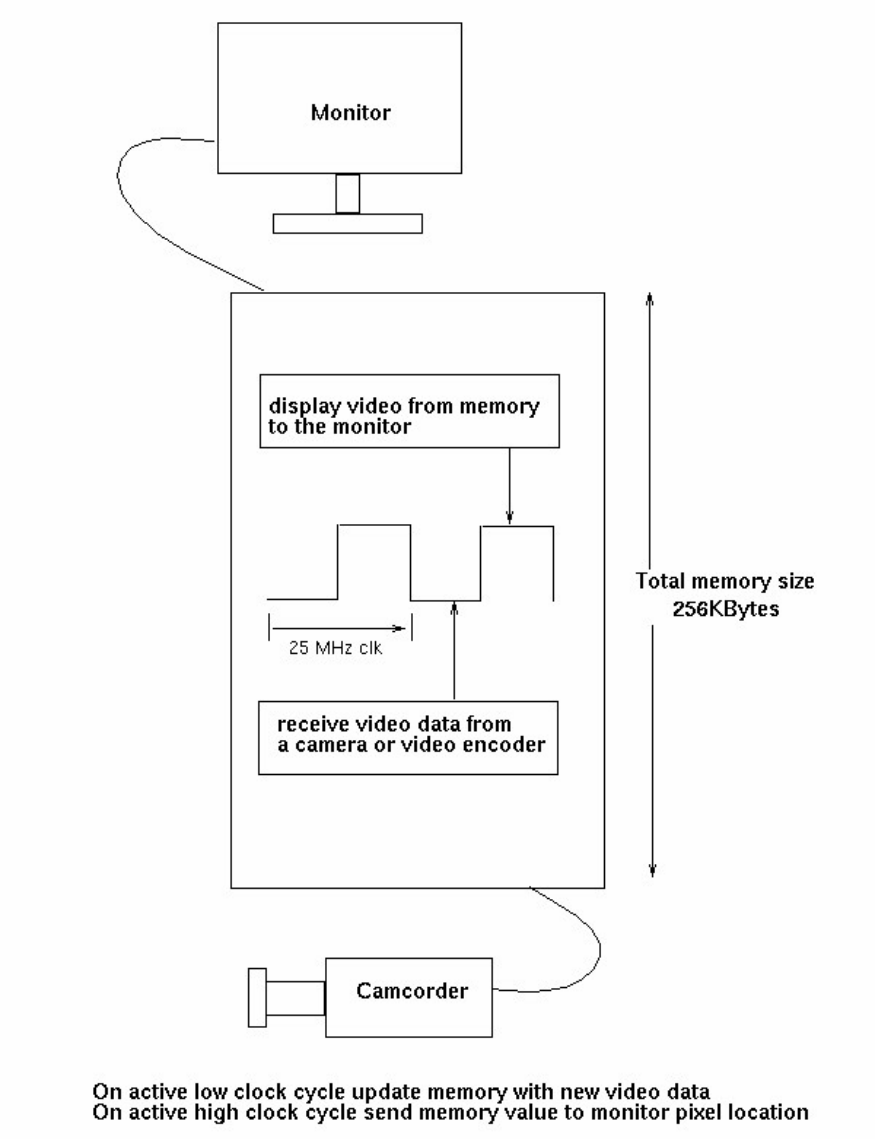


Figure 26 – Block diagram of the Substitution method

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 the video pixel data to the monitor. At the end of the frame update the roles are reversed. See **Figure 27** for further explanation. You need at least two times the frame memory size for this method to work.

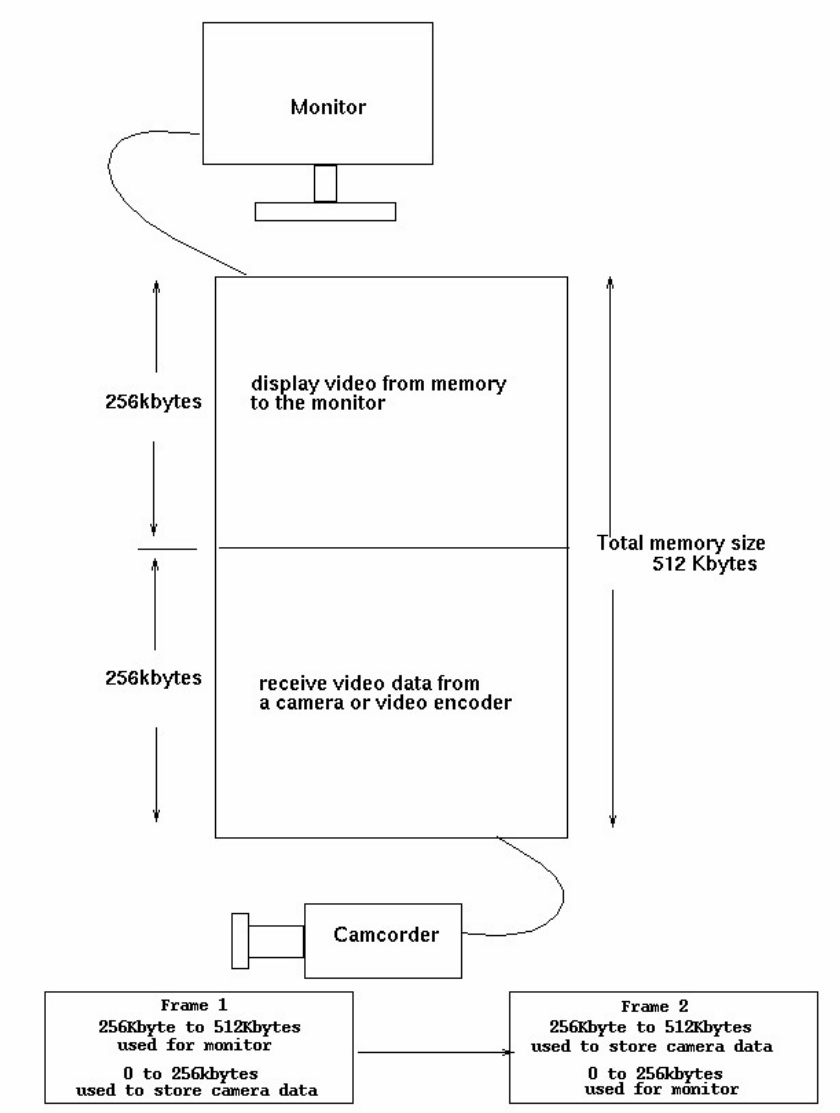


Figure 27 – Block diagram of the split memory

Tutorial 2-Capturing video data from the camcorder.

For the purpose of this tutorial we will use method 1, the substitution method. 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 (ADV7180).

The video decoder chip (ADV7180) 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 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, with the odd video line in one frame and the even video line in the second frame. So we require two frames to make a full frame, in order to be displayed on to a monitor.

Now we can make the following observations:

1. With each 27 MHz clock pulse an 8 bit video pixel will be stored in SRAM memory.
2. Hsync will represent each line of video data. Each line is 624X8 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. One full frame requires 2 Mbits. This is within the memory capacity of the on chip memory SRAM on the DE1-SOC board.

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 DE1-SOC on chip SRAM memory. The following counters, latches and enablers will need to be created:

- Address counter for the odd and even frames.
- Create on chip SRAM memory chip (2Meg). The ideal SRAM configuration is 16 address lines X 32 data bits which gives us 2^{16}X32= 2 Mbits of memory.

- Video line counter for each line (horizontal)
- Video row counter to keep track of each row (vertical)
- Data latches for pixel data for both odd and even data 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 -DESL Online Tutorials>Tutorial2.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.**

```
210   ////////////////////////////////////////
211   /// logic analyser test bit locations ///
212   /// on GPIO JP0 40 in header          ///
213   ////////////////////////////////////////
214
215
216      assign gpio[0] = vid_ldl;
217      assign gpio[1] = vid_ldh;
218      assign gpio[2] = vid_udl;
219      assign gpio[3] = vid_udh;
220      assign gpio[4] = frame;
221      assign gpio[5] = vert;
222      assign gpio[6] = horiz;
223      assign gpio[7] = vid_hs;
224
225      assign gpio[15:8] = address_cam[7:0];
```

Figure 29 - Test pins connected to the 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-X-3024A analyser.

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

	tatu	From	To	Assignment Name	Value	Enabled
1	✓		gpio[15]	Location	PIN_AG17	Yes
2	✓		gpio[14]	Location	PIN_AF16	Yes
3	✓		gpio[13]	Location	PIN_AE16	Yes
4	✓		gpio[12]	Location	PIN_AG16	Yes
5	✓		gpio[11]	Location	PIN_AH17	Yes
6	✓		gpio[10]	Location	PIN_AH18	Yes
7	✓		gpio[9]	Location	PIN_AJ16	Yes
8	✓		gpio[8]	Location	PIN_AJ17	Yes
9	✓		gpio[7]	Location	PIN_AJ19	Yes
10	✓		gpio[6]	Location	PIN_AK19	Yes
11	✓		gpio[5]	Location	PIN_AK18	Yes
12	✓		gpio[4]	Location	PIN_AK16	Yes
13	✓		gpio[3]	Location	PIN_Y18	Yes
14	✓		gpio[2]	Location	PIN_AD17	Yes
15	✓		gpio[1]	Location	PIN_Y17	Yes
16	✓		gpio[0]	Location	PIN_AC18	Yes

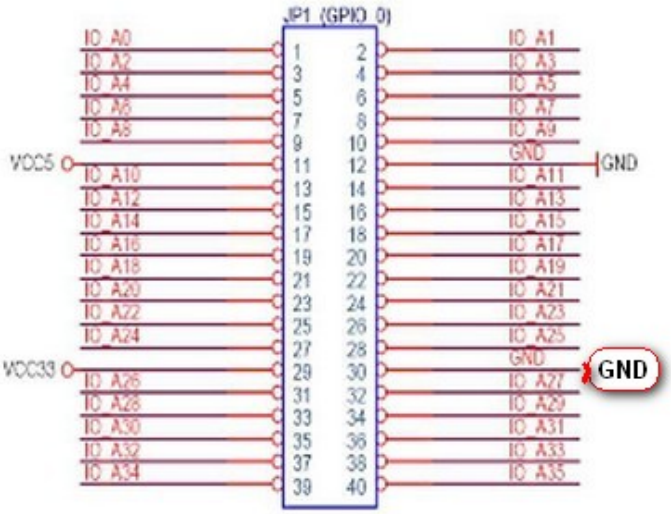


Figure 30 - Pin assignments GPIO on the Port GPIO-0

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]	AC18	VID_LDL	Digital Lead 0
GPIO[1]	Y17	VID_LDH	Digital Lead 1
GPIO[2]	AD17	VID_UDL	Digital Lead 2
GPIO[3]	Y18	VID_UDH	Digital Lead 3
GPIO[4]	AK16	FRAME	Digital Lead 4
GPIO[5]	AK18	VERT	Digital Lead 5
GPIO[6]	AK19	HORIZ	Digital Lead 6
GPIO[7]	AJ19	VID_HS	Digital Lead 7
GPIO[8]	AJ17	Address[0]	Digital Lead 8

GPIO[9]	AJ16	Address[1]	Digital Lead 9
GPIO[10]	AH18	Address[2]	Digital Lead 10
GPIO[11]	AH17	Address[3]	Digital Lead 11
GPIO[12]	AG16	Address[4]	Digital Lead 12
GPIO[13]	AE16	Address[5]	Digital Lead 13
GPIO[14]	AF16	Address[6]	Digital Lead 14
GPIO[15]	AG17	Address[7]	Digital Lead 15

Table 9 - Connections from the DE1-SOC GPIO-0 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 MSO-X-3024A digital logic pins to the DE1-SOC board. Using columns 1 and 4 of **Table 9** connect the digital logic pins to the GPIO-0 40 pin header. The result should look like **Figure 31.**

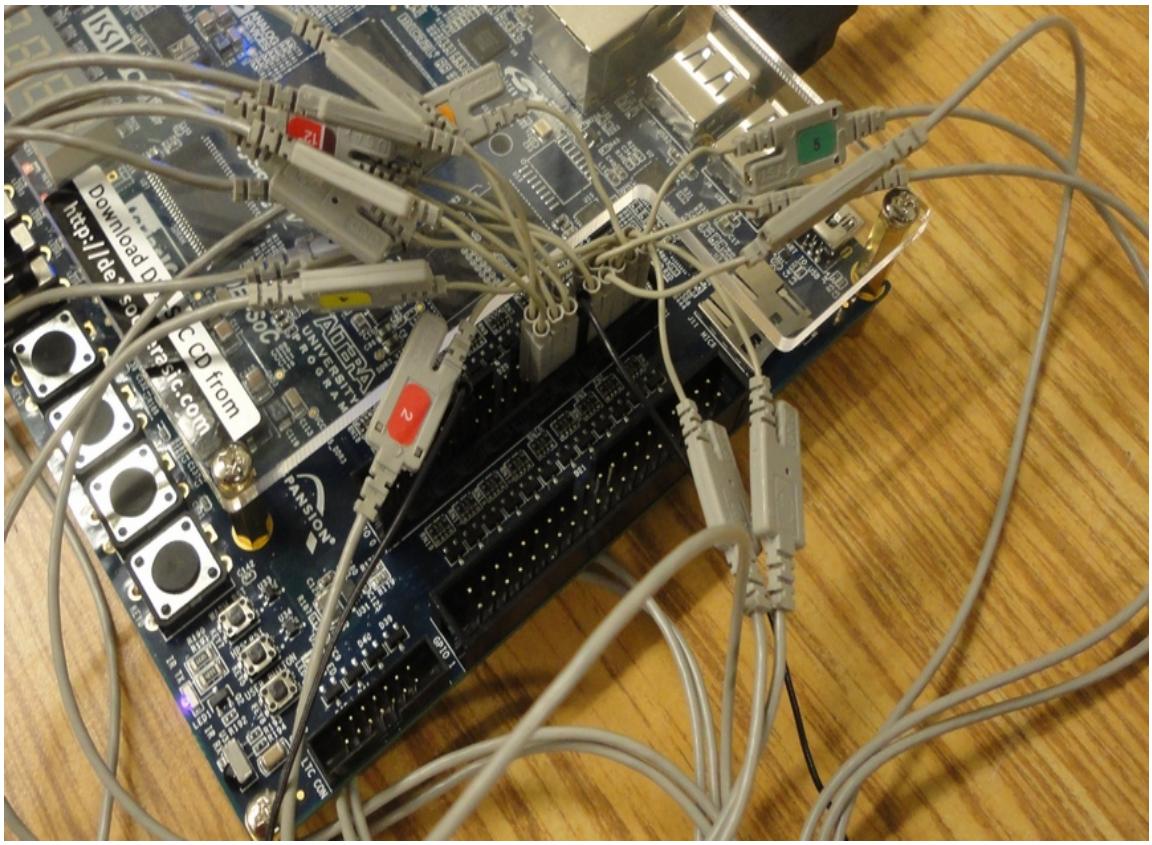


Figure 31- How to connect the digital logic probe pins from the analyzer to the DE1-SOC 40 pin header GPIO-0

Now power up the DE1-SOC board and download camera.sof, which should have been created when the project was compiled.

Switch 0 and **switch 1** on the DE1-SOC 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 DE1-SOC board. See **Figure 32**.

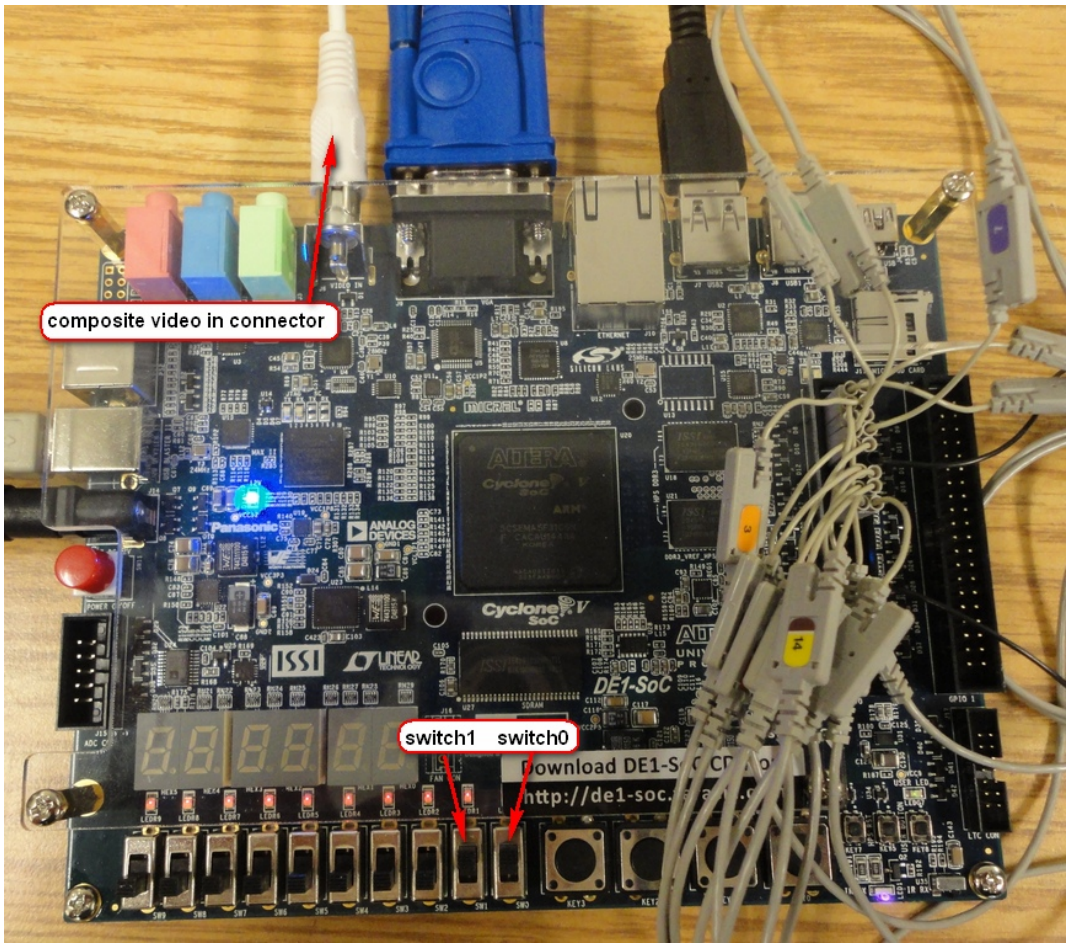


Figure 32 - Switch setting on the DE1-SOC board

Next we need to set up the logic analyzer:

- Press **Default Setting** on the logic analyzer.
- Press **Digital.**
- Turn **D15-D8** off and Turn **D7-D0** on.
- Press **Label.**
- Label D0 to D7 as in **Table 10.**

Digital lead default name	Rename label
D0	VID_LDL
D1	VID_LDH
D2	VID_UDL
D3	VID_UDH
D4	FRAME
D5	VERT
D6	HORIZ

D7	VID_HS

Table 10 -Label for the digital logic pins on the analyzer

- The result should look like the “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 **0.0**s. See **Figure 33**.
- Set **Horizontal** frequency to **5.000**m/s**.** See **Figure 33**.
- Press **Digital.**
- Set **Scale** to medium. See **Figure 33**.

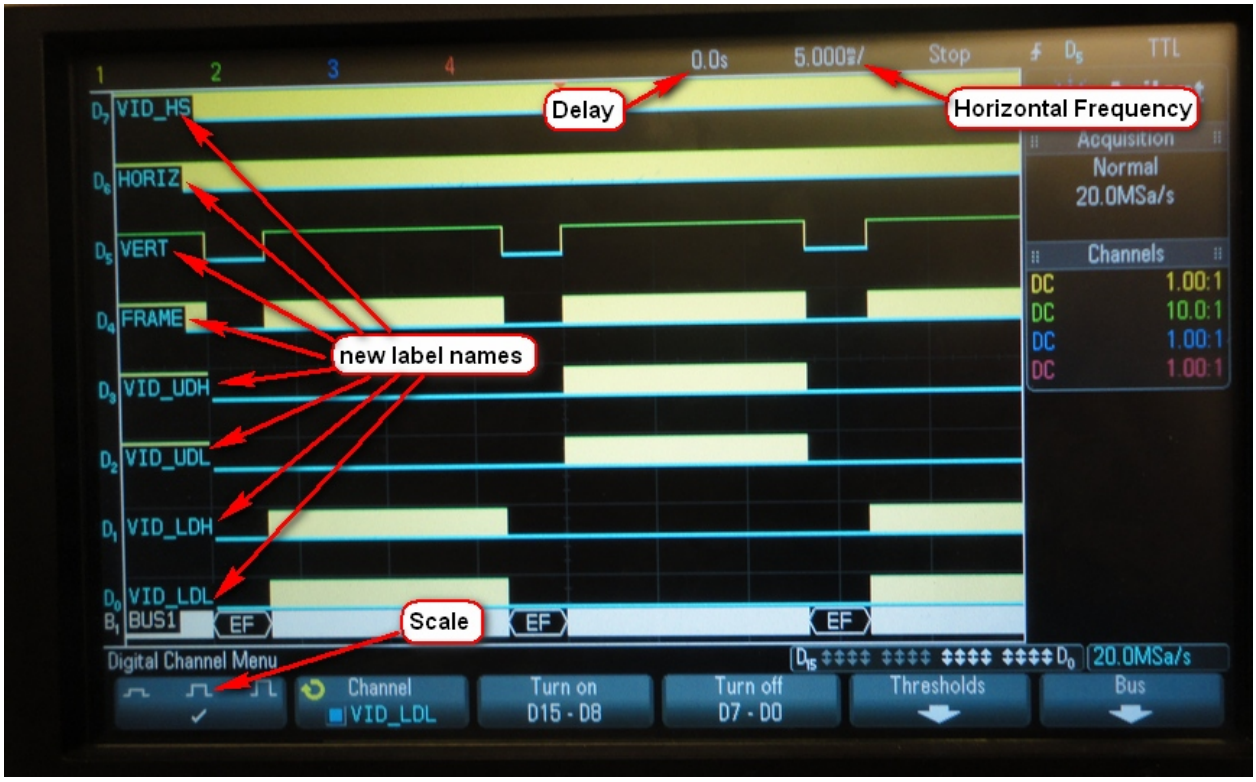


Figure 33 - New setting for labels, delay, and frequency

- 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 for multiple frames

- Here we see multiply frames. First the odd lines are captured (**VID_LDL** and **VID_LDH**) and then the even lines (**VID_UDL** and **VID_UDH**). Two vertical cycles equate to one full frame of video.

Go back to Quartus and scroll to line 74 of the camera.v Verilog code. See **Figure 35**

```
74  wire vid_ldl = (frame & !timer2[0] & !vid_address_low[1] & !vid_address_low[0] & clk_27 ) ? 1'b1 : 1'b0; // low byte odd addre
75  wire vid_ldh = (frame & !timer2[0] & vid_address_low[1] & !vid_address_low[0] & clk_27 ) ? 1'b1 : 1'b0; // high byte odd addre
76  wire vid_udl = (frame & timer2[0] & !vid_address_low[1] & !vid_address_low[0] & clk_27 ) ? 1'b1 : 1'b0; // low byte even addre
77  wire vid_udh = (frame & timer2[0] & vid_address_low[1] & !vid_address_low[0] & clk_27 ) ? 1'b1 : 1'b0; // high byte even addre
78
79  wire vert = ( ( timer1 > 30 &  timer1 <= 240 ) ) ? 1'b1 : 1'b0; // adjust vertical sync
80  wire horiz = ( ( horizontal > 300 & horizontal <= 1548) ) ? 1'b1 : 1'b0; // adjust horizontal sync
81  wire frame = ( horiz  & vert) ? 1'b1 : 1'b0 ; // address range enable
```

Figure 35- Timing Verilog code video in

- From line 79 (**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 wait until **30** vertical lines have been counted before starting to save. We stop when we have reached **240** lines. We are saving **210** vertical lines of video data (240-30= 210) in total.

- From line 80 (**Figure 35**), **horiz** represents the number of video bytes per horizontal line. Here we are waiting until 300 data pixels have been counted before we start saving to memory. Once the counter has reached 1548 it will stop saving to memory. Therefore, we are storing 1248 bytes per line (1548-300=1248). Note the 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 81 (**Figure 35**), **frame** represents 1 frame of video data. In this case 624 horizontal bytes times 210 vertical lines. (624 X 210= 131040)
- From line 74 and 75 (**Figure 35**), **vid_ldl** and **vid_ldh** are enables for storing odd lines to the on chip memory locations. If you scroll to line 166 you will see how address is set to **vid_address_low,** which starts at memory location 0 and updates the values of **video_in** to **data_caml[7:0]** or **data_caml[8-15],** depending on whether **vid_ldl** or **vid_ldh** is active low. See **Figure 36** for reference.
- From line 76 and 77 (**Figure35**), **vid_udl** and **vid_udh** are enables for storing even lines to the on chip memory locations. If you scroll to line 188 you will see how address is set to **vid_address_low,** which starts at memory location 0 and updates the values of **video_in** to **data_camh[0-7]** or **data_camh[8-15]** depending on whether **vid_udl** or **vid_udh** is active low. See **Figure 36** for reference.

```
164   // odd bytes memory frame load        ///
165   ////////////////////////////////////////
166      if ( vid_ldl )
167
168            begin
169            address_cam <= vid_address_low[17:2]; // low byte
170            data_caml[7:0] <= video_in;
171            end
172
173         else
174
175            if ( vid_ldh )
176
177            begin
178            address_cam <= vid_address_low[17:2]; // high byte
179            data_caml[15:8] <= video_in;
180            end
181
182         else
183
184   ////////////////////////////////////////
185   // even bytes memory frame load       ///
186   ////////////////////////////////////////
187
188            if ( vid_udl )
189
190            begin
191            address_cam <= vid_address_low[17:2]; // low byte
192            data_camh[7:0] <= video_in;
193            end
194
195         else
196
197
198            if ( vid_udh )
199
200            begin
201            address_cam <= vid_address_low[17:2]; // high byte
202            data_camh[15:8] <= video_in;
203            end
204
205
206      end
```

Figure 36 - Video data updates to upper and lower memory

Figure 37 gives a zoomed in view of what each single cycle looks like. A key observation can be made.

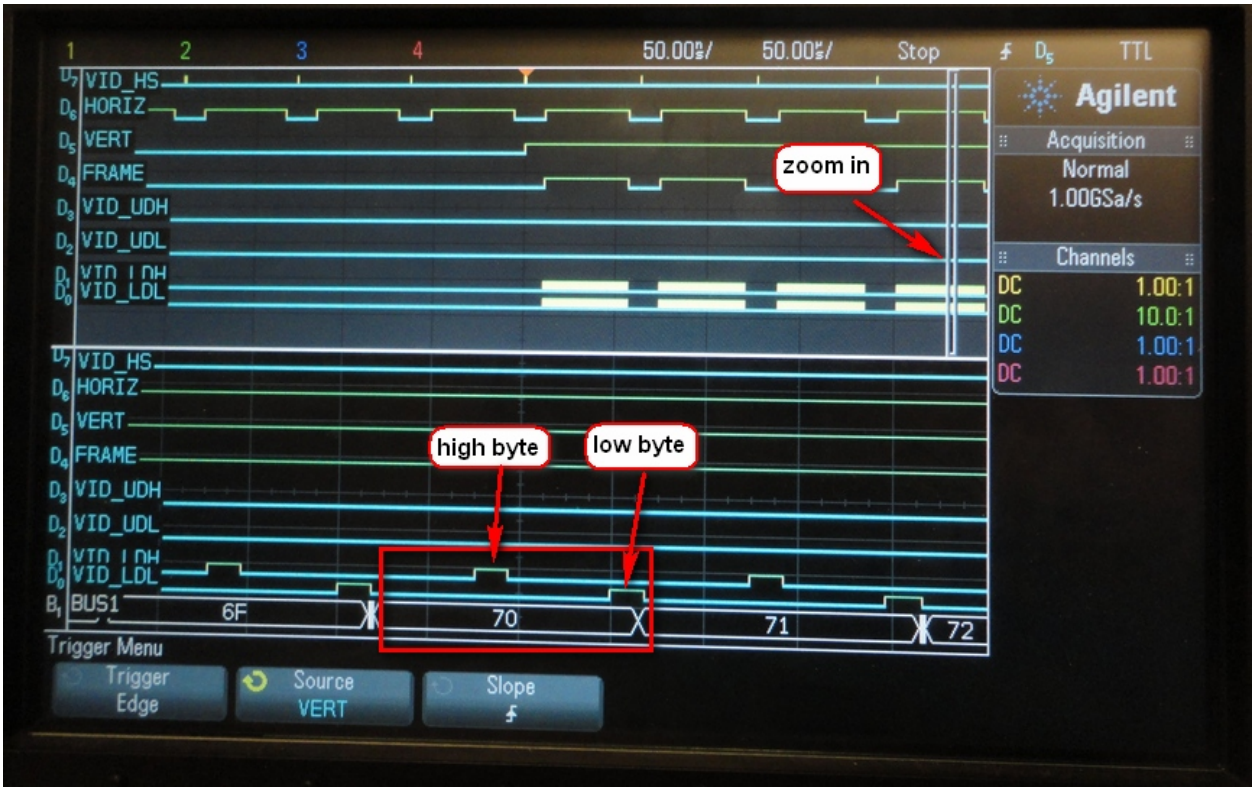


Figure 37- Zoomed in view of the interlace

- The **address** counter increments only after every two data latches (high byte and low byte) to store two 8 bit video values to the 16 bit memory register. This keeps in line with what we mentioned earlier that each video in bit is 8 bits but there are 16 bits of memory per address location.

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

VGA interface

The DE1-SOC 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 go to the following 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**.

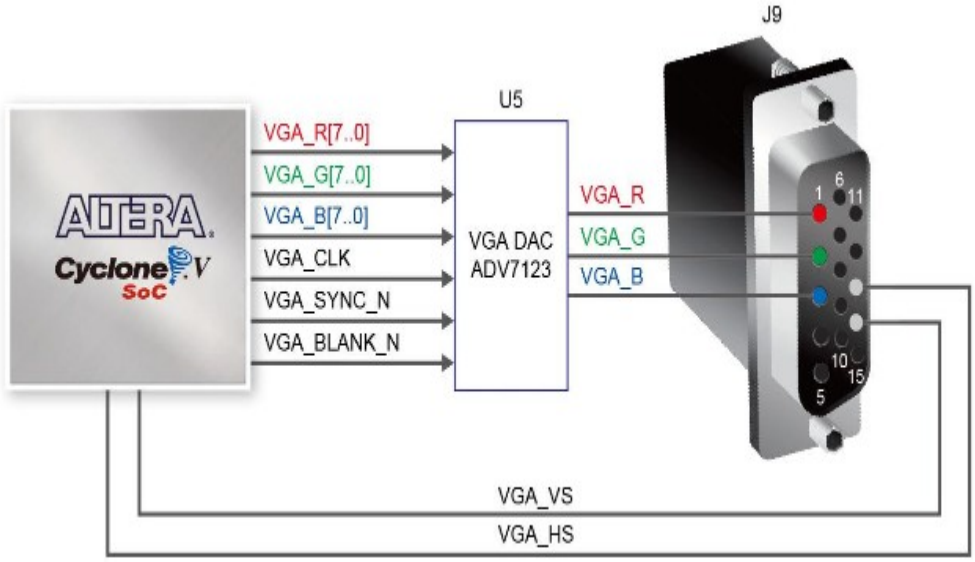


Figure 38-VGA to FPGA interface

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

Signal Name	FPGA Pin No.	Description	I/O Standard
VGA_R[0]	PIN_A13	VGA Red[0]	3.3V
VGA_R[1]	PIN_C13	VGA Red[1]	3.3V
VGA_R[2]	PIN_E13	VGA Red[2]	3.3V
VGA_R[3]	PIN_B12	VGA Red[3]	3.3V
VGA_R[4]	PIN_C12	VGA Red[4]	3.3V
VGA_R[5]	PIN_D12	VGA Red[5]	3.3V
VGA_R[6]	PIN_E12	VGA Red[6]	3.3V
VGA_R[7]	PIN_F13	VGA Red[7]	3.3V
VGA_G[0]	PIN_J9	VGA Green[0]	3.3V
VGA_G[1]	PIN_J10	VGA Green[1]	3.3V
VGA_G[2]	PIN_H12	VGA Green[2]	3.3V
VGA_G[3]	PIN_G10	VGA Green[3]	3.3V
VGA_G[4]	PIN_G11	VGA Green[4]	3.3V
VGA_G[5]	PIN_G12	VGA Green[5]	3.3V
VGA_G[6]	PIN_F11	VGA Green[6]	3.3V
VGA_G[7]	PIN_E11	VGA Green[7]	3.3V
VGA_B[0]	PIN_B13	VGA Blue[0]	3.3V
VGA_B[1]	PIN_G13	VGA Blue[1]	3.3V
VGA_B[2]	PIN_H13	VGA Blue[2]	3.3V
VGA_B[3]	PIN_F14	VGA Blue[3]	3.3V
VGA_B[4]	PIN_H14	VGA Blue[4]	3.3V
VGA_B[5]	PIN_F15	VGA Blue[5]	3.3V
VGA_B[6]	PIN_G15	VGA Blue[6]	3.3V
VGA_B[7]	PIN_J14	VGA Blue[7]	3.3V
VGA_CLK	PIN_A11	VGA Clock	3.3V
VGA_BLANK_N	PIN_F10	VGA BLANK	3.3V
VGA_HS	PIN_B11	VGA H_SYNC	3.3V
VGA_VS	PIN_D11	VGA V_SYNC	3.3V
VGA_SYNC_N	PIN_C10	VGA SYNC	3.3V

Table 11- Pin assignments for the VGA on the FPGA

- VGA_R[0-7] –These are the 8 digital bits that represent the analog RED colour output to the 15 pin D_SUB interface. 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[0-7] –These are the 8 digital bits that represent the analog GREEN colour output to the 15 pin D_SUB interface. 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[0-7] –These are the 8 digital bits that represent the analog BLUE colour output to the 15 pin D_SUB interface. 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 the standard frequency for most VGA monitors.

- VGA_BLANK_N –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 of 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 of 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 completed 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 signals

Unlike the video in chip ADV7180 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 the 25 MHz clock create a counter and generate the needed sync pulses.

A working example of what the Verilog code could look like can be found at:

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 45. The code should look the same as **Figure 39.** Vid_clk <= clkcount[0]. This represents the 50 MHz having been divided to create the needed 25 MHz clock.

```
40  ////////////////////////////////////////
41  /// general clock divider
42  ////////////////////////////////////////
43
44
45   always @ (posedge clk )
46
47  begin
48
49        clkcount <= clkcount + 1;
50
51  end
```

Figure 39 - Clock divided to generate the 25 Mhz clock

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

- Clrvidh is used to reset the horizontal counter to zero and then count up to a certain value. In this case we have chosen 800. See line 37 in **Figure 40**. The counter is generated in **Figure 41** starting at line 80
- Clrvidv is used to reset the vertical counter to zero and then count up to a certain value. In this case we have chosen 525. See line 38 in **Figure 40**. The counter is generated in **Figure 41** starting at line 99.

```
29  ////////////////////////////
30  ////    control values  /////
31  ////////////////////////////
32  reg         vid_clk;
33
34  wire        vsync = ((contvidv >= 491) & (contvidv < 493))? 1'b0 : 1'b1;
35  wire        hsync = ((contvidh >= 664) & (contvidh < 760))? 1'b0 : 1'b1;
36  wire        vid_blank = ((contvidv >= 8) & (contvidv <  420) &(contvidh >= 20) & (contvidh < 624))? 1'b1 : 1'b0;
37  wire        clrvidh = (contvidh <= 800) ? 1'b0 : 1'b1;
38  wire        clrvidv = (contvidv <= 525) ? 1'b0 : 1'b1;
39
```

Figure 40- Counters and sync pulses

```
72   ////////////////////////////////
73   // horizontal counter       /////
74   ////////////////////////////////
75
76   always @ (posedge vid_clk )
77
78   begin
79
80         if(clrvidh)
81         begin
82         contvidh <= 0;
83         end
84
85         else
86         begin
87         contvidh <= contvidh + 1;
88         end
89   end
90
91   ////////////////////////////////////////
92   //vertical counter when clrvidv is low /
93   ////////////////////////////////////////
94
95   always @ (posedge vid_clk)
96
97   begin
98
99         if (clrvidv)
100        begin
101        contvidv <= 0;
102        end
103
104        else
105        begin
106           if
107           (contvidh == 798)
108           begin
109           contvidv <= contvidv + 1;
110           end
111        end
112  end
```

Figure 41- Counters generated

- Vsync pulse - active low when contvidv is greater than or equal to 491 and less than 493, or else it is active high. See line 34 in **Figure 40**.
- Hsync pulse - active low when contvidh is greater than or equal to 664 and less than 760, or else it is active high. See line 35 in **Figure 40**.
- Vid_blank- active high if contvidv is great than or equal to 8 and less than 420 and if contvidh is greater than or equal to 20 and less than 624. See line 36 in **Figure 40.**

Let's examine the signals and verify the code above. Connect the digital logic pins of the logic analyzer to the 40 pin header GPIO-0 as described in columns 1 and 3 of **Table 12**.

40 pin HeaderJP0	Pin assignment	Digital lead default name	Rename label
GPIO[0]	AC18	D0	VID_CLK
GPIO[1]	Y17	D1	VSYNC
GPIO[2]	AD17	D2	HSYNC
GPIO[3]	Y18	D3	VID_BLANK
GPIO[4]	AK16	D4	CLRVIDH
GPIO[5]	AK18	D5	CLRVIDL

Table 12- Pin assignments for the clk and sync pulses

Perform the following procedures on the logic analyser:

- Press **Digital.**
- Press **Bus** and disable all the Buses.
- Press **Back**
- De-select channels D8 to D15 and select channels D5-D0.
- Press **Label** and rename D0-D5 according to column 4 in **Table 12**.
- Set delay to **0.0**s.
- Set **horizontal** frequency to **42.000**n/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**.

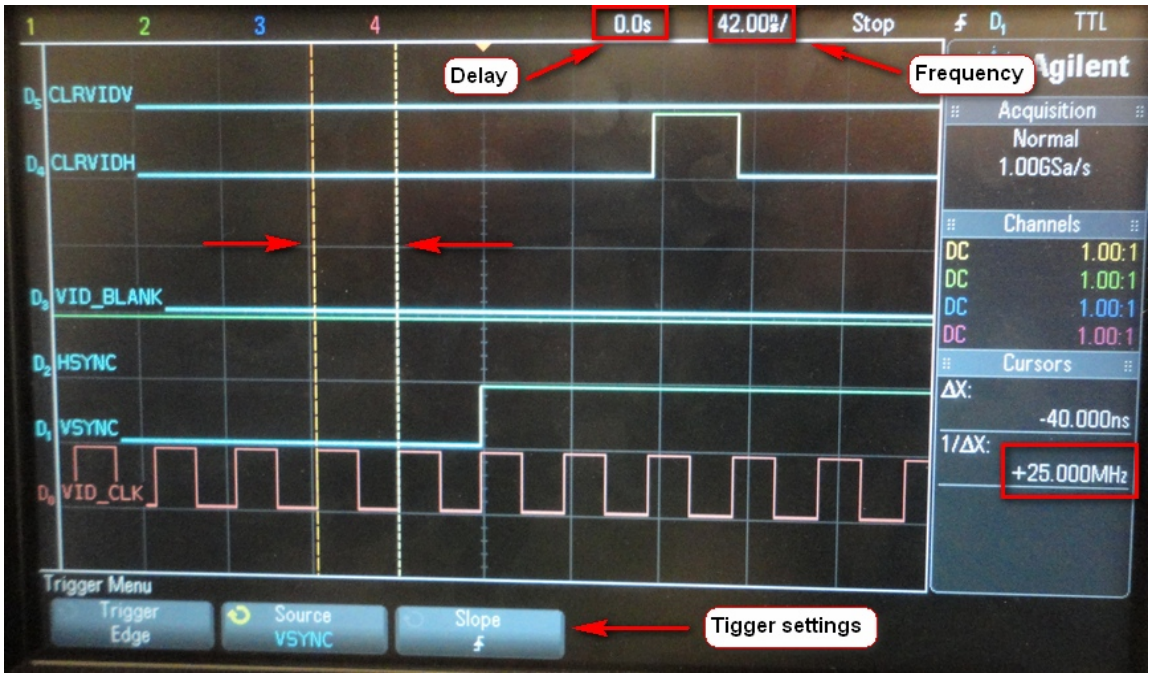


Figure 42-Clock frequency measurements

- Press **Cursers** and measure the (1/DELTA) frequency of Vid_clk.
- The result should be around 25 MHz. 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 **Restart label** is, is where the counters are reset to zero and a new count begins. Vid_blank becomes active high after 8 hsync cycles. This is also the beginning of a new video frame.

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

Part 2- Creating and examining address counters and enables

If you recall from **tutorial 2** we created latches that stored composite video from a camcorder. These latches were to be used to store digital pixel values in SRAM on the DE1-SOC. With that in mind we can recall the following:

- The memory configuration is 16 address lines and 32 data bits.
- We were using the substitution method for video capture and display.
- The Lower 16 bits [0-15] were for odd video line.
- The Upper 16 bits [16-31] were for even video lines.
- Two frames were required to create a full video frame.
- **Vid_udl** and **Vid_udh** are used to store all the even lines of video data.

- **Vid_Idl** and **Vid_Idh** are used to store all the odd lines of video data.
- There were a total of 210 vertical lines. Also note that we started after 30 vertical lines and ended at 240 vertical lines (240-30= 210). So the full frame is 210x2= 420
- There were a total of 624 horizontal video pixels (8 bits per pixel). Again note that we started the counter after 150 pixels 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:

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

With this information we can now create our address counters. A total of 2 address counters will be required:

- An address counter for odd lines.
- An address counter for even lines.

A working example of what the Verilog code should look like can be found at:

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

Select-DE1-SOC>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.v**.

Scroll to line 80. It should look like **Figure 44**. These are all the enables for the address counters.

```
76  ////////////////////////////////////////
77  //////// memory enables              ////
78  ////////////////////////////////////////
79
80  wire        ramvidv = ( ( (contvidv <= 420) ) ? 1'b0 : 1'b1);
81  wire        adden = ( (( contvidh < 624) & (contvidv <= 420) ) ? 1'b1 : 1'b0); // address enable
82
83  wire        read = (vid_clk & adden) ? 1'b1 : 1'b0; // oe to memory enable
84  wire        read_ll = ( adden & !ramaddressl_odd[0] & vid_clk & !oddeven ) ? 1'b0 : 1'b1; // low odd address enable
85  wire        read_hl = (adden & ramaddressl_odd[0] & vid_clk & !oddeven ) ? 1'b0 : 1'b1; // low even address enable
86  wire        read_lh = ( adden & !ramaddressl_even[0] & vid_clk & oddeven ) ? 1'b0 : 1'b1; // high odd address enable
87  wire        read_hh = ( adden & ramaddressl_even[0] & vid_clk & oddeven ) ? 1'b0 : 1'b1; // high even address enable
88
89  parameter address_low = 19'h00000;  // lower address start at 0 meg
90
```

Figure 44 - Address enables for odd and even lines

- Line 80 **ramvidv** is used to reset the address counters. As long as **contvidv** is less than or equal to 420, **ramvidv** will be active low the address count (**ramaddressl_odd** and **ramaddressl_even**) will keeps incrementing. See lines 147 and 169 in **Figure 45**.
- Line 81 -**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.
- Line 83-**Read** will be used to read or write to the SRAM memory. Active low will read data from the camera and active high will send data to the monitor.
- Line 84 and 192 -**Read_ll** is the odd line low byte memory enable (**data_motl [7-0]**) for active high. See **Figure 44** and **46**.
- Line 85and 202-**Read_hl** is the odd line high byte memory enable (**data_motl [15-8]**) for active high. See **Figures 44** and **46**.
- Line 86 and line -212 **Read_lh** is the even line low byte memory enable (**data_moth [7-0]**) for active high. See **Figures 44** and **46**.
- Line 87 and line 222- **Read_hh** is the even line high byte memory enable (**data_moth [15-8]**) for active high. See **Figures 44** and **46**.

```
139   ////////////////////////////////////////////////////////
140   // address counter out to monitor odd lines low memory //
141   ////////////////////////////////////////////////////////
142
143   always @ (posedge vid_clk )
144
145  begin
146
147         if(ramvidv)
148        begin
149         ramaddressl_odd <= address_low;// memory reset to "0"
150         end
151
152         else
153        begin
154         if (adden & !oddeven )
155            begin
156            ramaddressl_odd <= ramaddressl_odd + 1;
157            end
158         end
159   end
160
161   ////////////////////////////////////////////////////////
162   // address counter out to monitor even lines low memory  //
163   ////////////////////////////////////////////////////////
164
165   always @ (posedge vid_clk)
166
167  begin
168
169         if (ramvidv)
170        begin
171         ramaddressl_even <= address_low; // memory reset to "0"
172         end
173
174         else
175        begin
176            if
177            (adden & oddeven )
178           begin
179            ramaddressl_even <= ramaddressl_even + 1; // 798 horizontal pixels
180            end
181         end
182   end
183
184
```

Figure 45 - Memory counters for odd and even lines

```
188
189   always @ (negedge vid_clk)
190
191   begin
192         if (!read_ll )
193         begin
194
195         video_mot <= data_motl[7:0];
196         address_mot <= ramaddressl_odd[16:1]; // memory odd byte low
197         end
198
199         else
200
201
202         if (!read_hl )
203         begin
204
205         video_mot <= data_motl[15:8];
206         address_mot <= ramaddressl_odd[16:1]; // memory odd byte high
207         end
208
209         else
210
211
212         if (!read_lh )
213         begin
214
215         video_mot <= data_moth[7:0];
216         address_mot <= ramaddressl_even[16:1]; // memory even byte low
217         end
218
219         else
220
221
222         if (!read_hh )
223         begin
224
225         video_mot <= data_moth[15:8];
226         address_mot <= ramaddressl_even[16:1]; // memory even byte high
227         end
228
229   end
```

Figure 46 - Latch enablers and upper and lower memory

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

40 pin HeaderJP0	Pin assignment	Digital lead default name	Rename label
GPIO[0]	AC18	D0	VID_CLK
GPIO[1]	Y17	D1	READ
GPIO[2]	AD17	D2	ADDEN
GPIO[3]	Y18	D3	ODDEVEN
GPIO[4]	AK16	D4	READ_LL
GPIO[5]	AK18	D5	READ_HL
GPIO[6]	AK19	D6	READ_LH

GPIO[7]	AJ19	D7	READ_HH
GPIO[8]	AJ17	D8	Address (bus)
GPIO[9]	AJ16	D9	Address (bus)
GPIO[10]	AH18	D10	Address (bus)
GPIO[11]	AH17	D11	Address (bus)
GPIO[12]	AG16	D12	Address (bus)
GPIO[13]	AE16	D13	Address (bus)
GPIO[14]	AF16	D14	Address (bus)
GPIO[15]	AG17	D15	Address (bus)

Table 13 -Tutorial 3 Part 2

Do the following on the logic analyser:

- Press **Digital.**
- Press **Bus** and enable **Bus 1**.
- Set D15-D8 to represent that bus
- Press **Back.**
- Press **Digital.**
- Set **Scale** to **medium.**
- D5-D0 should still be selected from part 1. Select D7 and D6
- Press **Label** and rename D0-D7 according to column 4 in **Table 13.**
- Set delay to **0.0**s.
- Set **Horizontal** frequency to **20.00**u/s**.**
- Press **Trigger**.
- Set **Edge** trigger.
- Set **ADDEN** as source.
- Set **Rising Edge** as slope.
- Press **Single** to trigger a new event.

The result should look similar to **Figure47.**

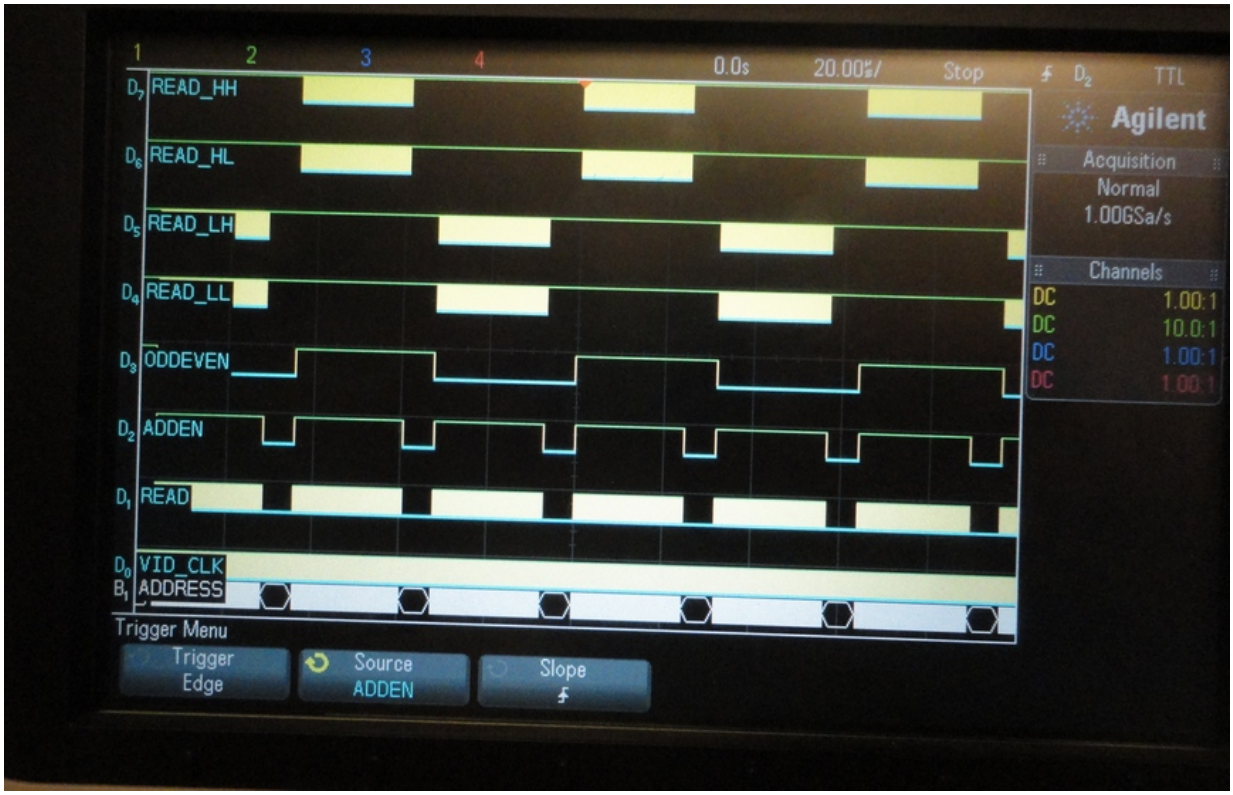


Figure 47- Address and data enable for multiple frames

Note that when **oddeven** is active low, READ_LL and READ_LH are active and READ_HL and READ_HH remain active high. This means that only the odd lines of data are being sent to the monitor from the memory. When **oddeven** is active high READ_HL and READ_HH are active and even lines of video data are being sent to the monitor from the memory.

- Press **Trigger**.
- Change **ADDEN** and make **ODDEVEN** as source.
- Set **Falling Edge** as the slope.
- Press **Single** to trigger a new event.
- Press **Zoom.**
- Set **Horizontal** frequency to **20**u/s.
- Set **Delay** to **50**u/s.

The result should look similar to **Figure 48**.

Here we can see that READ_LL and READ_LH are never active at the same time.

- READ_LL is active low when READ and ADDEN are active high and ODDEVEN is active low (see line 84 in **Figure 44**) This means that when READ_LL is active, data_motI[7:0] is loaded into video_mot and ramaddressl_odd is loaded into address_mot. See lines 195 and 196 from **Figure 46**.

- READ_LH is active low when READ and ADDEN are active high and when ODDEVEN is active low (see line 85 in **Figure 44**) This means that when READ_LH is active, high data_motI[15:8] is loaded into video_mot and ramaddressl_odd is loaded into address_mot. See lines 205 and 206 from **Figure 46**.
- Also note that the address count only changes every other cycle. This is because the data is 16 bits wide but the video buffer is only 8 bits wide. Therefore, it takes two cycles to collect video data from any one address location.

Figure 48 - Frame zoom in on enable odd lines

- Press **Zoom** again, in order to turn it off.
- Set **Rising Edge** as slope.
- Press **Single.**
- Press **Zoom.**

The result should look like **Figure 49**.

Figure 49 - Frame zoom in on enables even lines

Here we can see that READ_HL and READ_HH are never active at the same time.

- READ_HL is active low when READ, ADDEN and ODDEVEN are active high (see line 86 in **Figure 44**) This means that when READ_HL is active, data_moth [7:0] is loaded into video_mot and ramaddressl_even is loaded into address_mot. See lines 215 and 216 in **Figure 46**.
- READ_HH is active low when READ, ADDEN and ODDEVEN are active high (see line 87 in **Figure 44**) This means that when READ_HH is active, data_moth [15:8] is loaded into video_mot and ramaddressl_even is loaded into address_mot. See lines 225 and 226 from **Figure 46**.
- Also note that the address count only changes every other cycle. This is because the data is 16 bits wide and the video buffer is only 8 bits wide. Therefore, it takes two cycles to collect video data from any one address location.

We have now created the address counters and data enables to retrieve data to and from memory. 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 in part 2. Before we continue we will learn how to use the onboard libraries in Quartus to create a memory module. This concludes part 2.

Part 3 – Creating a memory module using the Quartus library

In this tutorial we will learn how to create an on chip SRAM memory module. Before we continue, create a new folder called **library**. Within that folder, create another folder called **memory**. We will use this folder to create the memory module. If you recall from **page 30** we said that the ideal memory size for creating a full video frame was ~2 Gbits of memory.

- 1 Gbit for odd memory lines.
- 1 Gbit for even memory lines.
- To get this configuration we choose to use 16 address lines and 32 data lines.
- $(32 \times 2^{16}) = 2{,}097{,}152$

With this information we can create a memory module.

Open Quartus and using the Quartus **new project wizard** create a new project called **memory**. This should be the empty folder just created.

On the right side of the Quartus folder you will see a menu that looks like **Figure 50.**

Figure 50-Quartus libraries

Select basic Functions > on chip Memory > RAM:2-PORT. A new folder will appear similar to **Figure 51**

Select Verilog (default is VHDL) and create a name. In this case it is **memory16**. Press **OK**.

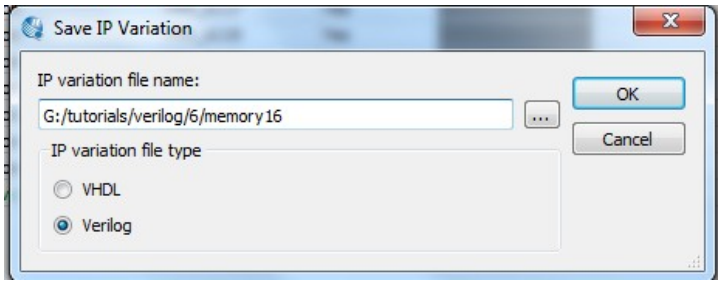


Figure 51 - Create a memory module

The result should look like **Figure 52**

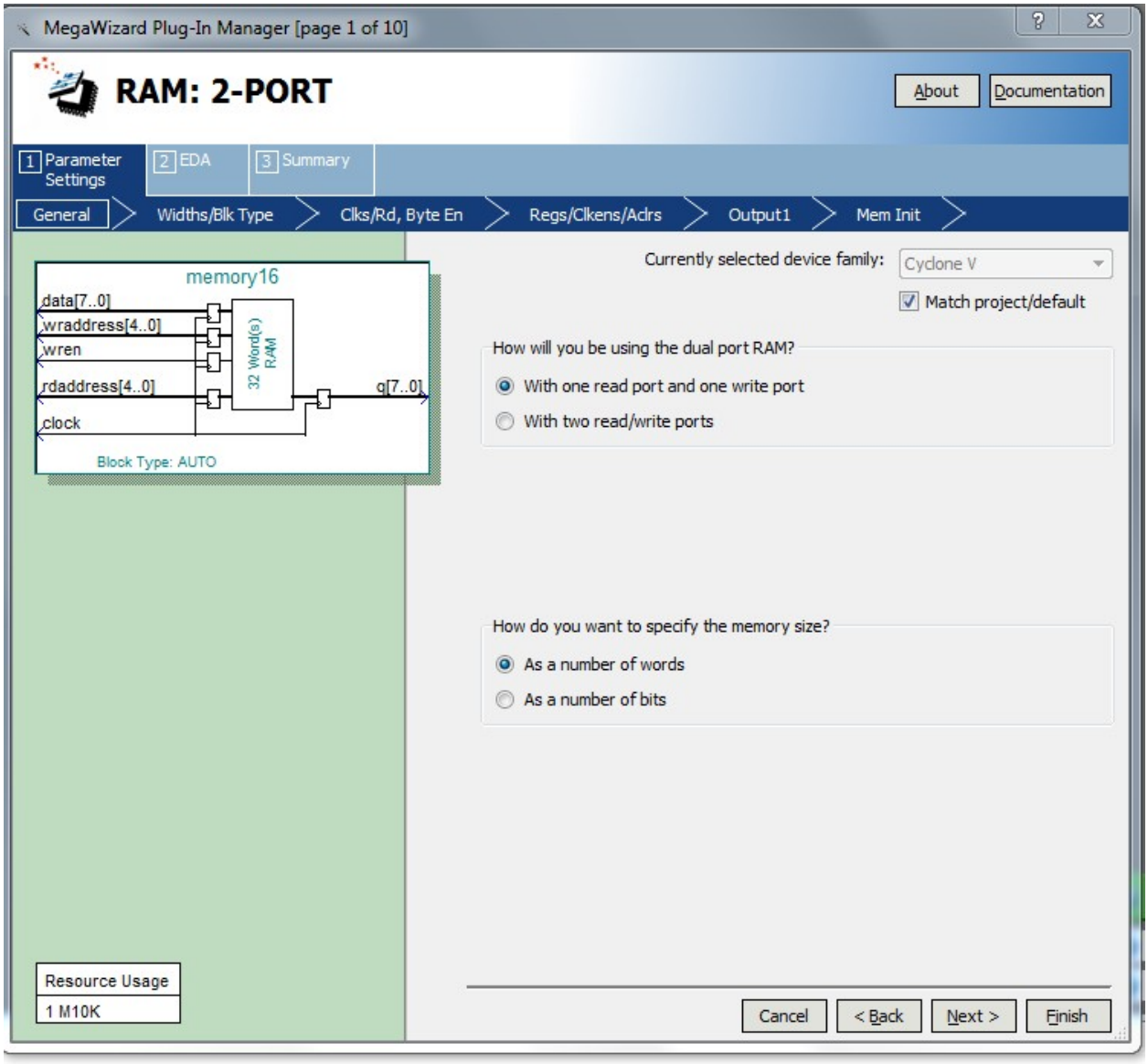


Figure 52 - Opening page of the chip memory module

Press next and the next page will appear. Here we will set up the memory size and the number of data bits. In the middle of the page you will find the following:

How wide should the 'data_a' input bus be? Change this to **32**

At the top of the page you will see the following;

How many 32 bit words of memory? Select 65532. You will have to scroll through the menu to get to this value. The result should look like **Figure 53**.

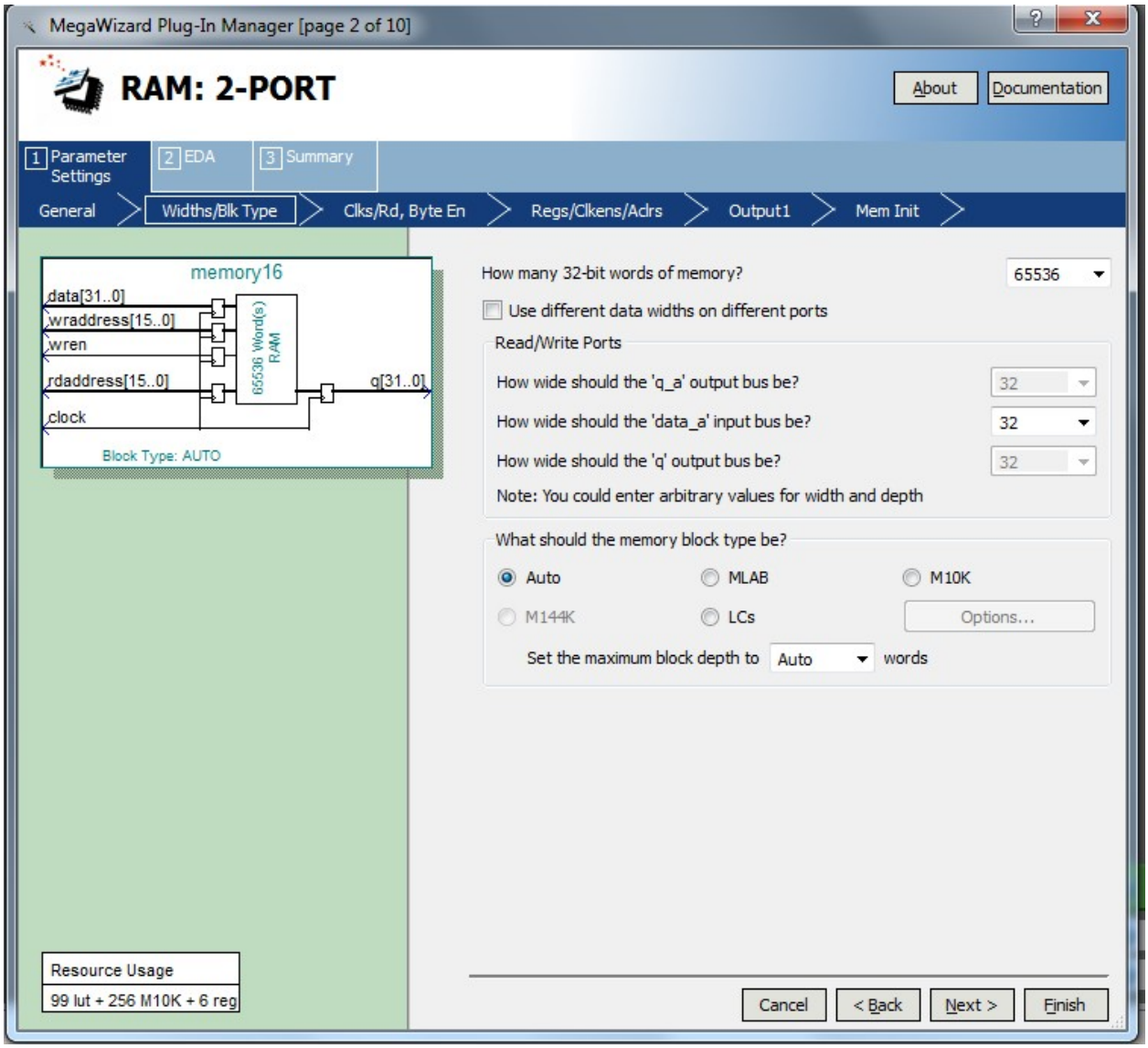


Figure 53-Select memory size and data size

Press next. Near the middle of the page you will see the following:

Create a 'rden' read enable signal. Select the box just beside this.

Below that you should see the following:

Create byte enable for Port A. Select this box.

This is going to create 4 byte enables, since the bus is 32 bits wide.

The result should look like **Figure 54**.

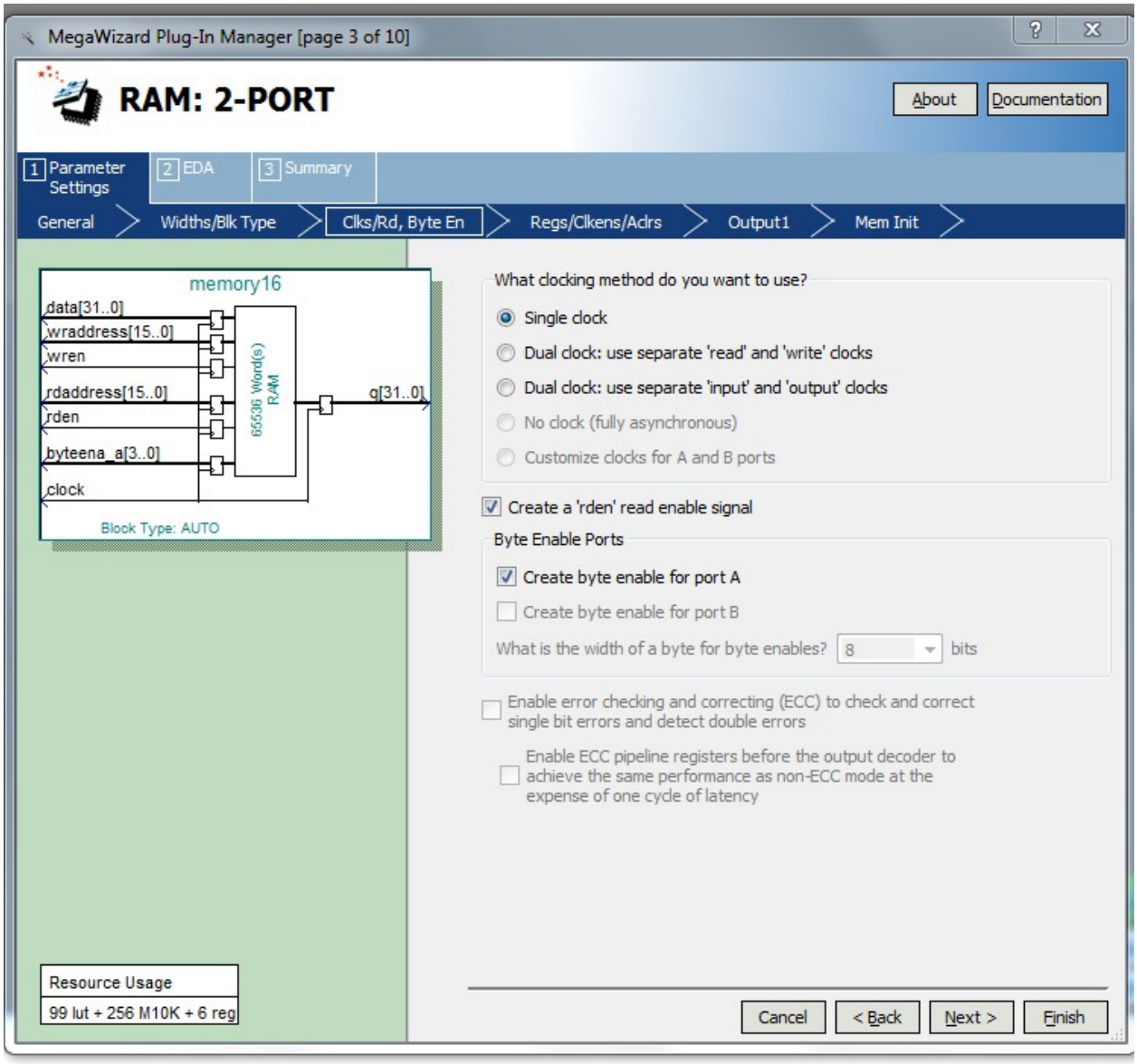


Figure 54 - Creating read enables (4 of them in this case)

Now we have created the memory module. Keep pressing next until the project is saved to the directory.

To open the memory module all you have to do is go to the directory you created it in and select the memory module name. In this case it is called **memory16.v**. This will open the first folder which was

Figure 52. This is valuable to know if ever you want to make any changes to the memory size. The first 16 data bits will be used for odd line video and the upper 16 bits will be used for the even line video. With this information we are now ready to create a state machine to capture video from a video source such, as a camcorder, store it in memory and then take those stored values and display them to a display source, such as a monitor.

This concludes Part 3.

Part 4- 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). Data bits D0-D16 were used to store odd lines of data and data bits D16-D31 were used to store even lines of data.

In part 2, we created a state machine that would take data values stored in memory and display them to a display device, such as a monitor. The state machine took into account the interlacing of odd and even lines to create a full screen picture.

In part 3, we learned how to create an on chip SRAM memory using the Quartus library. We will implement this memory into this section.

The objective in part 4 is to combine the state machines created in Part 1-3 (carmera.v, monitor.v and memory16.v). We will create another state machine that will display data and update data from memory using the 25 MHz clock. If we recall from earlier, we decided to use the substitution method since our memory size was limited. Now 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 on the monitor it can be updated
- **Address** counter will keep track of the location in memory, for both read and write video values
- **Data** is 32 bits wide so it will be divided into 4 bytes to make it compatible with the video data size (8 bits)
- **Data enables**- 4 byte enables will have to be created for both read and write, in order to satisfy the previous point
- A **Read** enable will need to be created in order to send 8 bit video data to the monitor
- A **Write** enable will need to be created in order to receive 8 bit video data from the camcorder
- We need to make a decision whether to sync the state machine to video in (camera.v -27 MHz) or video out (monitor.v – 25 MHz).

Expanding upon the last point, if we use video out (read from memory) as the sync for timing, we would have to asynchronously latch the new video during times when the video is not being read from

memory. This means we need to make a temporary location, in order to store values coming from the video in source and then update the memory during non-display enable times. Reference **Figure 55.**

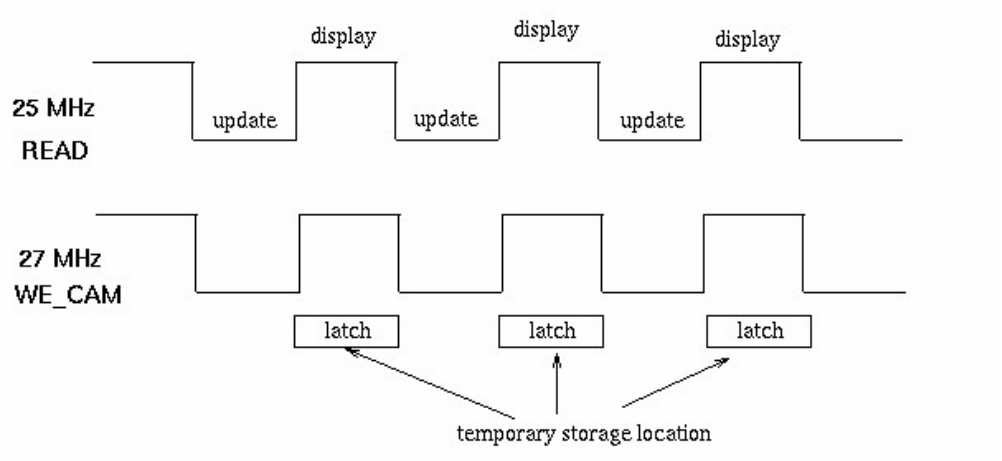


Figure 55-Sync video in with video out

- When **read** is active high video is displayed from memory

This means that during **read** active low periods, video_in (latched) can be updated. The values in the **latch** register are the latest value from the video source. They are updated according to we_cam, which is the combination of all the video in enables. This will become apparent when we examine the signals with a MSO-X-3024A scope.

A working example of what the Verilog code should look like can be found at:

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

Select -DESL Online Tutorials>Tutorial4_video.zip.

Create a directory and unzip the file. Using the Quartus **new project wizard** create a new project called **video**. Compile the project. Open the Verilog file called video. Scroll to line 156 of the file and you should see the same code displayed in **Figure 56.**

```
152    ////////////////////////////////////
153    // on board memory modules      ////
154    ////////////////////////////////////
155
156  ram_16 u3 (
157
158            .clock(clk),
159            .wraddress(wraddress),
160            .rdaddress(rdaddress),
161            .data(data_in),
162            .byteena_a(vid_en),
163            .q(data_q),
164            .rden(read),
165            .wren(!read)
166  );
167
168  ////////////////////////////////////////////
169  /// variables for loading I2C programmer ///
170  ////////////////////////////////////////////
171
```

Figure 56-Sync Verilog code

This is a duplicate of the on chip memory module created in part 3 of this tutorial. You do not have to create this module again for this tutorial because it has been included as part of the ZIP file just downloaded. If you look at all the Verilog files for this project you will find one called ram_16.v, which is this SRAM memory module.

Scroll to line 164 and 165 of **Figure 56**. Here you can see that **read** is connected to both **rden** and **wren**. This verifies that data updates and displays are synchronized to **read**, either being active low or high.

```
204   ////////////////////////////////
205   ///  memory module          ////
206   ////////////////////////////////
207   /// 16 address lines        ////
208   /// 32 data lines           ////
209   /// 4 byte enables          ////
210   ////////////////////////////////
211
212       wire[15:0] address = ((read) ? address_mot : address_cam);
213       assign wraddress = address_cam;    // address counter in
214       assign rdaddress = address_mot;    // address counter out
215
216       assign data_in[31:16] = data_camh; // upper 16 bits data in
217       assign data_in[15:0] = data_caml;  // lower 16 bits data in
218
219       assign data_motl = data_q[15:0]; // lower 16 bits data out
220       assign data_moth = data_q[31:16];// upper 16 bits data out
221
222       assign vid_en[0] = vid_ldl;  // video in byte enable
223       assign vid_en[1] = vid_ldh;  // video in byte enable
224       assign vid_en[2] = vid_udl;  // video in byte enable
225       assign vid_en[3] = vid_udl;  // video in byte enable
226
227       assign vid_clk = clkcount[0]; // 25 Mhz clock
228       assign clken = swt[0];        // camera reset
229       assign led = swt;
230       assign videorgb = video_mot;  // 8 bit video out display register
```

Figure 57-Monitor update enables and syncs

Figure 57 shows how all the signals from the camera.v module and monitor.v module are connected to the ram_16.v module. Some key observations:

- Line 214 - rdaddress is connected to address_mot. This is the address counter created in the monitor.v module.
- Line 213 - wraddress is connected to address_cam. This is the address counter created in the camera.v module.
- Line 216 and 217 – data_in[31-0] the 32 bits of data lines from ram_16.v are connected to the 32 data bits created in camera.v. These are the two data registers data_camh[15-0] and data_caml[15:0].
- Line 219 and 220 – data_q[31-0] the 32 bits of data lines from ram_16.v are connected to the 32 data bits created in monitor.v. These are the two data registers data-moth[15-0] and data_motl[15:0].
- Line 222 – 225 vid_en[3-0] these are the 4 enables (**vid_ldl**,**vidldh**,**vidudl** and **vidudh**) that were created in **camera.v.**

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

Connect the DE1-SOC as follows:

- The monitor cable to the monitor input (15 pin video cable)
- The composite connector to the camcorder (RCA Jack)
- The digital connectors from the logic analyser should still be connected the same way they were in part 2

The result should look similar to **Figure 58**

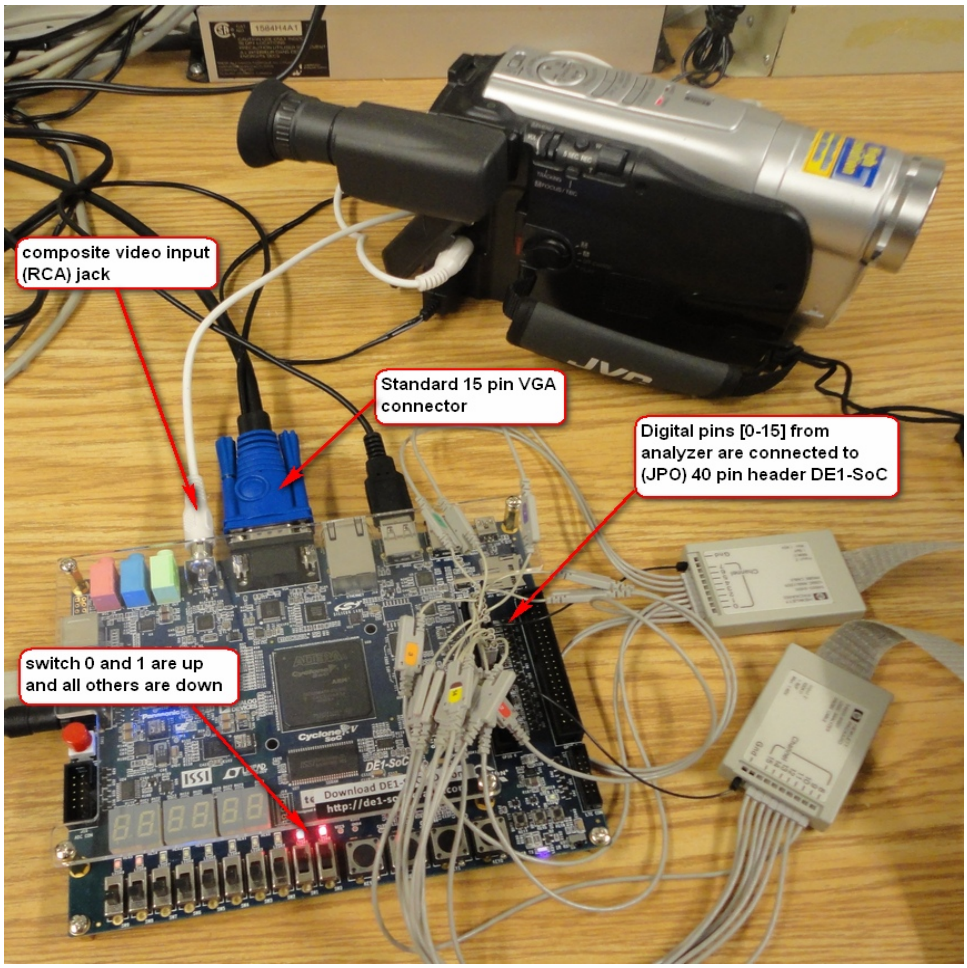


Figure 58-Layout of the video connectors

Download **the video.sof** file to the DE1-SOC board. Make sure switches **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.

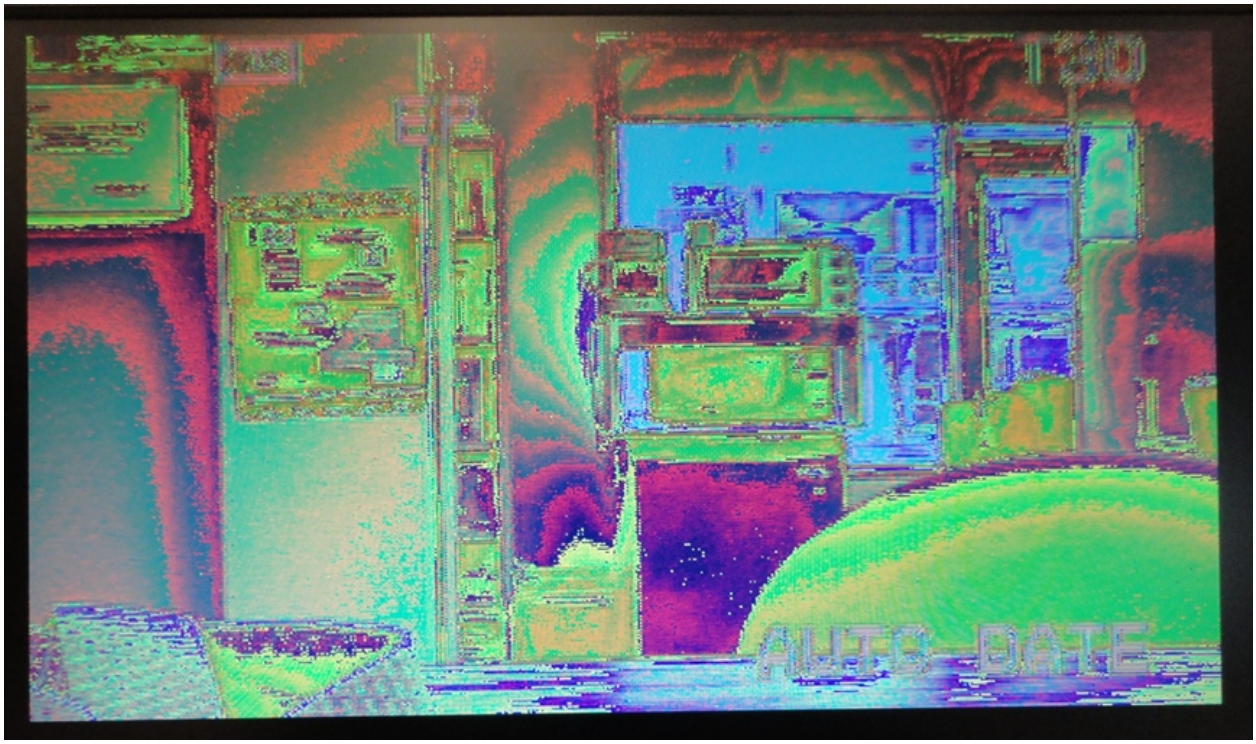


Figure 59-Sample video on the monitor

Note that this is a very simple video so there is no depth; it is only displaying colours according to shading. We will learn more about how to improve this video later.

Before continuing, open up the **video.v** file in Quartus and make sure the **GPIO** assignments match the values found in **Figure 60**.

```
232    ////////////////////////////////////////////////
233    // outputs to debug circuit gpio(0)       /////
234    ////////////////////////////////////////////////
235
236       assign gpio[0] = vid_clk;
237       assign gpio[1] = read;
238       assign gpio[2] = we_cam;
239       assign gpio[3] = oddeven;
240       assign gpio[4] = vid_ldl;
241       assign gpio[5] = vid_ldh;
242       assign gpio[6] = vid_udl;
243       assign gpio[7] = vid_udh;
244
245       assign gpio[15:8] = address_cam[7:0];
```

Figure 60 – Pin out assignments for write to memory from the camcorder

If not, update the assign values, recompile the code, and then download it to the DE1-SoC board.

Let's examine some of the signals using the MSO-X-3024A scope. Do the following:

- Press **Digital**
- Press **Bus** and enable **Bus 1**
- Set D15-D8 to represent that bus
- Press **Back**
- Set **Scale** to **medium**
- Press **Label** and rename D0-D15 according to column 4 in **Table 14**

40 pin HeaderJP0	Pin assignment	Digital lead default name	Rename label
GPIO[0]	AC18	D0	VID_CLK
GPIO[1]	Y17	D1	READ
GPIO[2]	AD17	D2	WE_CAM
GPIO[3]	Y18	D3	FRAME
GPIO[4]	AK16	D4	VID_LDL
GPIO[5]	AK18	D5	VID_LDH
GPIO[6]	AK19	D6	VID_UDL
GPIO[7]	AJ19	D7	VID_UDH
GPIO[8]	AJ17	D8	Address (bus)
GPIO[9]	AJ16	D9	Address (bus)
GPIO[10]	AH18	D10	Address (bus)
GPIO[11]	AH17	D11	Address (bus)
GPIO[12]	AG16	D12	Address (bus)
GPIO[13]	AE16	D13	Address (bus)
GPIO[14]	AF16	D14	Address (bus)
GPIO[15]	AG17	D15	Address (bus)

Table 14-Part 4 pin outs for the 40 pin header

- Set delay to **2.00**u/s.
- Set **Horizontal** frequency to **10.0**u/s.
- Press **Trigger**
- Select **Edge** then **Edge trigger**
- Set **Frame** as Arm A
- Set **Falling Edge** as slope A
- Set **VID_UDL** as Trigger B
- Set **Rising Edge** as slope B
- Press **Single** to trigger event

The result should look very similar to **Figure 61**. We can make several observations:

- The **Frame** going active high is the beginning of a new horizontal row

- While **Frame** is active high even lines of video data are being captured from the camcorder and stored in SRAM memory. This is verified by the fact that **WE_CAM**, **VID_UDL** and **VID_UDH** are active during this period of time.

Let us zoom in closer to get a better understanding of how the data transfer takes place

Figure 61-Even line enable from the camcorder

Press **Zoom** on the analyzer and the result should look similar to **Figure 62**.

Figure 62-Even line enable zoomed in

Here we can see that for every single address there are two 8 bit video values sent from the video source to the on chip memory. In this case it is the even line.

Perform the following:

- Turn off **Zoom**
- Press **Trigger**
- Change **Trigger B** to VID_LDL
- Press **Single** to trigger event

The result should look similar to **Figure 63**

Figure 63-Odd line enable from the camcorder

We can make several observations:

- The **Frame** going active high is the beginning of a new horizontal line
- While **Frame** is active high, odd lines of video data are being captured from the camcorder and stored in SRAM memory. This can be verified by the fact that **WE_CAM, VID_LDL** and **VID_LDH** are active during this period of time.

Let us zoom in closer to get a better understanding of how the data transfer takes place. Press **Zoom** on the analyzer and the result should look similar to **Figure 64**.

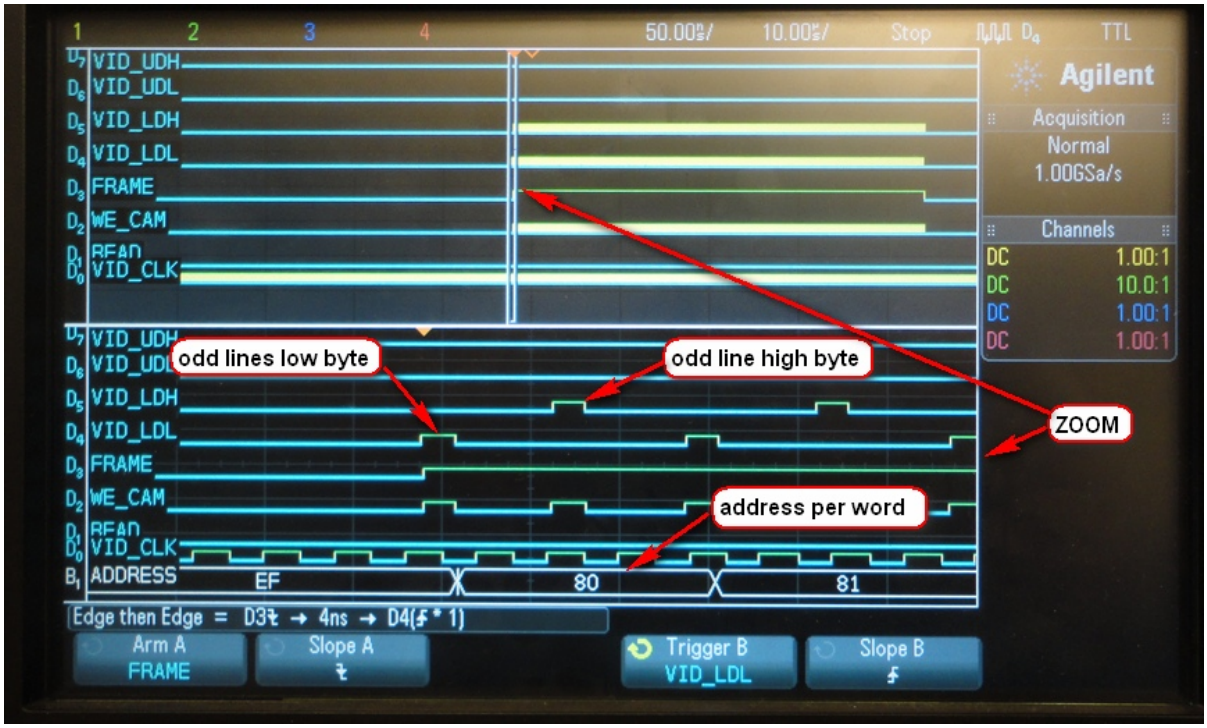


Figure 64 - Odd line enable zoomed in

Here we see that for every single address two 8 bit video values are sent from the video source to the on chip memory. In this case it is the odd line.

Open up your Verilog code and update the pin assignments according to **Figure 65**. Then recompile the Verilog code and download it to DE1-SoC.

```
232   ////////////////////////////////////////////////
233   // outputs to debug circuit gpio(0)       /////
234   ////////////////////////////////////////////////
235
236      assign gpio[0] = vid_clk;
237      assign gpio[1] = read;
238      assign gpio[2] = we_cam;
239      assign gpio[3] = oddeven;
240      assign gpio[4] = read_ll;
241      assign gpio[5] = read_lh;
242      assign gpio[6] = read_hl;
243      assign gpio[7] = read_hh;
244
245      assign gpio[15:8] = address_mot[7:0];
```

Figure 65 - Pin out assignments for read from memory to the monitor

Rename the following labels according to **Table 15** on the MS0-X-3024A scope.

40 pin HeaderJP0	Pin assignment	Digital lead default name	Rename label
GPIO[3]	Y18	D3	ODDEVEN
GPIO[4]	AK16	D4	READ_LL
GPIO[5]	AK18	D5	READ_LH
GPIO[6]	AK19	D6	READ_HL
GPIO[7]	AJ19	D7	READ_HH

Table 15 - New pin out assignments for the GPIO header

- Set delay to **0.0**s.
- Set **horizontal** frequency to **10.0**u/s.
- Press **Trigger**.
- Select **Edge** trigger.
- Set **ODDEVEN** as Trigger.
- Set **Falling Edge** as slope.
- Press **Single** to trigger event.

The result should look similar to **Figure 66**. We can make several observations:

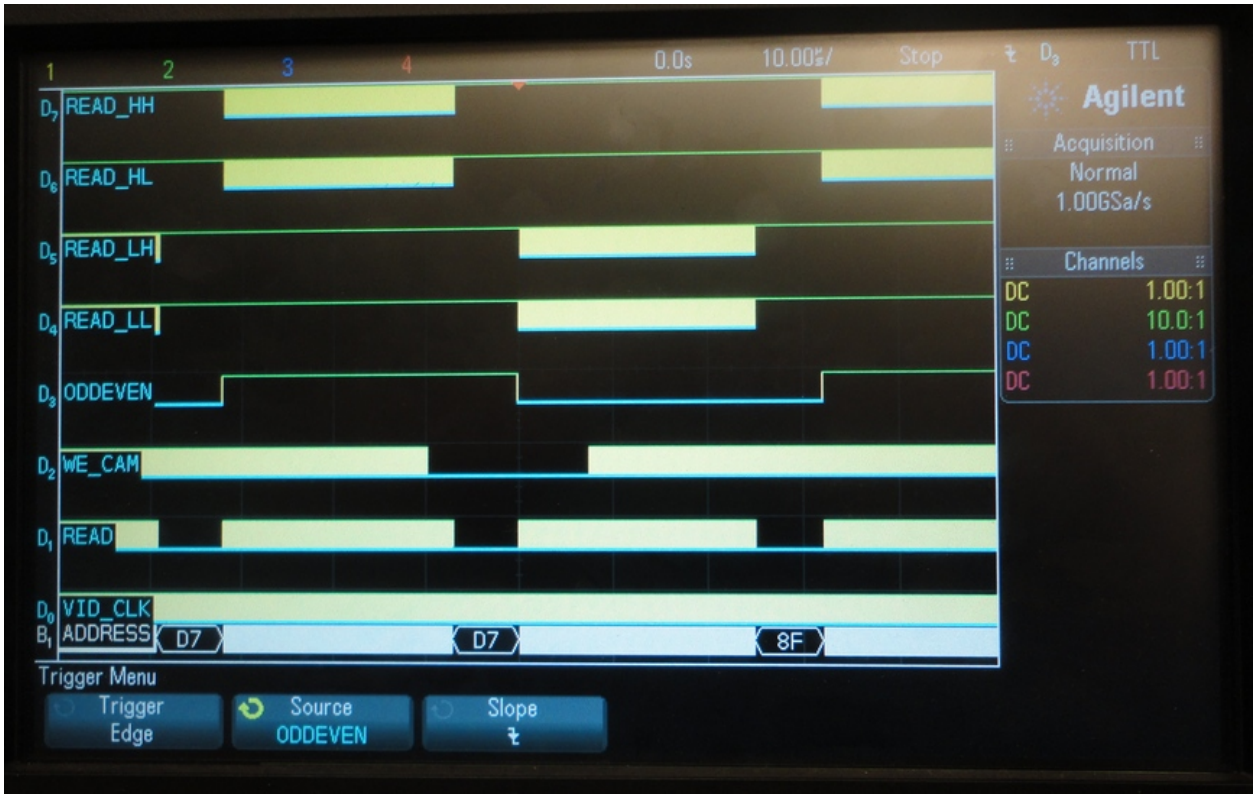


Figure 66-Sending data from the on chip memory to the monitor

- This is a trigger capture of video data being sent from the on chip memory to a display source, such as a monitor

- When **ODDEVEN** is active low, **READ_LL** and **READ_LH** are active and **READ_HL** and **READ_HH** remain active high. This means that odd lines of 8 bit video data are being sent to the monitor from the on chip memory
- When **ODDEVEN** is active high, **READ_HL** and **READ_HH** are active and **READ_LL** and **READ_LH** remain active high. This means that even lines of 8 bit video data are being sent to the monitor from the on chip memory

Let's look closer at the signals. Press **Zoom,** the result should look similar to **Figure 67.**

Here we see that for every single address there are two 8 bit video values sent from the on chip memory to the monitor. In this case it is the odd line.

Figure 67-Zoomed in oddeven odd lines

- Turn off **Zoom**
- Press **Trigger**
- Set **Rising Edge** as slope
- Press **Single** to trigger event
- Turn **Zoom** on

The result should look similar to **Figure 68**.

Figure 68- Zoomed in oddeven even lines

Here we see that for every single address there are two 8 bits video values sent from the on chip memory to the monitor. In this case it is the even line.

This concludes the tutorial. You are now familiar with how to get video data from a video source, store it in memory as a frame, and then display it to a video source.

As explained earlier, the video we just generated is a very simply video. To improve on it, 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 links:

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

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

Read starting on page 6 – Background

The DE1-SOC board uses the YCrCb 4:4:4 format. See **Figure 69** below.

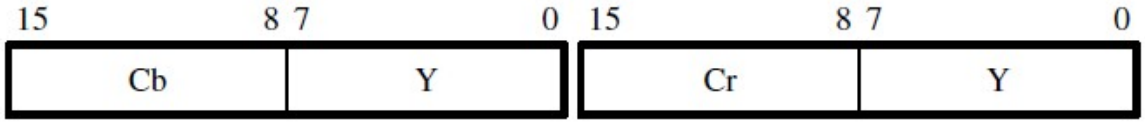


Figure 69-YCrCb video format

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. This file is from the Terasic home page, under projects DE1-SoC

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

Select-DE1-SOC>DESL Online Tutorials> DE1_SoC_TV.zip.

This project is already compiled, only the zip files and the DE1_SoC_TV.sof file need to be downloaded

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

PS2 interface

There are two common interfaces that can connect to the PS2 interface: the keyboard and the mouse. Both use different protocols to retrieve data information.

For further information on the PS2 interface follow this link:

http://www-ug.eecg.utoronto.ca/desl/manuals/ps2.pdf

Connecting a Keyboard to the PS2 interface (Part1)

The link above gives very good insight into the PS2 keyboard. We will now concentrate on the timing circuit. In this tutorial we will use both Quartus and the MSO-X-3024A, in order to investigate and get a better understanding of the PS2 protocol for the keyboard. **Figure 70** is a timing diagram, where by data is sent serially from a keyboard to a device, such as a computer. We can make the following observations:

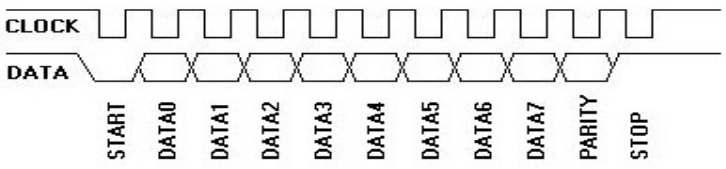


Figure 70-Timing circuit for the PS2 keyboard interface

- Both the **clock** and **data** signal lines are active high during an inactive state

- When the **data** signal goes active low, the **clock** signal will go low soon after. This indicates that **data** initiated the data transfer.
- There are a total of 8 data bits [0-7] generated serially. Each data bit is valid when the **clock** pulse is active low. (**Note** that the transfer follows the little Endian protocol of low bytes to high bytes)
- Data values can be high or low, depending on which character on the keyboard is pressed. More will be explained about this later.
- After all the data has been sent, a **parity** bit is determined. By default parity is odd. So if the count of active high data values [0-7] is even, then the parity bit would be 1 in order to make the parity odd. The opposite is true if the data count [0-7] is odd.
- Finally there is a **Stop** bit, which is active high and indicates that all the data has been sent from the keyboard to the device.

Let us investigate this by downloading the following zip file:

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

Select –DE1-SoC>DESL Online Tutorials>keyboard.zip.

Create a directory and unzip the file. Using the Quartus **new project wizard** create a new project called **keyboard**. Compile the project and download the program to the DE1-S0C.

- Connect digital leads 0-9 from the MSO-X-3024A to pins 0-9 on the JP1 (GPIO-0). For this lab we will use only 0-2 but in the next part of the lab we will use all 10 digital leads. Use **Figure 71** as a reference on how the pins should be hooked up for both ground leads (black) and digital leads (gray).

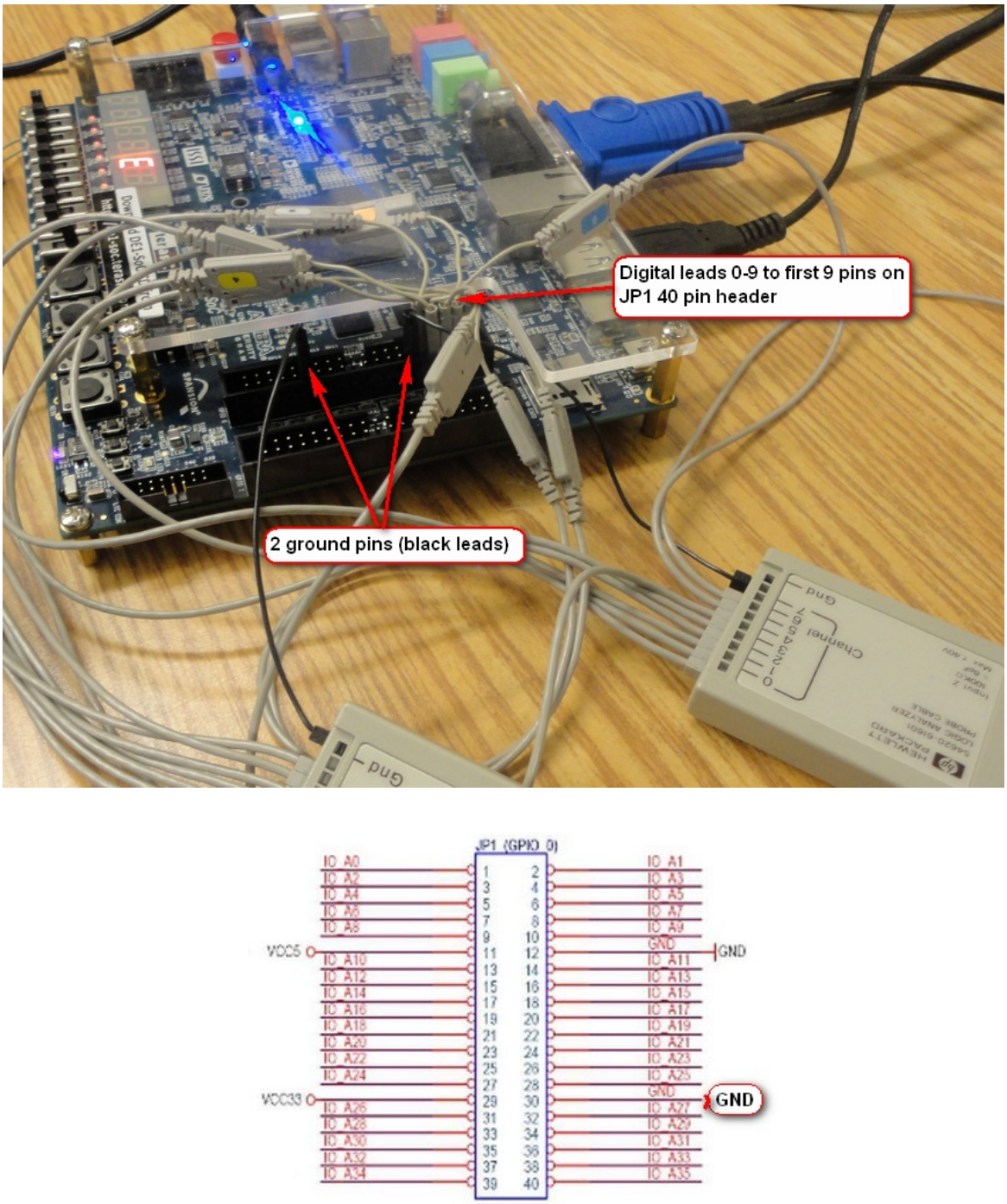


Figure 71- Digital pin assignments for this tutorial

- Press **Digital**
- Enable **D0-D2** and turn off all the other digital lines
- Set **Scale** to **Large**

- Press **Label** and rename D0-D2 according to column 4 in **Table 16**

40 pin HeaderJP0	Pin assignment	Digital lead default name	Rename label
GPIO[0]	AC18	D0	Clk_50
GPIO[1]	Y17	D1	PS_CLK
GPIO[2]	AD17	D2	PS_DATA

Table 16-Pin assignments for the Keyboard tutorial

- Set delay to **423**u/s.
- Set **Horizontal** frequency to **200.0**u/s.
- Press **Trigger**.
- Select **Edge** trigger.
- Set **PS_CLK** as Trigger.
- Set **Falling Edge** as slope.
- Connect a keyboard to the PS2 connector on the DE1-SoC (Top left of board).
- Press **Single** to trigger an event.
- Press the **ESC** key on the keyboard connected to the DE1-SoC board.

The result should look similar to **Figure 72**. We can make several observations:

- The PS_clk signal has 10 clock pulses and 11 active low states. This coincides with the clock in **Figure 70**.
- The data pulse pattern **PS_DATA** represents the **ESC** key on the keyboard. We will verify this later in this tutorial.
- The active low clock pulse happens after the data signal decreases.
- All data values are valid when the clock signal is active low.
- Once the data remains active high, the clock signal will go active high shortly after.
- The results displayed on the scope in **Figure 72** verify the diagram in **Figure 70**.

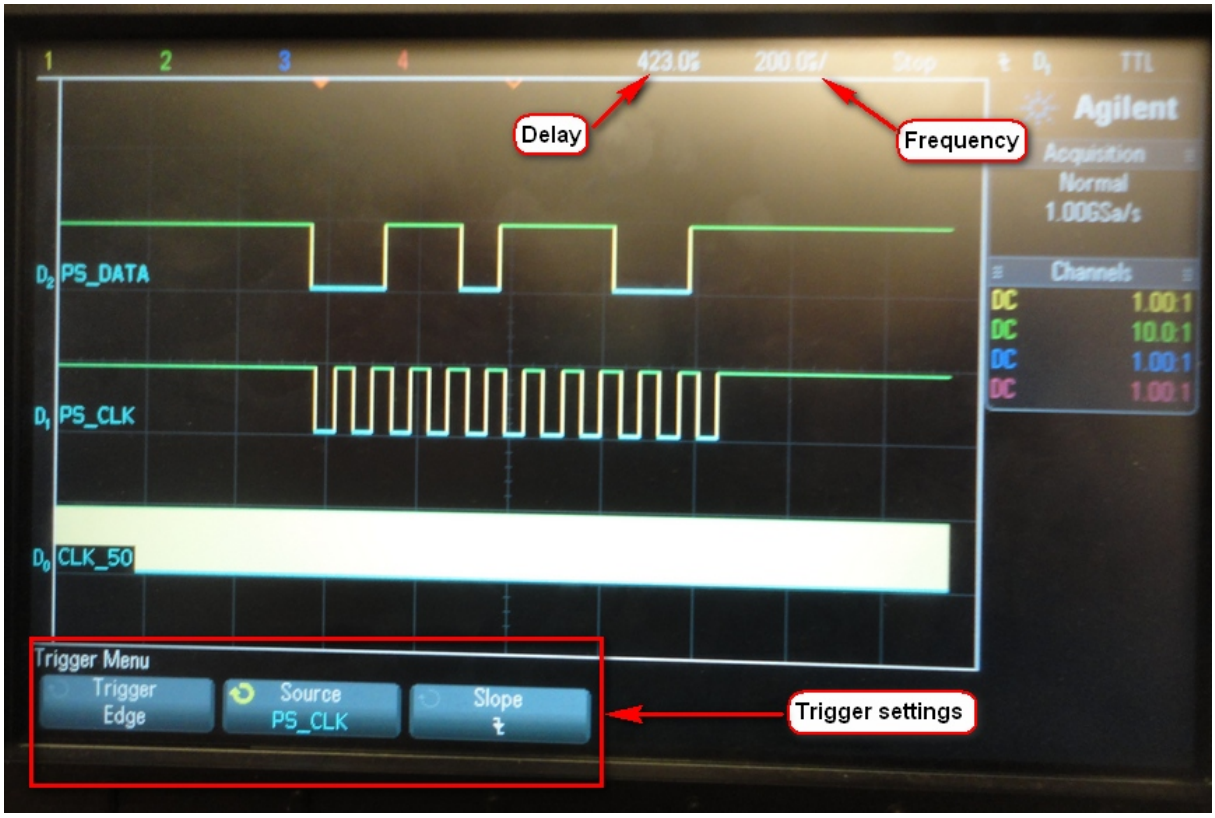


Figure 72-Sample of a data transfer from a keyboard to a device

Let us further investigate by zooming in and looking more closely at the signals.

- Change **Horizontal** frequency to **100.0**u/s.
- Press **Digital.**

The result should similar to **Figure 73**. We now have a better view of the data signal pattern **PS_DATA**. We can make the following observations:

- The first active low of the **PS_DATA** signal is the start of the transfer.
- The **Clock** signal goes active low shortly after the data line goes active low.
- This is followed by 8 active transitions for data. **Remember** that data is transferred data0 to data7 (low to high) so in this case the ESC key is [01101110], which is HEX value **76**(D7-D0). We call this the **key code** value for **ESC**.
- This is followed by the parity bit. **Remember** that parity must be **odd** and since we have an odd number of ones (5) in the data transfer, the parity bit remains low.
- Finally the data signal goes active high and shortly after that the clock signal goes active high. This indicates the transfer was successful and complete.

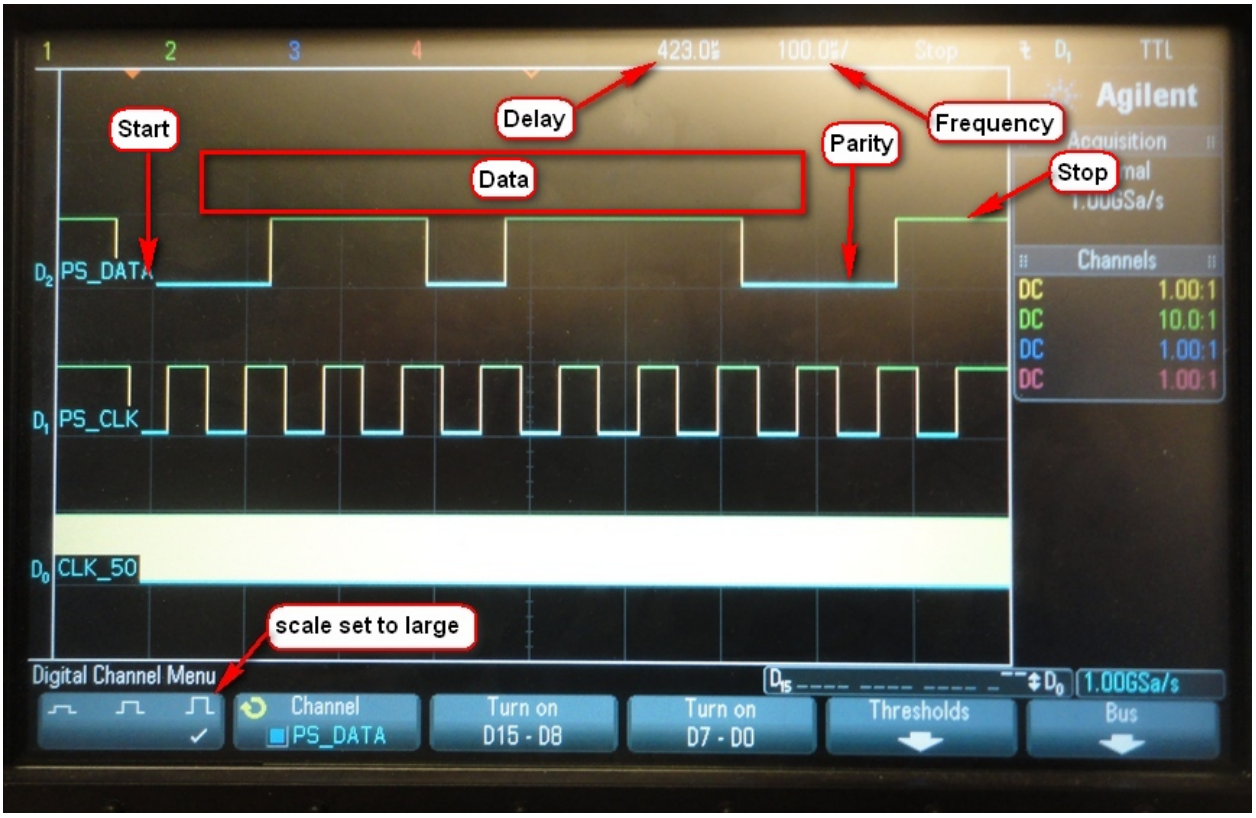


Figure 73- Zoomed in view of the data transfer

Now that we have an understanding of how each key value is transferred, we can create a table of the key codes. Below is a list of most of the key codes found on a standard PS2 keyboard. Refer to the last page of the PS2 manual (page 72 of this tutorial) to get a full list of all the key codes.

Key codes			Key codes			Key codes			Key codes	
Key	HEX Value		Key	HEX value		Key	HEX value		key	HEX Value
A	1C		N	31		0	45		BACKSPACE	66
B	32		O	44		1	16		Right Arrow	E0 74
C	21		P	4D		2	1E		Left Arrow	E0 6B
D	23		Q	15		3	26		Up Arrow	E0 75
E	24		R	2D		4	25		Down Arrow	E0 72
F	2B		S	1B		5	2E		DELETE	E0 71
G	34		T	2C		6	36		Equal	55
H	33		U	3C		7	3D		Tab	0D
I	43		V	2A		8	3E		Insert	E0 70
J	3B		W	1D		9	46		Home	E0 6C
K	42		X	22		Space	29		End	E0 69
L	4B		Y	35		ESC	76		Page UP	E0 7D
M	3A		Z	1A		ENTER	5A		Page Down	E0 7A

Table 17- Most common key codes for the PS2 Keyboard

We can use **Table 17** as a reference and see if the results displayed on the MSO-X-3024A match the key codes pressed on the keyboard. Perform the following:

- Press **Single.**
- Press the **A** key on the keyboard.
- Decode the result, knowing the data is sent from low to high byte (little endian format).
- The result should look similar to **Table 18**. This table ignores the **start** and **stop** signals

Key	HEX value							
A	1C							
Data0	Data1	Data2	Data3	Data4	Data5	Data6	Data7	Parity
0	0	1	1	1	0	0	0	0

Table 18- Decoding the key code for A key on the keyboard

Try other keys on the keyboard and see if the results match the key codes in **Table 17**. This concludes part 1.

Part 2- Creating a serial decoder and displaying the keyboard key results

Now that we have an understanding of how data is retrieved from the keyboard we can write a Verilog routine that will decode the key code values and then display the results. The following tasks will need to be fulfilled:

- A serial shift register will need to be created, in order to store each 8 bit data value.
- A reference table will need to be created.
- We can use this reference table to determine which key has been pressed by performing a comparison.
- We can use the HEX display to show what key was pressed. Note that since the HEX display has a limited number of segments to create letters and numbers, we will create symbols for some of the keyboard characters.
- We can use a timing circuit to make sure only valid data is collected.

Let us investigate by downloading this zip file:

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

Select –DE1-SoC>DESL Online Tutorials>keyboard_full.zip.

Create a directory and unzip the file. Using the Quartus **new project wizard** create a new project called **keyboard**. Compile the project and download the program to the DE1-S0C board. Open up keyboard.v and scroll down to line 45. See **Figure 74.**

```
38  ////////////////////////////////////////////////////////////////////////////
39  // characters and symbols created for displaying values on Hex display //
40  // some require two HEX displays others don't                          //
41  // Note not all key codes from keyboard are here. Some are missing     //
42  // from this list,you may create them by adding them                   //
43  ////////////////////////////////////////////////////////////////////////////
44
45      parameter HEX_0 = 14'b11111111000000;    // zero
46      parameter HEX_1 = 14'b11111111111001;    // one
47      parameter HEX_2 = 14'b11111110100100;    // two
48      parameter HEX_3 = 14'b11111110110000;    // three
49      parameter HEX_4 = 14'b11111110011001;    // four
50      parameter HEX_5 = 14'b11111110010010;    // five
51      parameter HEX_6 = 14'b11111110000010;    // six
52      parameter HEX_7 = 14'b11111111111000;    // seven
53      parameter HEX_8 = 14'b11111110000000;    // eight
54      parameter HEX_9 = 14'b11111110011000;    // nine
55      parameter HEX_a = 14'b11111110001000;    // a
56      parameter HEX_b = 14'b11111110000011;    // b
57      parameter HEX_c = 14'b11111111000110;    // c
58      parameter HEX_d = 14'b11111110100001;    // d
59      parameter HEX_e = 14'b11111110000110;    // e
60      parameter HEX_f = 14'b11111110001110;    // f
61      parameter HEX_g = 14'b01100101000110;    // g
62      parameter HEX_h = 14'b11111110001001;    // h
63      parameter HEX_i = 14'b10001101110110;    // I
64      parameter HEX_j = 14'b11111101100000;    // J
65      parameter HEX_k = 14'b01101010001011;    // K
66      parameter HEX_l = 14'b11111111000111;    // L
67      parameter HEX_m = 14'b10110001001100;    // M
68      parameter HEX_n = 14'b11100010001011;    // N
69      parameter HEX_o = 14'b11100001000110;    // O
70      parameter HEX_p = 14'b11111110001100;    // P
71      parameter HEX_q = 14'b11101111000000;    // Q
72      parameter HEX_r = 14'b01001000001110;    // R
73      parameter HEX_s = 14'b01100100010110;    // S
74      parameter HEX_t = 14'b10011101111110;    // T
75      parameter HEX_u = 14'b11111111000001;    // U
76      parameter HEX_v = 14'b11011011011011;    // V
77      parameter HEX_w = 14'b11000011000011;    // w
78      parameter HEX_x = 14'b01101010010011;    // x
79      parameter HEX_y = 14'b01111010011011;    // y

80      parameter HEX_z = 14'b01101000100110;    // Z
81      parameter HEX_en = 14'b01010110000110;   // enter
82      parameter HEX_ec = 14'b10001100000110;   // ESC
83      parameter HEX_bs = 14'b00100100000011;   // back space
84      parameter left =  14'b01100000111111;    // left arrow
85      parameter right = 14'b01111110000110;    // right arrow
86      parameter up   =  14'b10011001011000;    // up arrow
87      parameter down =  14'b10000111100001;    // down arrow
88      parameter off   = 7'b1111111;      // display off
89
```

Figure 74- Parameters representing key code characters

These parameters represent what each keyboard character will look like on the HEX display. In this Verilog program we are using the first two HEX displays to create the character or symbol. To further explain this press the **4** and **G** keys

Figure 75-Two characters displayed on the 2 HEX displays

Figure 75 shows the results for both cases. Note **4** looks as it should but **G** is created using both HEX0 and HEX1.

- Scroll down to line 99. See **Figure 76.**
- Lines 99 to 129 represent a state machine, which will serially shift data from the keyboard to the FPGA.
- **CNT1** keeps track of the value of the **counter**. If it exceeds digital value **11** then the state will change from active low to active high.
- Lines 105 to 128 are a counter that freely counts up values from 0 to when **CNT1** changes from an active low to active high state.

Let's use the MSO-X-3024A to look at these signals and verify whether the above does occur. From **Part 1**, digital leads 0 to 9 should have been connected. All that is left to do is rename the pin assignments, by following the procedure below:

- Press **Digital.**
- Set **Scale** to **medium.**
- Press **Bus**. Enable **bus 1** and set **D5-D9** to represent the bus.
- Enable **D0-D4** and unselect all the other digital lines.
- Press **Label** and rename D0-D9 according to **column 4** in **Table 19**.
- Press **Trigger**.
- Change source to **CNT**.
- Leave slope as **Falling Edge**.
- Press **Single**
- Press any key on the keyboard.

```
91  ////////////////////////////////////////
92  /////// counters enables       ////////
93  ////////////////////////////////////////
94
95  wire cnt1 = (counter >= 8'd11 )? 1'b1 : 1'b0;
96
97  ////////////////////////////////
98  /// clock for PS2      //////
99  ////////////////////////////////
100
101     always @ (negedge ps_clk or posedge cnt1 )
102
103     begin
104
105       if (cnt1)
106       begin
107          counter <= 0;
108          cnt <= 1;
109       end
110
111          else
112
113       begin
114
115          counter <= counter + 1;
116          cnt <= 0;
117
118       end
119    end
120
121
```

Figure 76- State machine that serially shifts data from the keyboard to the FPGA

40 pin HeaderJP0	Pin assignment	Digital lead default name	Rename label
GPIO[0]	AC18	D0	VALID
GPIO[1]	Y17	D1	PS_CLK
GPIO[2]	AD17	D2	PS_DATA
GPIO[3]	Y18	D3	CNT
GPIO[4]	AK16	D4	CNT1
GPIO[5]	AK18	D5	COUNTER0
GPIO[6]	AK19	D6	COUNTER1
GPIO[7]	AJ19	D7	COUNTER2
GPIO[8]	AJ17	D8	COUNTER3
GPIO[9]	AJ16	D9	COUNTER4

Table 19-The digital lead assignments for Part 2

Depending on which key is pressed, the result will look similar to or the same as **Figure 77**.

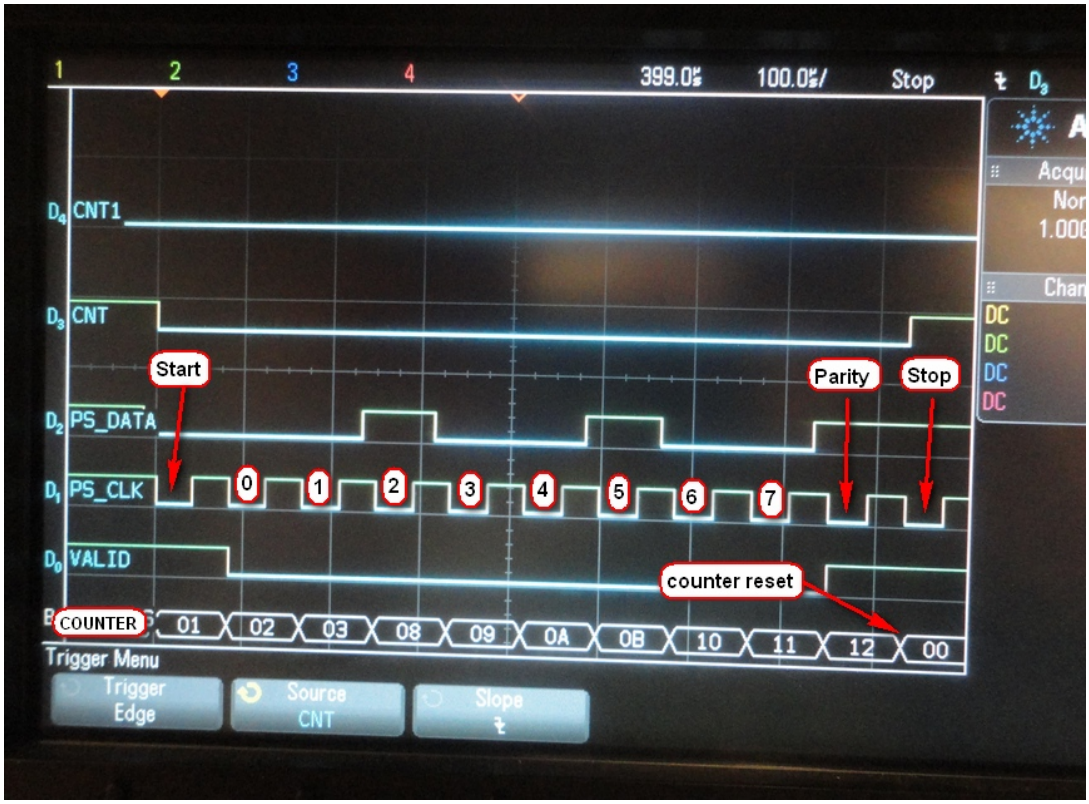


Figure 77-Trigger results for the data transfer

From **Figure 77** we can verify several things:

- If the signal **COUNTER** is less than or equal to digital value **12,** then the counter keeps counting.
- As soon as the signal **COUNTER** is greater than digital value 12, the counter gets reset to zero.
- **Note** that **CNT1** does not go active high because when the counter is reset, **CNT1** immediately returns back to active low.
- The signal **VALID** shows when valid data can be retrieved.

This concludes **Part 2** of this tutorial. If you are feeling ambitious, rather than displaying the keyboard values on the HEX display, you can try and write a Verilog program to display them to a monitor. Use the video tutorial to help you do this.

Connecting the PS2 Mouse Interface to the DE1-SoC

Unlike the keyboard, when you plug the PS2 mouse interface into the DE1-SoC board it does not automatically work. The mouse must be programmed before it will transmit information. After being programmed it will send 3 bytes of data, the format of which can be seen in **Figure 78**.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 0	Y overflow	X overflow	Y sign	X sign	Always 1	Middle button	Right button	Left button
Byte 1	X movement							
Byte 2	Y movement							

Figure 78 - 3 byte data packets sent by the mouse

- **Byte 0** gives information about the button pushes and whether the X or Y motion is a positive or negative value. More will be explained about this later in this tutorial.
- **Byte 1** gives information about the movement of the X motion. This value is a 9 bit 2's compliment value, so the values are in the range of -255 to +255.
- **Byte 2** gives information about the movement of the Y motion. This value is a 9 bit 2's compliment value, so the values are in the range of -255 to +255.

Before we can examine what a streaming packet from the mouse looks like, we need to create a Verilog program to initialize the streaming mode. By default, when the mouse is plugged into the PS2 port it does not get initialized. To make the mouse go into streaming mode we need to write the HEX FA to the mouse by performing the following:

- Create a slower clock by using the 50 MHz clock as a reference and then dividing it to get a 50 KHz clock. This is in the range of clock timing for the mouse.
- Generate a start pulse.
- Create a shift register, in order to send and receive serial data from the mouse.
- Make sure valid data has been sent. As previously explained, the HEX FA needs to be sent to the mouse to initialize streaming mode.
- Create a tri state buffer that allows both data and clock signals to be sent and received.

Let's investigate by downloading the following zip file:

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

Select –DE1-SoC>DESL Online Tutorials>mouse_part1.zip.

Create a directory and unzip the file. Using the Quartus **new project wizard** create a new project called **mouse** and compile the project. Let's examine the code and see if all of the conditions above

have been met. While in Quartus, open up mouse.v and scroll down to line 45. You should see the lines of Verilog code found in **Figure 79**.

- In line 45 we set the reference frequency to 50 MHz.
- In line 46 we set the desired frequency to 50 KHz.
- In lines 53-75 we created a state machine, which constantly divides the 50 MHz clock until it equals our desired clock. At that point the output signal, **CLK**, on line 73 is toggled, which in turn generates our 50 KHz clock.

```
43  ////////////////////////////////////////////////////
44
45  parameter clk_freq = 50000000;  // 50 Mhz
46  parameter ps2_freq = 50000;     // 50 Khz
47  parameter mouse_stream = 9 'b011110100; // streaming mouse with parity F4
48
49 //////////////////////////////////////////////
50  ////// clock divider from 50 MHz to 50 KHz ///
51  //////////////////////////////////////////////
52
53   always @ (posedge clk_50 or negedge reset)
54   begin
55       if (!reset)
56       begin
57          clk_div <= 0;  // counter
58          clk <= 0;      // frequency
59
60       end
61
62       else
63
64       begin
65
66          if (clk_div <  (clk_freq/ps2_freq) )  // keeps dividing until reaches desired frequency
67          clk_div <= clk_div + 1;
68
69
70          else
71          begin
72                clk_div <= 0;
73                clk <= ~clk; // 50 KHz clock used for ps2 clock circuit
74          end
75       end
76    end
77
```

Figure 79 – Code dividing the clock from 50 MHz to 50 KHz.

Now scroll down to line 83, where we are creating a state machine that will tell the mouse to start the clock. When send_enable (swt 2) is active high, the clk_count will increase. See **Figure 80**.

```
79    ////////////////////////////////////
80    /// serial clock state machine  ///
81    ////////////////////////////////////
82
83  always @(negedge reset or posedge clk ) begin
84      if (!reset) clk_counter =  4 'b1111;
85      else begin
86      if (send_enable==0)
87         clk_counter=0;
88         else
89         if ((clk_counter < 7'd9) & (clk_stop ==0)) clk_counter = clk_counter + 1;
90      end
91      end
92
```

Figure 80-Clock counter enabled and increasing.

Now scroll down to line 97. The Verilog code from lines 97-125 act as a shift register and with each increase of the clk_counter, a new value of PSCLK is added to the PS2_CLK line.

```
93    ////////////////////////////
94    // counter for PS2 clock  ///
95    ////////////////////////////
96
97  always @ (negedge reset or posedge clk ) begin
98
99    if (!reset) begin clk_stop = 0;  send_start = 1 ; PSCLK = 1;  end
100   else
101 case (clk_counter)
102
103       // begin clock load command pulse ~ 100 120 us
104
105       4'd0  : begin clk_stop = 0; send_start = 1; PSCLK = 1; send_data = 0; trigger_data = 0; end // send command
106
107       4'd1  : begin clk_stop = 0; send_start = 1; PSCLK = 0; send_data = 0; trigger_data = 0;end // active clock low
108
109       4'd2  : begin clk_stop = 0; send_start = 1; PSCLK = 0; send_data = 0; trigger_data = 0; end // active clock low
110
111       4'd3  : begin clk_stop = 0; send_start = 1; PSCLK = 0; send_data = 0; trigger_data = 0;end // active clock low
112
113       4'd4  : begin clk_stop = 0; send_start = 1; PSCLK = 1; send_data = 1; trigger_data = 1;end // await data transmission
114
115       4'd5  : begin clk_stop = 0; send_start = 0; PSCLK = 1; send_data = 1; trigger_data = 1; end // keep data enabled
116
117       4'd6  : begin clk_stop = 0; send_start = 0; PSCLK = 1; send_data = 1; trigger_data = 1; end // keep data enabled
118
119       4'd7  : begin clk_stop = 0; send_start = 0; PSCLK = 1; send_data = 1; trigger_data = 1; end // keep data enabled
120
121       4'd8  : begin clk_stop = 0; send_start = 0; PSCLK = 1; send_data = 1; trigger_data = 0; end // tri state data
122
123       4'd9  : begin clk_stop = 1; send_start = 0; PSCLK = 1; send_data = 1; trigger_data = 0; end // stop counter
124
125         endcase
126   end
```

Figure 81-Serial shift register of the PS2 Clock

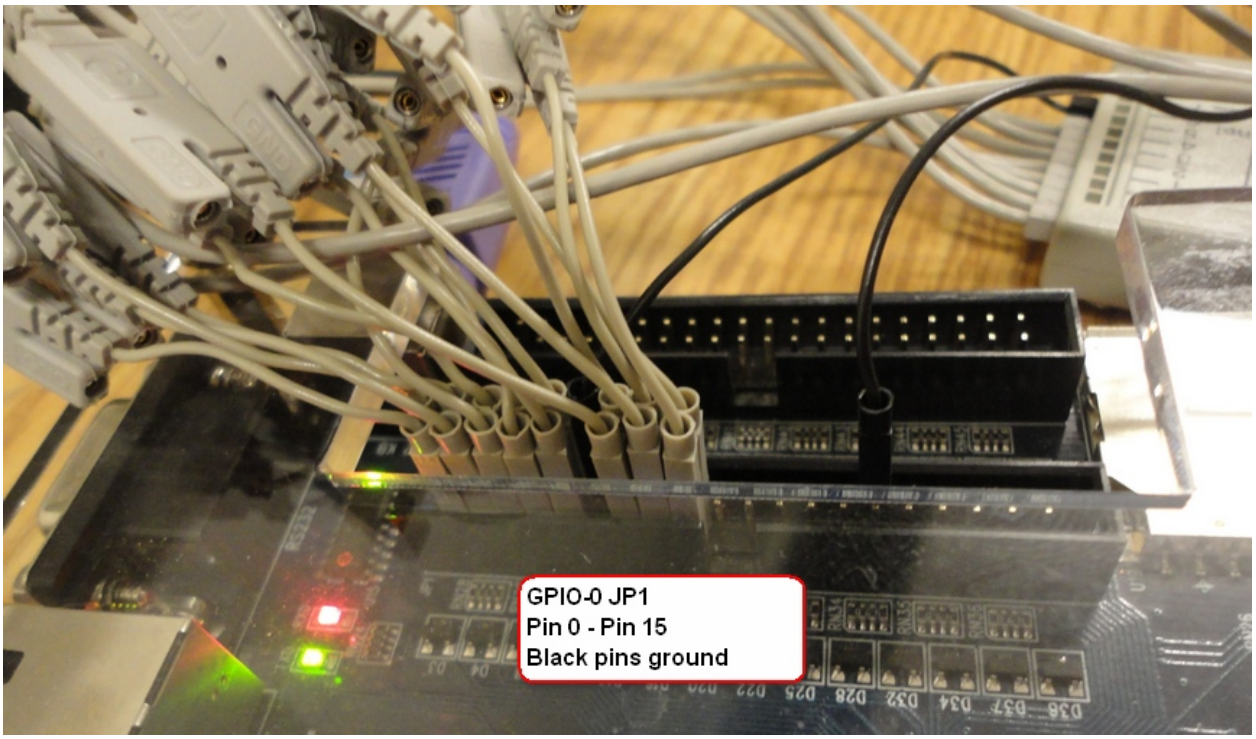
- The signal **PSCLK** represents the serial clock signal on the bus.
- The signal **send_data** is used to enable the serial data bus. When active high, the serial data can be transmitted.

- The signal **send_start** is used to determine whether the clock signal is tri state or not. When **send_start** is active low, the clock signal is tri state. See line 208 in **Figure 82**.
- The code in **Figures 80 and 81** are used to create the PS2 serial clock.
- The code from lines 133-201 is the combined state machine used to create the serial shift register and the tri state buffer for the PS2 serial bus. Depending on whether **send_recieved** or **trigger_data** is active high, the signal **PSO** is transmitted on the ps2_data signal. See line 209 in **Figure 82**. It will be up to you to verify this.
- The mouse will send the HEX value **FA,** which indicates that the mouse is enabled and the values of the X and Y movements.
- The serial initialization data must be sent with odd parity. Since the HEX value is HEX F4, it is odd already and the parity bit is 0. See line 47 in **Figure 79**.
- Now download the mouse.sof file to the DE1-SoC.

```
208     assign ps2_clk = send_start ? PSCLK : 1'bz;  //bi-directional
209     assign ps2_data = (send_recieved | trigger_data) ? PSO : 1'bz; //bi-directional
```

Figure 82 - PS2 Clk and data assignment statements

Connect digital leads 0-15 (gray) from the MSO-X-3024A to pins 0-15 on the JP1 (GPIO-0). Connect the ground leads (Black) to pins 12 and 30 on the JP1 (GPIO) 40 pin header. Use **Figure 83** as a reference on how the pins should be connected.



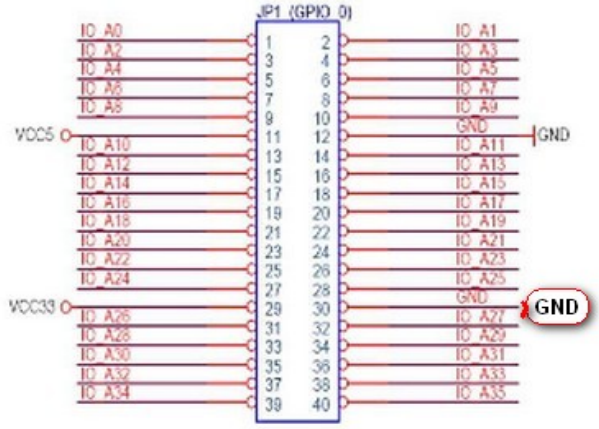


Figure 83 – Pin connections for the GPIO-0 40 pin header

- Press **Digital.**
- Enable **D0-D8.** Disable the other digital lines.
- Set **Scale** to be **medium**
- Press **Label** and rename D0-D8 according to column 4 in **Table 20**.
- Press **Digital**.
- Press **Bus.**
- Create a bus for D15-D9.
- Press **Back.**

40 pin HeaderJP0	Pin assignment	Digital lead default name	Rename label
GPIO[0]	AC18	D0	CLK
GPIO[1]	Y17	D1	PS2_CLK
GPIO[2]	AD17	D2	PS2_DATA
GPIO[3]	Y18	D3	SEND_REC
GPIO[4]	AK16	D4	SEND_START
GPIO[5]	AK18	D5	TRIG_DAT
GPIO[6]	AK19	D6	CLK_STOP
GPIO[7]	AJ19	D7	RESET
GPIO[8]	AJ17	D8	SEND_ENBLE
GPIO[9]	AJ16	D9	Address0
GPIO[10]	AH18	D10	Address1
GPIO[11]	AH17	D11	Address2
GPIO[12]	AG16	D12	Address3
GPIO[13]	AE16	D13	Address4
GPIO[14]	AF16	D14	Address5
GPIO[15]	AG17	D15	Address6

Table 20-Pin out assignments for the mouse tutorial

- Set **Delay** to **1.000**m/s.
- Set **Horizontal** frequency to **500.0**u/s.
- Make sure all **switches** on the DE1-SoC are down.
- Press **Trigger**.
- Select **Edge** as trigger.
- Set **SEND_ENABLE** as Trigger.
- Set **Rising Edge** as slope.
- Connect a PS2 mouse to the PS2 connector on the DE1-SoC (Top left on the board).
- Press **Digital**.
- Press **Single** to trigger an event.
- Move **switch 0** up on the DE1-SoC board. This resets all the counters
- Move the mouse. Notice that neither the **PS2_CLK** nor the **PS2_DATA** state values change on the scope. Look at the bottom right of **Figure 84**. This shows that the mouse is still reset.
- Move **switch 1** up on the DE1-SoC board

The result should look similar to **Figure 84**.

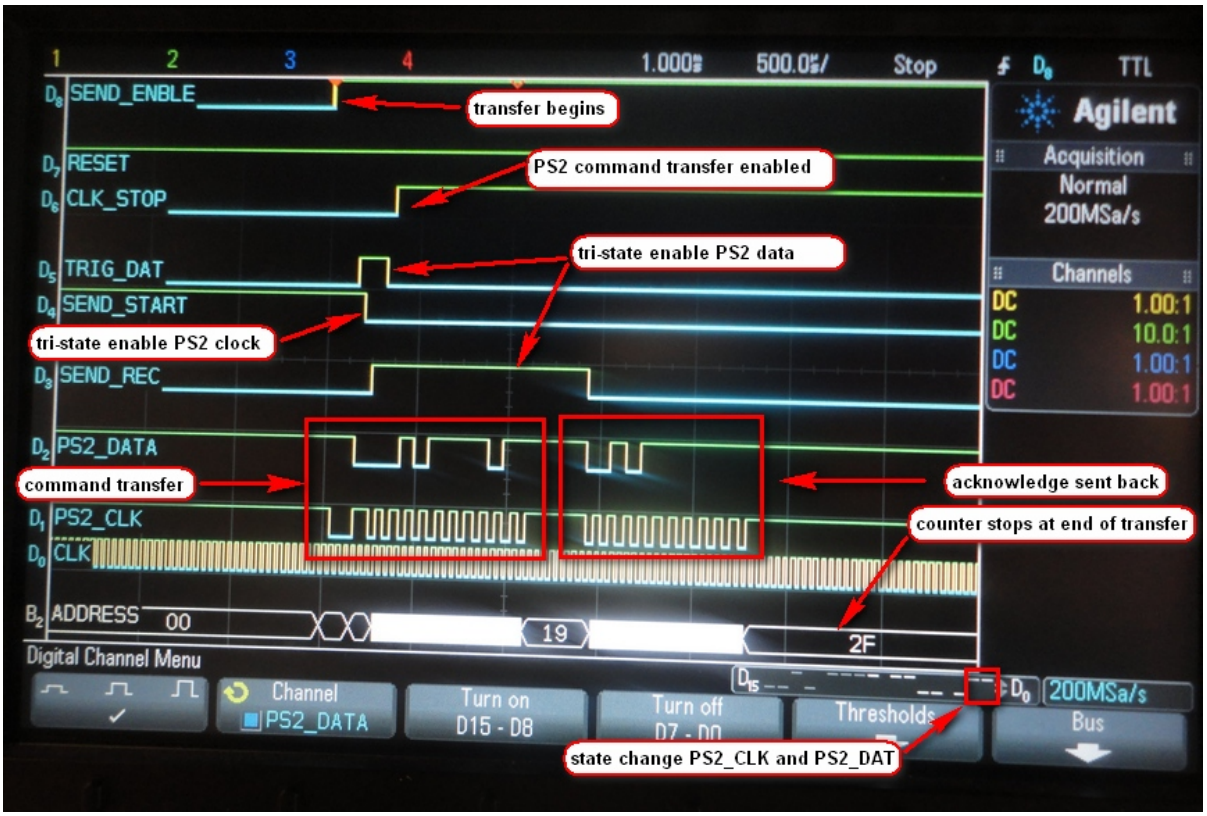


Figure 84 - Trigger command for the enabling of streaming data mode

We can make several observations from this trigger event:

- **TRIG_DAT** and **SEND_REC** are the direction enables for the PS2_DATA signal. When at least one of the signals is active high, serial data is sent to the mouse from the FPGA based on the value at PSO, otherwise the signal is tri-stated. See line 209 in **Figure 82** of the Verilog code for further verification.
- **SEND_START** is the direction enable for the PS2_CLK. When active high, serial data is sent to the mouse from the FPGA based on the value at PSCLK, otherwise the signal is tri-state. See line 208 in **Figure 82** of the Verilog code for further verification**.**
- You can see that the transfer has two parts to it: the command transfer and the acknowledgement transfer sent by the mouse to verify that the value that was sent.
- Press **Zoom**
- Set **Horizontal** to **100**u/s.
- The result should look similar to **Figure 85**.

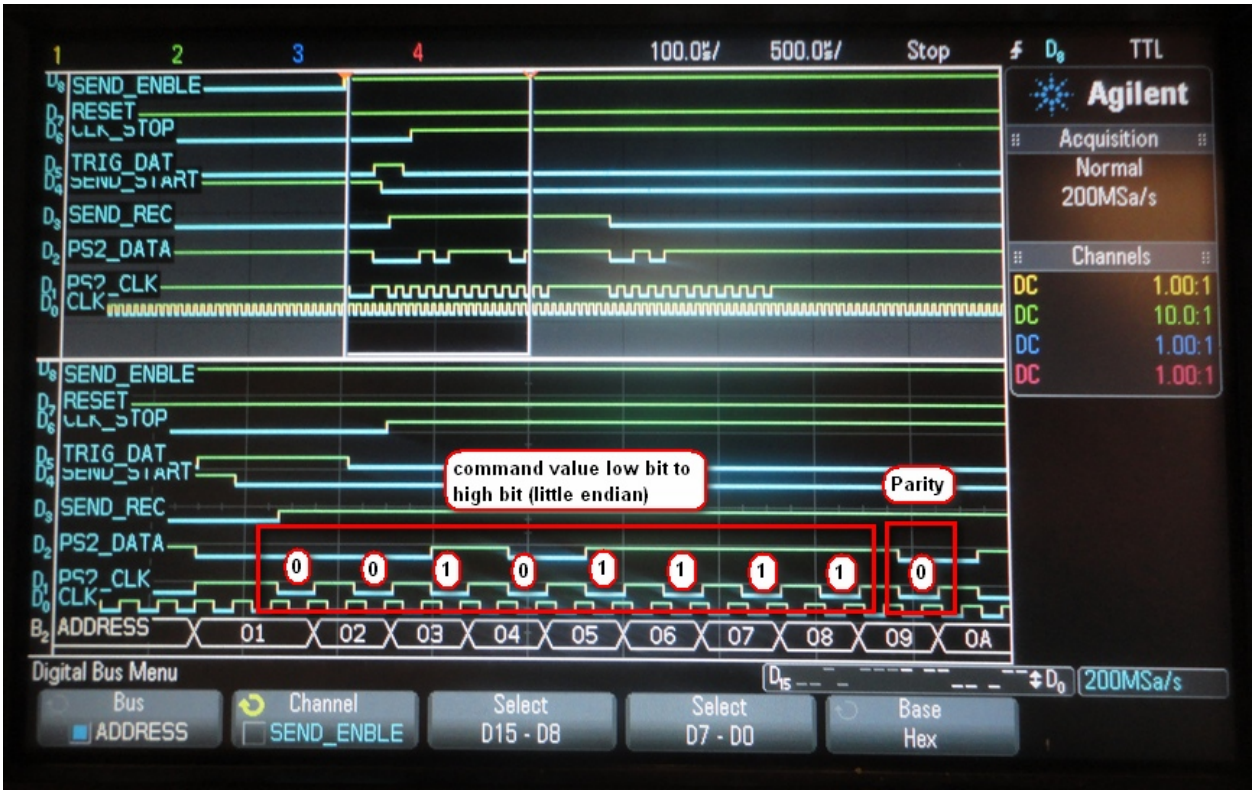


Figure 85 - Zoomed in transfer sequence

Here we get a better view of the counter.

- Notice that at count HEX **0A** the command transfer is over.
- As you can see the data transfer is low bit to high bit (Little Endian).

- Parity for all transfers is always odd. In this case, since the number of high bits is odd already, the parity bit is low.
- If you move the **Delay** dial to the end of the command transfer you will notice that the counter is at HEX 17, which equates to the digital value of 23.
- If you go to line 197 of the Verilog code you will notice that **send_stop** goes active high. This causes the counter to stop counting.
- At this point the command load has finished and the mouse is in stream mode

We can verify this by the fact that if you move the mouse both the **PS2-CLK** and **PS2-DATA** become active. Now we need to create a state machine that can capture the streaming data information. We will do this in Part 2 of this tutorial.

This concludes Part 1 of the mouse tutorial.

Part 2 - Creating a serial shift register to save the streaming data

In Part 1 we concentrated on creating a command serial transfer register to initialize the mouse. We used this register to send HEX F4 to activate streaming mode on the mouse.

In Part 2 we will be creating a serial transfer register to capture streaming data from the PS2 data bus. From **Figure 78** we know that three bytes are transferred. We can use this information to perform the following:

- Create a serial register to capture all three bytes.
- Extract data values from the serial register and store them in three registers
- The first register would be an 8 bit value and hold information about button pushes and the value of the X and Y movements.
- The second register would be a 9 bit 2's complement value, which determines the x movements of the mouse.
- The third register would be a 9 bit 2's complement value, which determines the y movements of the mouse.
- Create a state machine to detect if a button on the mouse had been pushed.
- Create a state machine to store two sequential x values, determine if the movement has changed, and whether the mouse is moving left or right.
- Create a state machine to store two sequential y values, determine if the movement has changed, and whether the mouse is moving up or down.
- Use the HEX display to display the movement of the mouse or what button was pushed.

Now that we have a list of goals, we can create the Verilog code to accomplish these tasks. Download the following zip file:

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

Select –DE1-SoC>DESL Online Tutorials>mouse_part2.zip.

Create a directory and unzip the file. Using the Quartus **new project wizard** create a new project called **mouse** and compile the project. Let's examine the code and see if all of the conditions above have been met. For this tutorial we have two Verilog files: **mouse.v** and **transmit.v**. Open up **transmit.v** and scroll down to line 116. You should see the lines of Verilog code found in **Figure 86**.

- It will only become active once the mouse has been programmed and streaming mode is working. If you look at line 118, **counter xy_counter** will not increase until **enable_send** is active high.
- The **sync_time** condition is determined on line 136. This states that as long as the **xy_counter** is less than the digital value 32 then it will keep increasing. Once the xy_counter reaches digital value 32, then the counter is reset and will repeat the sequence again.
- The two state machines just mentioned work in tandem to load the 3 bytes from each mouse transmission.

```
always @(negedge enable_send or posedge ps2_clk) begin

  if  (!enable_send)

  begin
  xy_counter = 7'b0; // reset counter
  end

  else begin

  if ( sync_time)
  xy_counter = 7'b0;

  else
  xy_counter = xy_counter + 1;
  end

  end


wire sync_time = (xy_counter < 7'd32)? 1'b0 : 1'b1; // end of each load cycle


always @ (negedge enable_send or negedge ps2_clk) begin

if (!enable_send) begin xyb_stop = 0 ; end
else
case (xy_counter)

     // begin load byte 0
     7'd0  : begin xyb_value[0] = ps2_data ; xyb_stop = 0; end // bit 0 - start

     7'd1  : begin xyb_value[1] = ps2_data ; xyb_stop = 0; end // bit 1 - valid bit 0 (left button)

     7'd2  : begin xyb_value[2] = ps2_data ; xyb_stop = 0; end // bit 2 - valid bit 1 (right button)

     7'd3  : begin xyb_value[3] = ps2_data ; xyb_stop = 0; end // bit 3 - valid bit 2 (middle button)

     7'd4  : begin xyb_value[4] = ps2_data ; xyb_stop = 0; end // bit 4 - valid bit 3 (always high)

     7'd5  : begin xyb_value[5] = ps2_data ; xyb_stop = 0; end // bit 5 - valid bit 4 (x sign)

```

Figure 86- The serial shift register that captures the 3 bytes of streaming data

Scroll down to line 227. The code will look similar to **Figure 87**.

```
////////////////////////////////////////////
/// store values in 8 bit register for  ////
/// button pushes middle left right     ////
/// x and y value                       ////
////////////////////////////////////////////


   always @ ( posedge clk ) begin

      if (xyb_stop) begin

       b_value [7:0] <= xyb_value [8:1];
       x_value [8:0] <= xyb_value [19:11];
       y_value [8:0] <= xyb_value [30:22];

       end
    end
```

Figure 87 – The 3 byte save registers

Here we are extracting the information from the 33 bit xyb_value register and putting them into three byte registers.

- The B_value register is 8 bits and gives information about which button on the mouse has been pressed and whether the value for X or Y is positive or negative. Notice that only the bits that refer to this register are extracted [bits 8 to 1]. More will be explained about the value later in this tutorial.
- The X_value register is 9 bits and gives information about movements of the mouse in the X direction (left or right). Notice that only the bits that refer to this register are extracted [bits 19 to 11].
- The Y_value register is 9 bits and gives information about movements of the mouse in the Y direction (up or down). Notice that only the bits that refer to this register are extracted [bits 19 to 11].

We need to create a state machine to detect if a button has been pushed. Scroll down to line 242 and observe **Figure 88**.

```
239     ////////////////////////////////////////////////
240     /// Testing to see which button was pushed   ///
241     ////////////////////////////////////////////////
242     always @  ( posedge clk or negedge enable_send ) begin
243
244     if (!enable_send) begin
245   // default setting
246       hex_out5 = dash;
247       hex_out4 = dash;
248       end
249
250       else
251
252     case (b_value [3:0])
253
254       4'b1100: begin  hex_out5 = HEX_11; hex_out4 = middle; end // middle button
255       4'b1010: begin  hex_out5 = HEX_11; hex_out4 = right; end  // right button
256       4'b1001: begin  hex_out5 = HEX_11; hex_out4 = left; end // left button
257       4'b1000: begin  hex_out5 = dash; hex_out4 = dash; end // no button push
258
259       endcase
260
261       end
```

Figure 88 – The button push detector state machine

From **Figure 78** we know that the first 3 bits of byte 1 indicate whether a button has been pushed. If all three bits are low then no button has been pushed but if any of them go high then a button has been pushed. This state machine takes care of all the conditions previously explained. HEX displays 4 and 5 on the DE1-SoC are used to display the result.

We need to get two consecutive X and Y values, in order to do a comparison. Scroll down to line 267, which will look similar to **Figure 89**.

```
266
267   always @ ( posedge xyb_stop ) begin
268
269     if (!reset) begin counter_xy = 2'b0; end
270
271        else
272
273        counter_xy = counter_xy +1;
274      end
275
276
277   ////////////////////////////////
278   ////   testing y movement    ////
279   ////////////////////////////////
280
281
282   always @   ( posedge clk or negedge enable_send ) begin
283
284     if  (!enable_send)
285   // default setting
286     begin
287     state_value_y = load_value_y1;
288     y_first = 8'b1;
289     y_second = 8'b1;
290     hex_out1 = dash;
291     hex_out0 = dash;
292     end
293
294     else
295
296     case (state_value_y)
297     load_value_y1: //000
298   // load first y value
299     begin
300
301      if (!counter_xy[0])
302      y_first = y_value;
303      else
304      state_value_y = load_value_y2;
305      end
```

Figure 89 - X_value two byte capture

- Here notice that **xyb_stop** is used as a clock. On line 214 of the **transmit.v** code you will notice that if **xyb_stop** goes from active low to active high, then **counter_xy** increases but only if **reset** is active high. When **xyb_stop** goes active high, then the three byte transfer is complete. It is at this point that valid values can be loaded into the X_value and Y_value registers.
- Scroll down to line 282. Here we have a state machine that will obtain two consecutive y_values.
- If you scroll down to line 376 you will notice that there is an identical state machine, which will obtain two consecutive x_values.

There is more to the state machine then what was previously mentioned but before we continue let's look at the previously mentioned signals on the MSO-X-3024A scope. All pins from Part 1 should still be connected. If not, reconnect them using **Figure 83** as a reference.

- Press **Digital**.
- Enable D0–D2 and disable D3-D15
- Press label and rename D0-D2 as indicated in column 4 of **Table 22**
- Set **Scale** to **large**.
- Press **Bus.**
- Enable **BUS 1**. Select D3-D9
- Enable **BUS2**. Select D10-D15.
- Re-Label **BUS1** to X_VALUE. See column 4 in **Table 22** as a reference
- Re-label **BUS2** XY_COUNTER. See column 4 in **Table 22** as a reference.

40 pin HeaderJP0	Pin assignment	Digital lead default name	Rename label
GPIO[0]	AC18	D0	XYB_STOP
GPIO[1]	Y17	D1	PS2_CLK
GPIO[2]	AD17	D2	PS2_DATA
GPIO[3]	Y18	D3	X_VALUE
GPIO[4]	AK16	D4	X_VALUE
GPIO[5]	AK18	D5	X_VALUE
GPIO[6]	AK19	D6	X_VALUE
GPIO[7]	AJ19	D7	X_VALUE
GPIO[8]	AJ17	D8	X_VALUE
GPIO[9]	AJ16	D9	X_VALUE
GPIO[10]	AH18	D10	XY_CNTER
GPIO[11]	AH17	D11	XY_CNTER
GPIO[12]	AG16	D12	XY_CNTER
GPIO[13]	AE16	D13	XY_CNTER
GPIO[14]	AF16	D14	XY_CNTER
GPIO[15]	AG17	D15	XY_CNTER

Table 21 - Pin assignments for Part 2 of the mouse tutorial

- Press **Trigger.**
- Set **XYB_STOP** as Trigger.
- Set **Rising Edge** as slope.
- Set **Delay** to **-1.500**m/s
- Set **Horizontal** to **500**u/s
- Download the mouse.sof file to the DE1-SoC board
- Make sure all switches on the DE1-SoC are down
- Move **switch 0** up (This resets all the counters to zero)
- Press **Single** to trigger an event.
- Move **switch 1** up (This loads the command register with **FA** and enables streaming mode)
- Move the mouse in any direction.

The result should look similar to F**igure 90.**

Figure 90 – The trigger event in streaming mode

Here we can see that each streaming transfer is 3 bytes. We can make several observations:

- When **XYB_STOP** is active low, streaming data is transferred from the mouse. We can use this to enable the **XY_COUNTER**.
- **XY_COUNTER** is reset to zero at the end of the transfer. It will begin counting again when the next transfer occurs and **XYB_STOP** goes low again.
- Move the mouse to the right and notice that the active state signals of X_VALUE (Digital pins [9-3]) for the most part are active low, except for (possibly) the lower three.
- Now move the mouse to the left and notice that the active state signals of X_VALUE (Digital pins [9-3]) for the most part are active high, except for (possibly) the lower three bits.
- From this we can conclude that when the mouse is moving in a positive (right) direction the data values are positive.
- If the mouse moves in a negative (left) direction the data values are negative.

Earlier we told you that the 9 bit movement value was 2s compliment. So since this is a 9 bit value, 2^9= 512. Therefore, the range of values goes from -255 to +255.

- Any value from -255 to 0 is a left movement of the mouse, so bit 8 is always 1.

- Any value from 0 to 255 is a right movement of the mouse, so bit 8 is always 0.
- If the first and second movement values [bit 8] are active low, then the movement is going right
- If the first and second movement values [bit 8] are active high, then the movement is going left
- We can use this information to show the direction that the mouse is moving.
- Scroll down to line 411 of the **transmit.v** code. See **Figure 91**.
- Here we have the 4 possible conditions that can happen with each change in movement of the mouse
- Both x_first and x_second are active high (left movement)
- X_first is low and x_second is active high (left movement)
- Both x_first and x_second are active low (right movement)
- X_first is high and x_second is active low (right movement)

```
410  ////////////////////////////////////////////////////////
411      test_sign_x://010
412  // compare to find direction
413      begin
414
415      if (x_first[8] & x_second[8]) // both high values
416       state_value_x = compare_left;
417       else if
418        (!x_first[8] & x_second[8] ) // first low and second high
419        state_value_x = compare_left;
420        else if
421        (x_first[8] & !x_second[8]) // first high and second low
422        state_value_x = compare_right ;
423        else if
424        (!x_first[8] & !x_second[8]) // both low values
425        state_value_x = compare_right;
426
427        else
428
429        state_value_x = test_sign_x; // once condition must be met
430
431        end
432
433  ////////////////////////////////////////////////////////
```

Figure 91 - Test and comparison for left or right movements

Once the result has been determined, it will be shown on the HEX display. The same concept can be applied for Y movements of the mouse. Scroll down to line 314, where you will find a similar state machine but this one determines vertical movements of the mouse.

The result on the display will look similar to **Figure 92**

```
442 ////////////////////////////////////
443       compare_left: // 110
444 // display value on HEX display
445       begin
446
447       hex_out3 = left;
448       hex_out2 = HEX_15;
449       state_value_x = no_change_x;
450       end
451 ////////////////////////////////////
```

```
348 ////////////////////////////////////
349       compare_down: // 100
350 // display HEX value
351       begin
352
353       hex_out1 = HEX_13;
354       hex_out0 = middle;
355       state_value_y = no_change_y;
356       end
357 ////////////////////////////////////
```


```
442 ////////////////////////////////////
443       compare_left: // 110
444 // display value on HEX display
445       begin
446
447       hex_out3 = left;
448       hex_out2 = HEX_15;
449       state_value_x = no_change_x;
450       end
451 ////////////////////////////////////
```

```
339 ////////////////////////////////////
340       compare_up: //011
341 // display HEX value
342       begin
343       hex_out1 = you;
344       hex_out0 = pee;
345       state_value_y = no_change_y;
346       end
347
348 ////////////////////////////////////
```


```
433 ////////////////////////////////////
434       compare_right: // 101
435 // display value on HEX display
436       begin
437       hex_out3 = right;
438       hex_out2 = tee;
439       state_value_x = no_change_x;
440       end
441
442 ////////////////////////////////////
```

```
339 ////////////////////////////////////
340       compare_up: //011
341 // display HEX value
342       begin
343       hex_out1 = you;
344       hex_out0 = pee;
345       state_value_y = no_change_y;
346       end
347
348 ////////////////////////////////////
```

```
433  ////////////////////////////////////
434        compare_right: // 101
435  // display value on HEX display
436        begin
437        hex_out3 = right;
438        hex_out2 = tee;
439        state_value_x = no_change_x;
440        end
441
442  ////////////////////////////////////
```

```
339  ////////////////////////////////////
340        compare_up: //011
341  // display HEX value
342        begin
343        hex_out1 = you;
344        hex_out0 = pee;
345        state_value_y = no_change_y;
346        end
347
348  ////////////////////////////////////
```

Figure 92 - HEX values describing the direction of the mouse

- HEX displays 0 and 1 are used to display the result of Y movements of the mouse
- HEX displays 2 and 3 are used to display the result of X movements of the mouse
- HEX displays 4 and 5 display the button pushes. If nothing is pressed then the display shows dashes, **br** (right button) , **bm** (middle button) and **bl** (left button)

Note this is for broad directional movements of the mouse and only takes the most basic movements of the mouse into consideration. More involved comparison statements would have to be done, in order to get accurate mouse movements. This just gets the concept across as to how to decode the basic mouse movements. You can further add code to take into account the more accurate movements of the mouse.

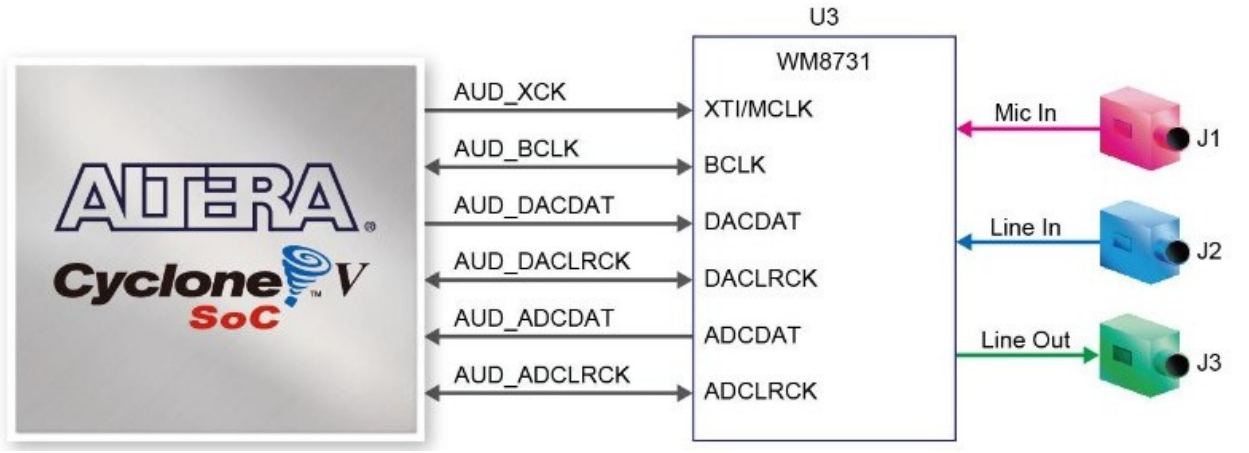
This concludes the mouse tutorial.

Audio interface

The audio interface is a Wolfson **WM8731** CODEC (analog to digital converter [**ADC**] and digital to analog converter [**DAC]**). For further information follow this link:

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

Figure 93 gives an overview of how the audio CODEC chip is connected to the FPGA and the pin assignments.



Signal Name	*FPGA Pin No.*	*Description*
AUD_ADCLRCK	PIN_K8	Audio CODEC ADC LR Clock
AUD_ADCDAT	PIN_K7	Audio CODEC ADC Data
AUD_DACLRCK	PIN_H8	Audio CODEC DAC LR Clock
AUD_DACDAT	PIN_J7	Audio CODEC DAC Data
AUD_XCK	PIN_G7	Audio CODEC Chip Clock
AUD_BCLK	PIN_H7	Audio CODEC Bit-stream Clock
I2C_SCLK	PIN_J12 or PIN_E23	I2C Clock
I2C_SDAT	PIN_K12 or PIN_C24	I2C Data

Figure 93 – Connection of the audio CODEC chip to FPGA

Key features of this chip are:

- Programmable I2C serial interface.
- Multiple baud rate settings, ranging from 8K to 96K.
- Master mode or slave mode settings.
- Internal clock, which can be set depending on the mode.
- Multiple bit length, ranging from 16 to 32 bits.
- 2 inputs [line input and microphone input]
- 1 output to speaker [line out].

This CODEC has multiple settings but for the purpose of this tutorial we will use:

- Master mode.
- 48 KHz baud rate.
- A microphone input.
- A speaker output.

In order to do this, we will use the I2C (Inter-Integrated Circuit) protocol to program the Audio CODEC. If you are not familiar with this protocol return to **Page 5**, where you can learn all the details about it and receive a tutorial on how it functions.

Table 29 on Page 46 of the Wolfson_audio manual gives an overview of all the registers that need to be programmed. The table will look similar to **Figure 94**.

REGISTER	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B8	B7	B6	B5	B4	B3	B2	B1	B0
R0 (00h)	0	0	0	0	0	0	0	LRIN BOTH	LIN MUTE	0	0	LINVOL				
R1 (02h)	0	0	0	0	0	0	1	RLIN BOTH	RIN MUTE	0	0	RINVOL				
R2 (04h)	0	0	0	0	0	1	0	LRHP BOTH	LZCEN	LHPVOL						
R3 (06h)	0	0	0	0	0	1	1	RLHP BOTH	RZCEN	RHPVOL						
R4 (08h)	0	0	0	0	1	0	0	0	SIDEATT		SIDETONE	DAC SEL	BY PASS	INSEL	MUTE MIC	MIC BOOST
R5 (0Ah)	0	0	0	0	1	0	1	0	0	0	0	HPOR	DAC MU	DEEMPH		ADC HPD
R6 (0Ch)	0	0	0	0	1	1	0	0	PWR OFF	CLK OUTPD	OSCPD	OUTPD	DACPD	ADCPD	MICPD	LINEINPD
R7 (0Eh)	0	0	0	0	1	1	1	0	BCLK INV	MS	LR SWAP	LRP	IWL		FORMAT	
R8 (10h)	0	0	0	1	0	0	0	0	CLKO DIV2	CLKI DIV2	SR				BOSR	USB/ NORM
R9 (12h)	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	ACTIVE
R15(1Eh)	0	0	0	1	1	1	1	RESET								
	ADDRESS							DATA								

Figure 94-I2C programmable registers

Here we see there are a total of 11 different registers (R0 – R9, R15). For the purpose of this tutorial the R15 register can be ignored.

When first powered up the registers are reset to a default value but it may not be the desired value.

Table 23 shows what the values should be and gives an explanation for what each update does.

register	Address and Value (HEX)	Explanation
R0	001A	Volume setting for left channel input
R1	021A	Volume setting for right channel input
R2	047B	Volume setting for left channel output
R3	067B	Volume setting for right channel output
R4	08FC	Analog control and input enable (set for Microphone)
R5	0A06	Digital audio control setting (set to 48 KHz)
R6	0C00	Power on setting as opposed to power down setting
R7	0E4A	Format setting (I2C protocol) mode setting (Master mode)
R8	1000	Sampling control stays to same
R9	1201	Set Audio CODEC to active as opposed to inactive

Table 22 - I2C settings for this tutorial

We can now create an I2C serial protocol interface to load the values from **Table 23** into the audio CODEC internal registers.

From the video tutorial, we have already developed Verilog code to generate and load I2C values. We can modify this code to work with the audio CODEC chip. Download the following zip file for an example of what the Verilog code should look like:

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

Select –DE1-SoC>DESL Online Tutorials>audio_part1.zip.

Create a directory and unzip the file. Using the Quartus **new project wizard** create a new project called **audio** and compile the project. For this tutorial there are three Verilog files:

- **audio.v-** Is the main program where all the input and output signals for the audio CODEC chip are located. Also, it has all the connections for the GPIO-0 40 pin header, which will be connected to a logic analyzer. This will be used to view the signals and verify the data.
- **i2c_av_cfg.v –** This module is used to create a state machine, which updates the address and data values that will be programmed into the audio CODEC chip via the sdata I/O pin.
- **i2c_programmer.v** - This module is used to program to I2C registers on the audio chip. The Clock pin is **sclk** and data pin is **sdat**.

Open up **audio.v** and scroll down to line 84. The Verilog code should look similar to **Figure 95**. The purpose of this code is to create the 12.288 MHz input needed for the **AUD_XCK** input on the audio CODEC chip. On **Page 10** of the Wolfson_audio manual you will see that this is the required input frequency, in order to create the proper audio frequency of **48 KHz**. See **R5** in **Table 23**.

```
80    ////////////////////////////////////////////////////
81    ////// I2C clock (50 Mhz)used for DE1-SoC video in chip ///
82    ////////////////////////////////////////////////////
83
84     always @ (posedge i2c_clk or negedge RESET)
85    begin
86        if (!RESET)
87        begin
88           mi2c_clk_div <= 0;
89           mi2c_ctrl_clk <= 0;
90        end
91
92        else
93
94        begin
95
96           if (mi2c_clk_div <  (clk_freq/i2c_freq) )  // keeps dividing until reaches desired frequency
97           mi2c_clk_div <= mi2c_clk_div + 1;
98
99           else
100          begin
101                mi2c_clk_div <= 0;
102                mi2c_ctrl_clk <= ~mi2c_ctrl_clk;
103          end
104       end
105    end
106
107   always @(negedge RESET or posedge CLOCK) begin
108   if (!RESET) SD_COUNTER =  7 'b1111111;
109   else begin
110   if (GO==0)
111      SD_COUNTER=0;
112      else
113      if ((SD_COUNTER < 7 'b1110111) & (TRN_END ==0)) SD_COUNTER = SD_COUNTER + 1;
114   end
115   end
```

Figure 95 - Clock inputs for AUD_XCk

Open up **i2c_cfg.v** and scroll down to line 42. See **Figure 96**. This parameter represents the total size of all the registers that need to be programmed, in this case it is 10. Lines 46-55 indicate what registers on the audio CODEC chip will be programmed as the counter increases from 0 to 9. Each parameter represents a value with a name attached to it.

```
40    // LUT data size value for both audio and video register
41
42    parameter   LUT_size    = 10; // number of values loaded both audio and serial
43
44    //Audio register values  ( 9 in total)
45
46    parameter   set_lin_l   = 0;
47    parameter   set_lin_r   = 1;
48    parameter   set_head_l  = 2;
49    parameter   set_head_r  = 3;
50    parameter   a_path_cntrl = 4;
51    parameter   d_path_cntrl = 5;
52    parameter   power_on    = 6;
53    parameter   set_format  = 7;
54    parameter   sample_cntrl = 8;
55    parameter   set_active  = 9;
56
```

Figure 96-Parameter names of the registers

Scroll down to line 60 and see **Figure 97**. This state machine uses the information from **Figure 96**, in order to update the serial I2C register with the data and address values. From line 68, we see that the LUT_index (a counter) is 0 at reset. After each consecutive load of the register value, the counter will be increased by the amount indicated in line 105 of **Figure 97**. Once LUT_size reaches the value of 10, the load is complete and stops. This is indicated by the statement in line 77 of **Figure 97**.

```
60    always @ (posedge clk or negedge reset)
61
62    begin
63
64       if (!reset)
65
66          begin
67
68          LUT_index   <= 0;
69          mstep       <= 0;
70          mgo         <= 0;
71
72          end
73          else
74
75          begin
76
77          if (LUT_index < LUT_size)
78          begin
79
80          case(mstep)
81             0: begin
82                   if (SCLK)
83
84                   i2c_data <= {8'h34,LUT_data};
85
86                   mgo <= 1;
87                   mstep <= 1;
88                end
89
90             1: begin
91
92                   if (mend)
93                   begin
94
95                      if (mack)
96                      mstep <= 2;
97                      else
98                      mstep <= 0;
99                      mgo <= 0;
100                     end
101                  end
```

```
103             2: begin
104
105                     LUT_index <= LUT_index + 1;
106                     mstep <= 0;
107
108                     end
109
110                endcase
111             end
112          end
113       end
114
115
116    always
117
118    begin
119
120    case ( LUT_index)
121
122    // audio config values
123
124    set_lin_l       : LUT_data <= 16'h001a;
125    set_lin_r       : LUT_data <= 16'h021a;
126    set_head_l      : LUT_data <= 16'h047b;
127    set_head_r      : LUT_data <= 16'h067b;
128    a_path_cntrl    : LUT_data <= 16'h08fc;
129    d_path_cntrl    : LUT_data <= 16'h0a06;
130    power_on        : LUT_data <= 16'h0c00;
131    set_format      : LUT_data <= 16'h0e4a;
132    sample_cntrl    : LUT_data <= 16'h1000;
133    set_active      : LUT_data <= 16'h1201;
```

Figure 97 - Serial I2C address data update

Now **open i2c_programmer.v** and scroll down to line 124. See **Figure 98**. Note this is just part of the whole program and for the purpose of this explanation it is not necessary to show the entire Verilog code. This state machine takes the updated address and data values loaded in **i2c_av_cfg.v** and shifts them out onto the **SDAT** and **SCLK** lines using the I2C format. There is some hand shaking that takes place between the **i2c_programmer.v** module and the **i2c_av_cfg.v** module, in order to make sure all the data and address values are loaded into the audio CODEC chip registers. Review the code to see how both modules interact with each other.

```
always @ (negedge RESET or posedge CLOCK) begin

if (!RESET) begin ACK1 = 0; ACK2 = 0; ACK3 = 0; TRN_END = 1;  ACK_enable = 1; SCLK = 1; SDO = 1; end
else
case (SD_COUNTER)



  7'd0  : begin ACK1 = 0; ACK2 = 0; ACK3 = 0; TRN_END = 0; SDO = 1; SCLK = 1; ACK_enable =1; end
     7'd1  : begin SD= (data_23); SDO = 0; end
     // begin load
     // slave address
     7'd2  : begin SDO = SD[23]; SCLK = 0; end
     7'd3  : begin SDO = SD[23]; SCLK = 1; end
     7'd4  : begin SDO = SD[23]; SCLK = 1; end
     7'd5  : begin SDO = SD[23]; SCLK = 0; end

     7'd6  : begin SDO = SD[22]; SCLK = 0; end
     7'd7  : begin SDO = SD[22]; SCLK = 1; end
     7'd8  : begin SDO = SD[22]; SCLK = 1; end
     7'd9  : begin SDO = SD[22]; SCLK = 0; end

     7'd10 : begin SDO = SD[21]; SCLK = 0; end
     7'd11 : begin SDO = SD[21]; SCLK = 1; end
     7'd12 : begin SDO = SD[21]; SCLK = 1; end
     7'd13 : begin SDO = SD[21]; SCLK = 0; end

     7'd14 : begin SDO = SD[20]; SCLK = 0; end
     7'd15 : begin SDO = SD[20]; SCLK = 1; end
     7'd16 : begin SDO = SD[20]; SCLK = 1; end
     7'd17 : begin SDO = SD[20]; SCLK = 0; end

     7'd18 : begin SDO = SD[19]; SCLK = 0; end
     7'd19 : begin SDO = SD[19]; SCLK = 1; end
     7'd20 : begin SDO = SD[19]; SCLK = 1; end
     7'd21 : begin SDO = SD[19]; SCLK = 0; end

     7'd22 : begin SDO = SD[18]; SCLK = 0; end
     7'd23 : begin SDO = SD[18]; SCLK = 1; end
     7'd24 : begin SDO = SD[18]; SCLK = 1; end
     7'd25 : begin SDO = SD[18]; SCLK = 0; end

```

Figure 98 - Serial data shift register and SCLK

Using the MSO-X-3024A scope we can analyze these signals.

- Press **Digital.**
- Enable **D0-D7.** Disable the other digital lines.
- Set **Scale** to **medium**
- Press **Label** and rename D0-D7 according to column 4 in **Table 24**.

40 pin HeaderJP0	Pin assignment	Digital lead default name	Rename label
GPIO[0]	AC18	D0	CTRL_CLK
GPIO[1]	Y17	D1	BCLK
GPIO[2]	AD17	D2	DACDAT
GPIO[3]	Y18	D3	DACLRCK
GPIO[4]	AK16	D4	ADCDAT
GPIO[5]	AK18	D5	ADCLRCK
GPIO[6]	AK19	D6	SCLK
GPIO[7]	AJ19	D7	SDAT

Table 23 - Pin assignments for audio tutorial Part 1

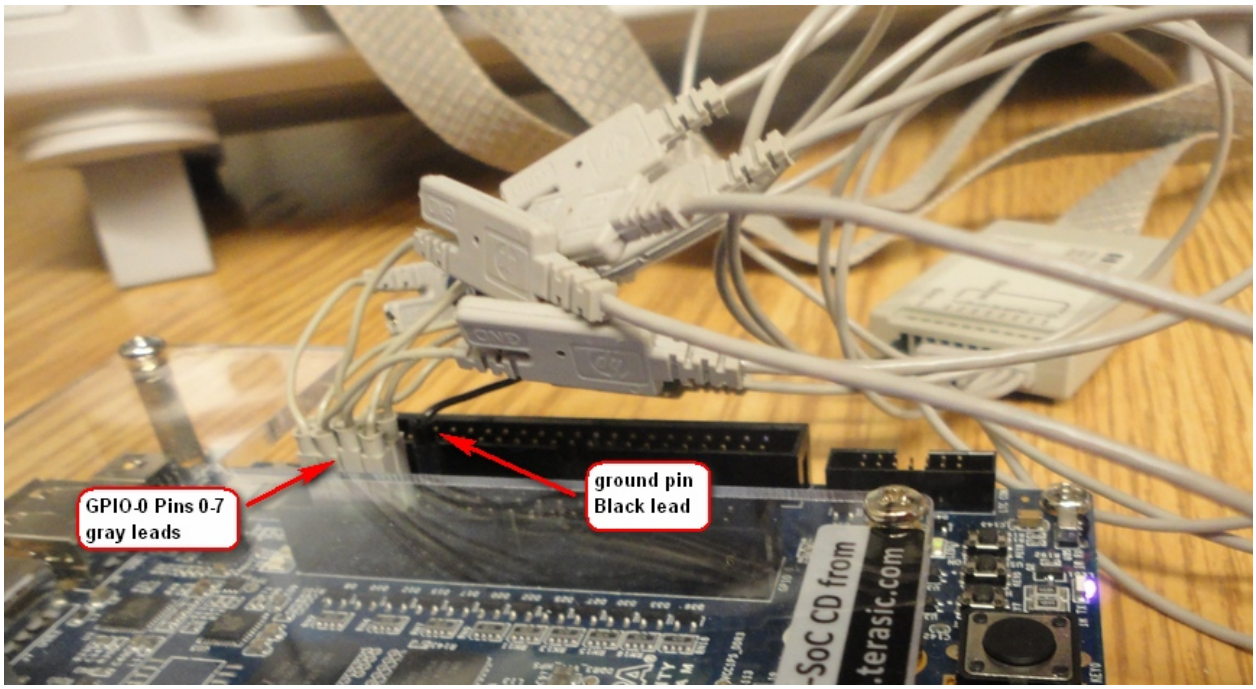


Figure 99 - Connecting the MSO-X-3024A leads from the scope to the DE1-SoC 40 pin header (GPIO-0)

- Connect the leads to the GPIO-0 header on the DE1-SoC board. See **Figure 99**.
- Make sure all switches on the DE1-SoC are down.
- Download the **audio.sof** file to the DE1-SoC.
- Press **Serial**.
- Press **Signals.**
- Select **D6** for SCL.
- Select **D7** for SDA.
- Press **Trigger**.
- Select **Serial1(I2C).**
- Select **Start** as trigger mode.
- Set **Horizontal 10**m/s.
- Set **Delay 30**m/s.
- Press **Single**.
- **Note** that none of the digital channels are active. See **Figure 100**.
- Flip **Switch 0** up.

The trigger event should look similar to **Figure 101**.

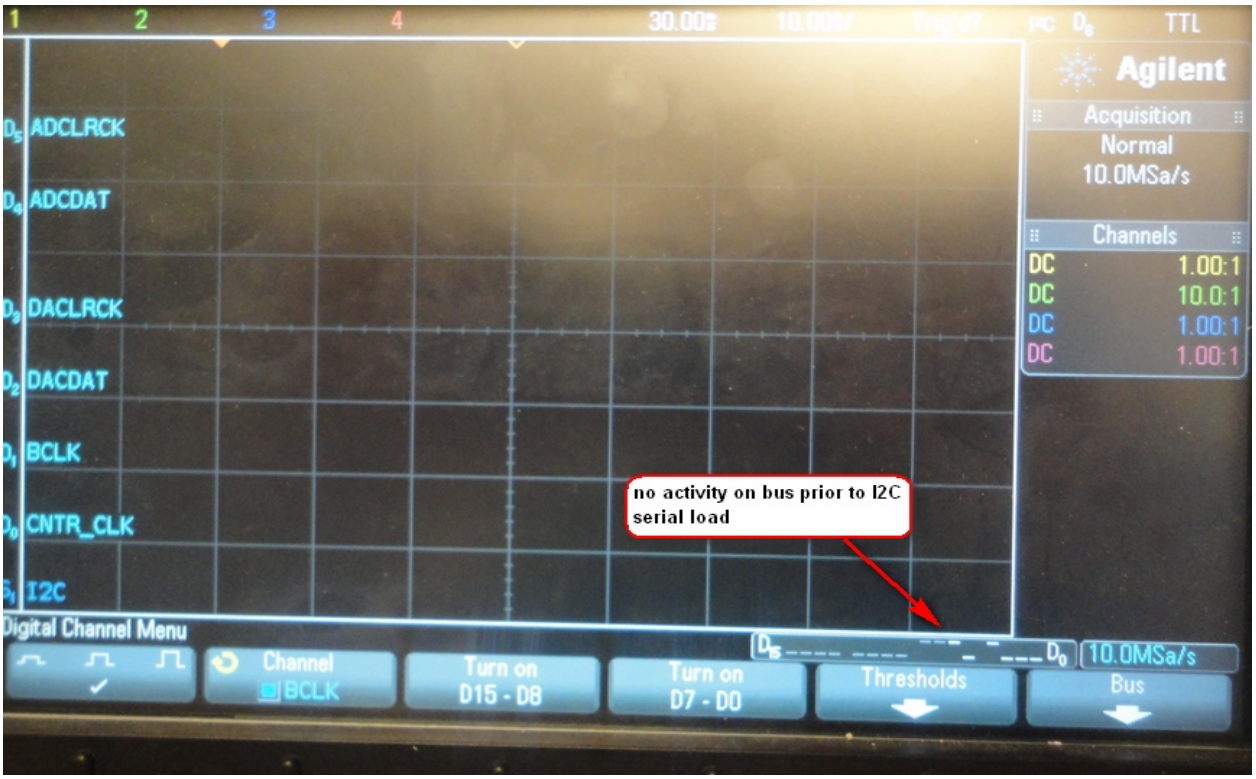


Figure 100 - No signal activity prior to the trigger event

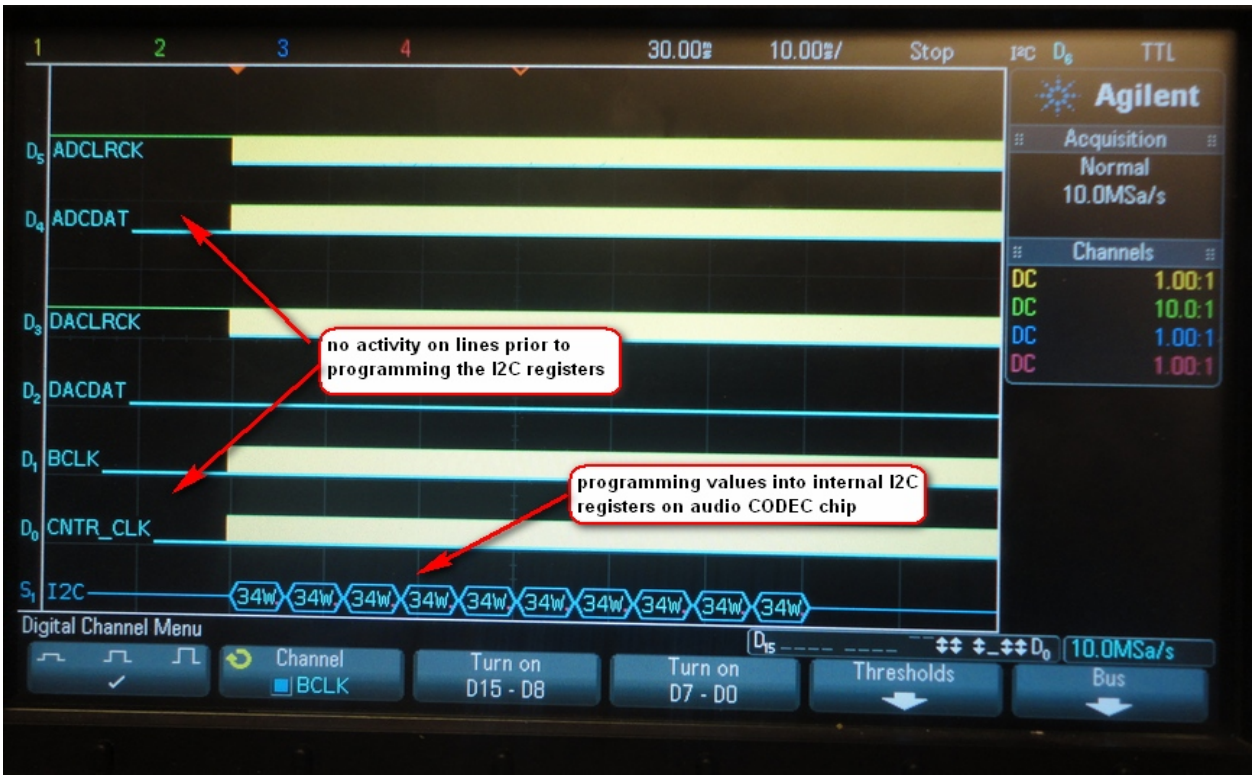


Figure 101 – Programming the internal I2C registers

- Press **Serial**.
- Set **Addr Size** to 8 bit.
- Press **Lister**.

The result will look similar to **Figure 102**. Here we get an overview of all the address and data values being loaded into the audio CODEC, via the I2C data bus. If you compare the values listed in **Figure 102** and **Table 23** they should be the same.

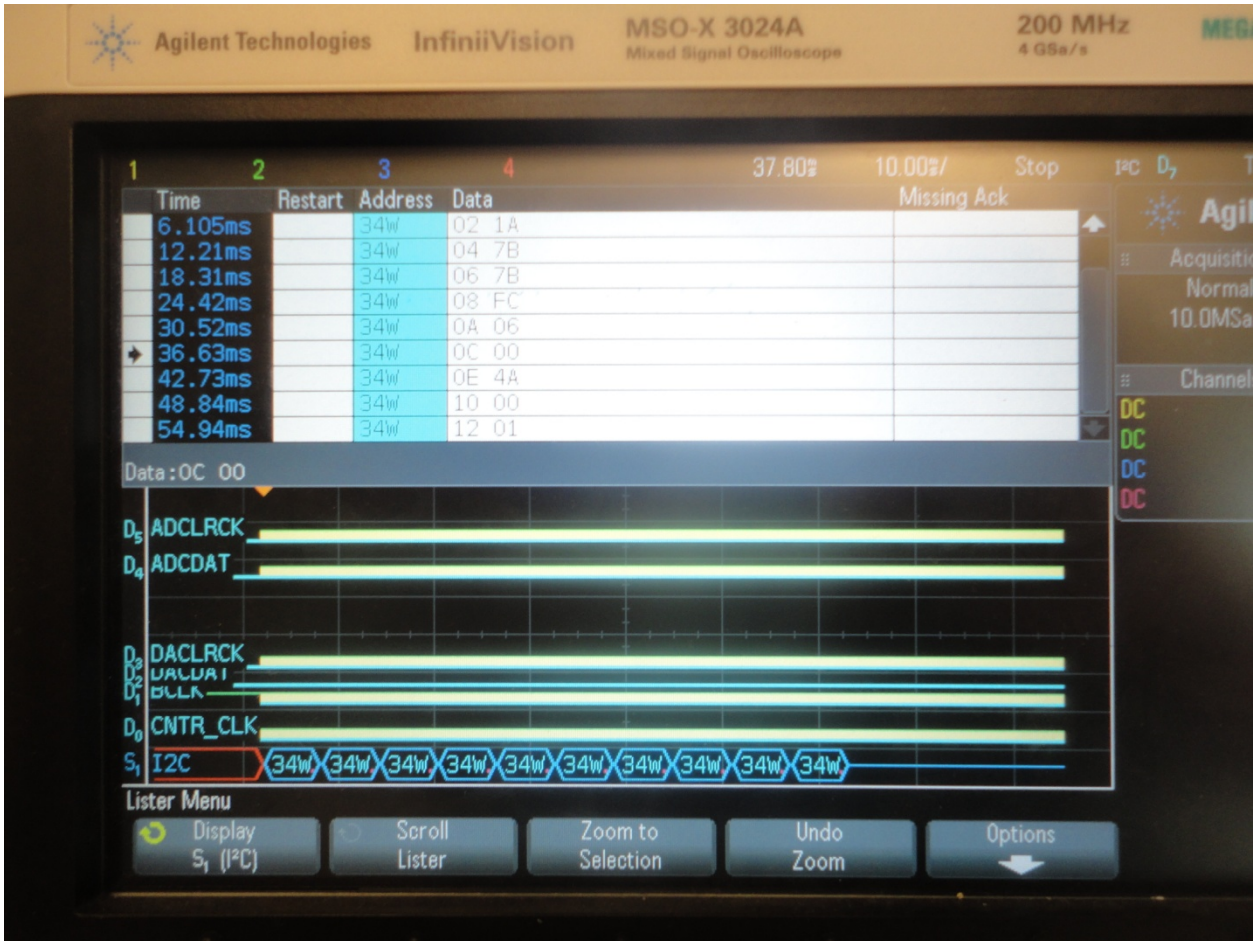


Figure 102-List of all the program registers

Once the Audio CODEC chip has been programmed, the **BCLK**, **DACLRCK**, and **ADCLRCK** signals become active. **DACDAT** is inactive, since it has not been programmed yet. It will be programmed in Part 2 of this tutorial. **ADCDAT** is active but it is just random noise generated by the input.

- Press **Trigger.**
- Select **Edge** mode.
- Set **Source** to BCLK.
- Set **Slope** to falling Edge.
- Set **Delay** to **0.0**s.
- Set **Horizontal** to **500**u/s.

- Press **Single.**
- Press **Zoom.**
- Set **Horizontal** to **200**n/s.

The result should look similar to **Figure 103**.

Figure 103 - Zoomed in on BCLK

- Press **Cursor**
- Press **Cursors.**
- Select **X1** and move to the rising edge of **BCLK**
- Select **X2** and move to the next rising edge of **BCLK.**
- The resulting delta frequency should be 1.25 MHz, see **Figure 104**.

Figure 104 – Measuring the BCLK delta frequency

- Press **Cursors** again. This should remove the cursors from the screen.
- Set **Horizontal** to **20**u/s.

The result should look similar to **Figure 105**.

- Notice that we now have a better view of the **ADCLRCK** and **DACLRCK** signals. To further understand what these signals mean, go to **Page 33** of the Wolfson_audio manual and look at **Figure 26**. It will look similar to **Figure 106**.
- When the **ADCLRCK** signal is active high and on the positive edge of **BCLK**, valid audio data is received by the left channel of the audio CODEC.
- When the **ADCLRCK** signal is active low and on the positive edge of **BCLK,** valid audio data is received by the right channel of the audio CODEC.
- The same applies for the **DACLRCK** signal, with the only difference being that digital data is converted to a serial output.
- When observing the trigger event in **Figure 105** we see that the signals are behaving in this manner. This indicates that the serial I2C internal registers have been properly programmed.
- Also note that **DACDAT** does not have any signal value. This is because we need to create a data register in order to convert the 32 bit audio value to a serial output.

- As noted earlier, the signal values on the **ADCDAT** line are random noise. To get proper signal values a serial to parallel data register needs to be created.
- We will address both of these registers in Part 2 of this tutorial.

Figure 105 - Zoomed out view of the left right channel enables for the I2C format

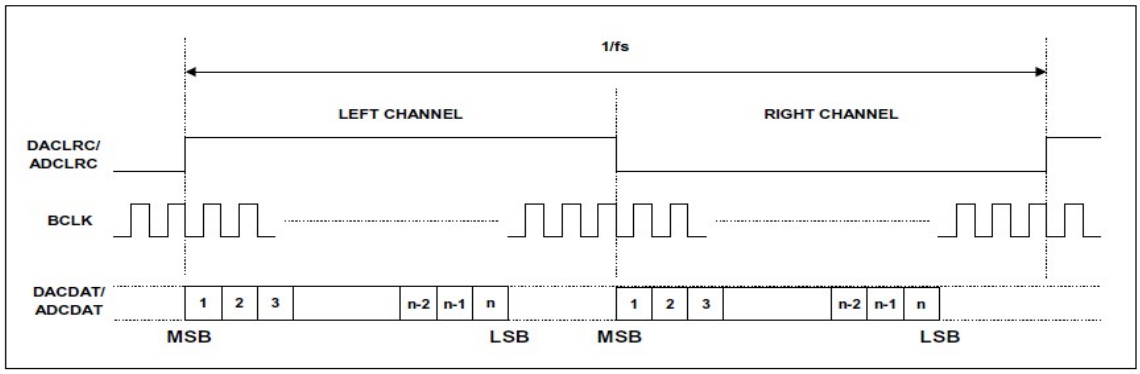


Figure 26 Left Justified Mode

I^2S mode is where the MSB is available on the 2nd rising edge of BCLK following a DACLRC or ADCLRC transition.

Figure 106 - A copy of page 33 of the Wolfson_audio manual

This concludes this tutorial.

Part 2 - Creating the data register for audio in and out

In Part 1 our objective was to create an I2C data signal and clock signal, in order to load address and data values and transmit them to the internal register of the audio CODEC.

In Part 2 we will perform the following:

- Create a serial to parallel data register, which will capture analog data from the audio CODEC input.
- Create a parallel to serial register, which will send data out the analog output to a speaker.
- From **Figure 106**, recall that when **ADCLRCK** and **DACLRCK** are active high, the left channel analog data is captured or sent on the rising edge of **BCLK**.
- When **ADCLRCK** and **DACLRCK** are active low, the right channel analog data is captured or sent on the rising edge of **BCLK**.
- There are a total of 33 bits sent or received when either **ADCLRCK** or **DACLRCK** is active high or low. We will need to know this information when building the counter state machine.
- The MSO-X-3024A scope setup from Part 1 will be used to aid in verifying that the number of **BCLK** cycles between each **ADCLRCK** and **DACLRCK** level change is indeed 33.
- **Page 50** of the Wolfson_audio manual indicates that the maximum size of any audio packet is 32 bits. We need to keep this in mind for the register.
- Another important point to keep in mind is that **MSB** is sent first, see **Figure 106**.

The following Verilog code is an example of what the code should look like. Download the following zip file:

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

Select –DE1-SoC>DESL Online Tutorials>audio_part2.zip.

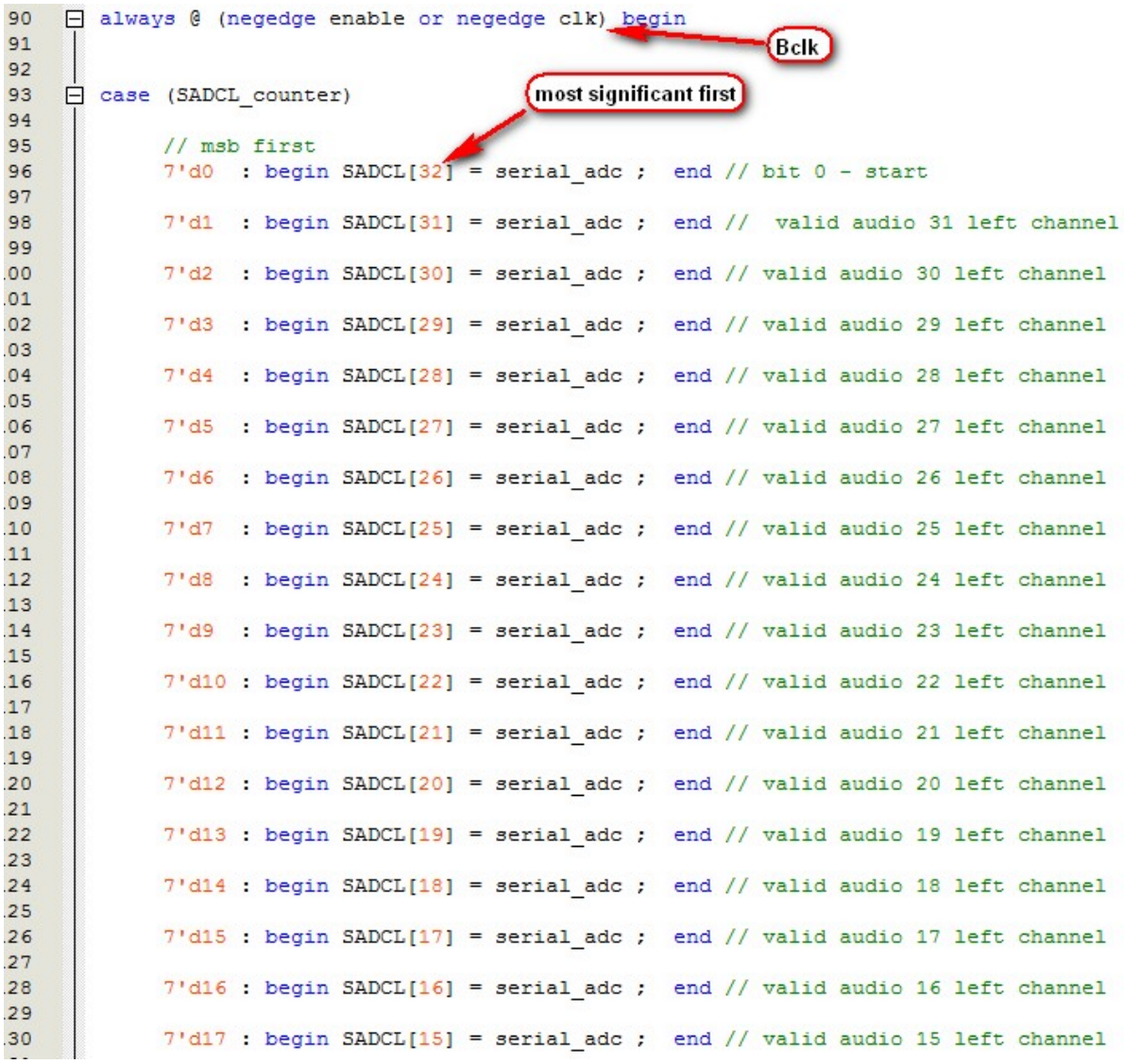
Create a directory and unzip the file. Using the Quartus **new project wizard** create a new project called **audio** and compile the project. For this tutorial we have five Verilog files: **audio.v**, **i2c_programmer.v , i2c_av_cfg.v**, **serial_adc.v**, and **serial_dac.v**.

Open **serial_adc.v** and scroll down to line 45. Lines 45-64 are the counter enable for the right channel. When **adc_lr** is active high, the right channel counter is enabled. Lines 68-86 are the counter enable for left channel. See **Figure 107**.

```
always @(negedge enable or posedge clk) begin

  if  (!enable)

  begin

  SADCR_counter = 7'b0; // reset right channel counter
  end

  else begin

  if (!adc_lr)
  SADCR_counter = 7'b0;


  else
  SADCR_counter = SADCR_counter + 1; // right channel captures audio
  end

  end

////////////////////////////////////////////////////////////////

always @(negedge enable or posedge clk) begin

  if  (!enable)

  begin
  SADCL_counter = 7'b0; // reset left channel counter

  end

  else begin

  if (adc_lr)
  SADCL_counter = 7'b0;


  else
  SADCL_counter = SADCL_counter + 1; // left channel captures audio

  end
```

Figure 107 - Left and right channel enables

The right counter (SADCR_counter) from **Figure 107** is used to save right channel data. The left counter (SADCL_counter) from **Figure 107** is used to save left channel data.




```
90  always @ (negedge enable or negedge clk) begin
91
92
93  case (SADCL_counter)
94
95       // msb first
96       7'd0  : begin SADCL[32] = serial_adc ;  end // bit 0 - start
97
98       7'd1  : begin SADCL[31] = serial_adc ;  end //  valid audio 31 left channel
99
00       7'd2  : begin SADCL[30] = serial_adc ;  end // valid audio 30 left channel
01
02       7'd3  : begin SADCL[29] = serial_adc ;  end // valid audio 29 left channel
03
04       7'd4  : begin SADCL[28] = serial_adc ;  end // valid audio 28 left channel
05
06       7'd5  : begin SADCL[27] = serial_adc ;  end // valid audio 27 left channel
07
08       7'd6  : begin SADCL[26] = serial_adc ;  end // valid audio 26 left channel
09
10       7'd7  : begin SADCL[25] = serial_adc ;  end // valid audio 25 left channel
11
12       7'd8  : begin SADCL[24] = serial_adc ;  end // valid audio 24 left channel
13
14       7'd9  : begin SADCL[23] = serial_adc ;  end // valid audio 23 left channel
15
16       7'd10 : begin SADCL[22] = serial_adc ;  end // valid audio 22 left channel
17
18       7'd11 : begin SADCL[21] = serial_adc ;  end // valid audio 21 left channel
19
20       7'd12 : begin SADCL[20] = serial_adc ;  end // valid audio 20 left channel
21
22       7'd13 : begin SADCL[19] = serial_adc ;  end // valid audio 19 left channel
23
24       7'd14 : begin SADCL[18] = serial_adc ;  end // valid audio 18 left channel
25
26       7'd15 : begin SADCL[17] = serial_adc ;  end // valid audio 17 left channel
27
28       7'd16 : begin SADCL[16] = serial_adc ;  end // valid audio 16 left channel
29
30       7'd17 : begin SADCL[15] = serial_adc ;  end // valid audio 15 left channel
```

Figure 108 - Left channel serial to parallel load

At the negative edge of **clk** (**Line 90** in **Figure 108)** a new value is loaded from the analog input **serial_adc** into **SADCL** and the counter is increased.

- **Note** since the audio data from the audio chip is updated on the positive edge of Bclk, in order to make sure we get updated audio data values, we have chosen to use the negative edge of the clock in line 90.
- **Note** that the load is MSB to LSB as indicated above.

Scroll to line 167 and see **Figure 109**. The behavior of this state machine is the same as **Figure 108** but it is for the right channel. Both these state machines are used to get analog data from a sound source, which in this case will be a microphone, and store it in a 33 bit register called **SADCL** and **SADCR**.

```
167        always @ (negedge enable or negedge clk) begin
168
169
170  case (SADCR_counter)
171
172        // msb first
173        7'd0  : begin SADCR[32] = serial_adc ;  end // bit 0 - start
174
175        7'd1  : begin SADCR[31] = serial_adc ;  end //  valid audio 31 right channel
176
177        7'd2  : begin SADCR[30] = serial_adc ;  end // valid audio 30 right channel
178
179        7'd3  : begin SADCR[29] = serial_adc ;  end // valid audio 29 right channel
180
181        7'd4  : begin SADCR[28] = serial_adc ;  end // valid audio 28 right channel
182
183        7'd5  : begin SADCR[27] = serial_adc ;  end // valid audio 27 right channel
184
185        7'd6  : begin SADCR[26] = serial_adc ;  end // valid audio 26 right channel
186
187        7'd7  : begin SADCR[25] = serial_adc ;  end // valid audio 25 right channel
188
189        7'd8  : begin SADCR[24] = serial_adc ;  end // valid audio 24 right channel
190
191        7'd9  : begin SADCR[23] = serial_adc ;  end // valid audio 23 right channel
192
193        7'd10 : begin SADCR[22] = serial_adc ;  end // valid audio 22 right channel
194
195        7'd11 : begin SADCR[21] = serial_adc ;  end // valid audio 21 right channel
196
197        7'd12 : begin SADCR[20] = serial_adc ;  end // valid audio 20 right channel
198
199        7'd13 : begin SADCR[19] = serial_adc ;  end // valid audio 19 right channel
200
201        7'd14 : begin SADCR[18] = serial_adc ;  end // valid audio 18 right channel
202
203        7'd15 : begin SADCR[17] = serial_adc ;  end // valid audio 17 right channel
204
205        7'd16 : begin SADCR[16] = serial_adc ;  end // valid audio 16 right channel
206
```

Right channel serial save MSB first

Figure 109 - Right channel serial to parallel load

Open up **audio.v** and scroll down to lines 52 and 53 (**Figure 110)**. Now scroll down to lines 63 and 64 (**Figure 110)**. You will notice that **serial_If** is connected to **SADCL** and **SDACL**. This means that **serial_If** acts as a buffer between the analog input and output. The digital conversion and audio sync is performed by the FPGA state machine. The same applies for the right channel, **serial_rt.**

Open up **serial_dac.v** and scroll down to line 87, see **Figure 111**. Again we have a counter and at each negative edge of **clk** (BCLK) the value in **SDACL** is loaded into **serial_1**. See **Figure 111**. Similarly, the same takes place for the right channel with the only difference being that it is loaded into **serial_2**. See **Figure 112**. Now scroll down to line 39 and see **Figure 113**. Here the values from either **serial_1** or **serial _2** are sent out to **serial_dat,** depending on the state value of **dac_lr**.

If **dac_lr** is active high, **serial_1** sends data out the DAC and when **dac_lr** is active low, **serial_2** sends data out the DAC. We now have a full analog to digital and a digital to analog conversion.

```
49   serial_adc u2 (
50
51       .serial_adc(adcdat),    // 32 bit serial in data
52       .SADCL(serial_lf),
53       .SADCR(serial_rt),
54       .adc_lr(adclrck),
55       .clk(clk),              // 50 KHz clock
56       .enable(swt)            // master reset
57
58   );
59
60   serial_dac u3(
61
62       .serial_dac(dacdat),    // 32 bit serial in data
63       .SDACL(serial_lf),
64       .SDACR(serial_rt),
65       .dac_lr(daclrck),
66       .clk(clk),              // 50 KHz clock
67       .enable(swt)            // master reset
68
69   );
```

Figure 110 - Modules for serial conversion

```
87   always @ (negedge enable or negedge clk) begin
88
89
90   case (SDACL_counter)
91
92        // msb first
93        7'd0  : begin serial_1 = SDACL[32]  ;  end // bit 0 - start
94
95        7'd1  : begin serial_1 = SDACL[31]  ;  end //  valid audio 31 left channel
96
97        7'd2  : begin serial_1 = SDACL[30] ;  end // valid audio 30 left channel
98
99        7'd3  : begin serial_1 = SDACL[29]  ;  end // valid audio 29 left channel
100
101       7'd4  : begin serial_1 = SDACL[28]  ;  end // valid audio 28 left channel
102
103       7'd5  : begin serial_1 = SDACL[27]  ;  end // valid audio 27 left channel
104
105       7'd6  : begin serial_1 = SDACL[26]  ;  end // valid audio 26 left channel
106
107       7'd7  : begin serial_1 = SDACL[25]  ;  end // valid audio 25 left channel
108
109       7'd8  : begin serial_1 = SDACL[24]  ;  end // valid audio 24 left channel
110
111       7'd9  : begin serial_1 = SDACL[23]  ;  end // valid audio 23 left channel
112
113       7'd10 : begin serial_1 = SDACL[22]  ;  end // valid audio 22 left channel
114
115       7'd11 : begin serial_1 = SDACL[21]  ;  end // valid audio 21 left channel
116
117       7'd12 : begin serial_1 = SDACL[20]  ;  end // valid audio 20 left channel
118
119       7'd13 : begin serial_1 = SDACL[19]  ;  end // valid audio 19 left channel
120
121       7'd14 : begin serial_1 = SDACL[18]  ;  end // valid audio 18 left channel
122
123       7'd15 : begin serial_1 = SDACL[17]  ;  end // valid audio 17 left channel
124
```

Figure 111 - Left channel digital to analog conversion

```
164     always @ (negedge enable or negedge clk) begin
165
166
167     case (SDACR_counter)
168
169     // msb first
170     7'd0  : begin serial_2 = SDACR[32] ;  end // bit 0 - start
171
172     7'd1  : begin serial_2 = SDACR[31] ;  end // valid audio 31 right channel
173
174     7'd2  : begin serial_2 = SDACR[30] ;  end // valid audio 30 right channel
175
176     7'd3  : begin serial_2 = SDACR[29] ;  end // valid audio 29 right channel
177
178     7'd4  : begin serial_2 = SDACR[28] ;  end // valid audio 28 right channel
179
180     7'd5  : begin serial_2 = SDACR[27] ;  end // valid audio 27 right channel
181
182     7'd6  : begin serial_2 = SDACR[26] ;  end // valid audio 26 right channel
183
184     7'd7  : begin serial_2 = SDACR[25] ;  end // valid audio 25 right channel
185
186     7'd8  : begin serial_2 = SDACR[24] ;  end // valid audio 24 right channel
187
188     7'd9  : begin serial_2 = SDACR[23] ;  end // valid audio 23 right channel
189
190     7'd10 : begin serial_2 = SDACR[22] ;  end // valid audio 22 right channel
191
192     7'd11 : begin serial_2 = SDACR[21] ;  end // valid audio 21 right channel
193
194     7'd12 : begin serial_2 = SDACR[20] ;  end // valid audio 20 right channel
195
196     7'd13 : begin serial_2 = SDACR[19] ;  end // valid audio 19 right channel
197
198     7'd14 : begin serial_2 = SDACR[18] ;  end // valid audio 18 right channel
199
200     7'd15 : begin serial_2 = SDACR[17] ;  end // valid audio 17 right channel
201
202     7'd16 : begin serial_2 = SDACR[16] ;  end // valid audio 16 right channel
```

Figure 112 - Right channel digital to analog conversion

```
35    ////////////////////////////////////////
36    // state machine for serial counter   //
37    ////////////////////////////////////////
38
39    assign serial_dac = (dac_lr)? serial_1 : serial_2 ;
40    reg serial_1;
41    reg serial_2;
42
```

Figure 113 - Left and right enables for DAC

Let's test this out:

- Connect a microphone to the audio input connector, see **Figure 114**.
- Connect a speaker to the audio output connector, see **Figure 114**.
- Make sure all switches on the DE1-SoC board are down.
- Download the **audio.sof** file to the DE1-SoC board.

- Flip **switch 0** to the up position.
- Speak into the microphone.

You should hear yourself on the speaker. If not, increase the volume until you do.

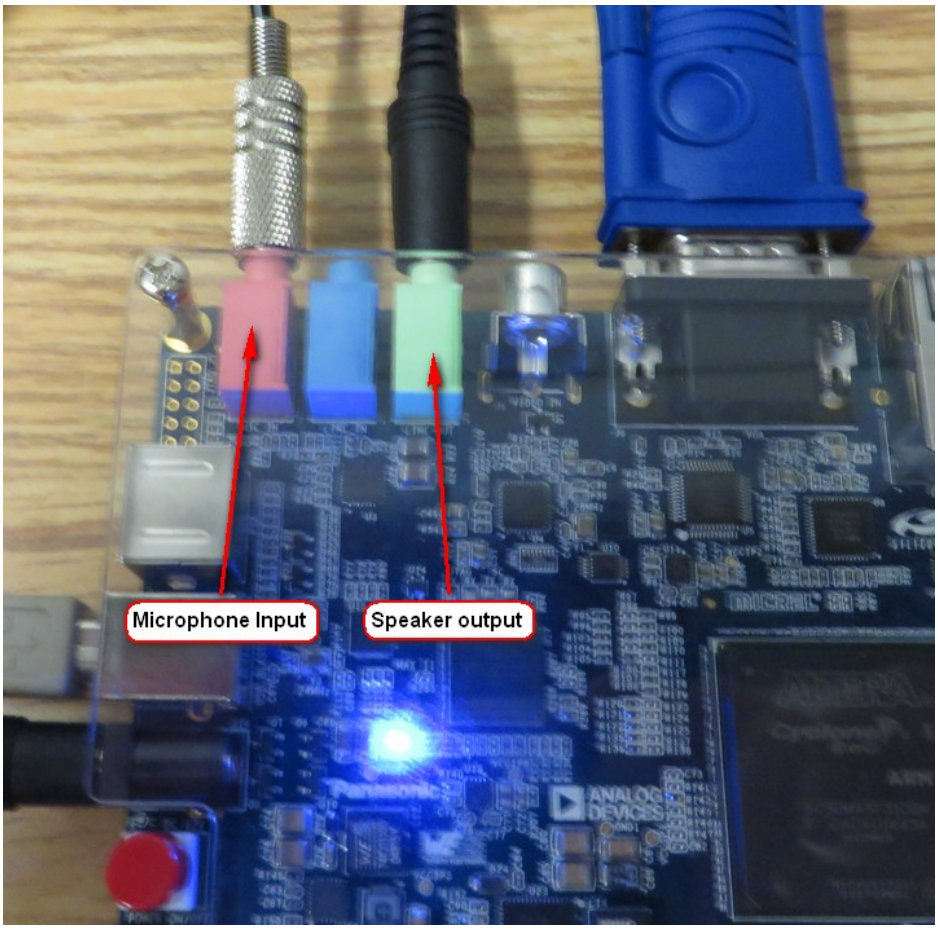


Figure 114 - Speaker and microphone connections

This concludes this tutorial and the manual. I hope it was helpful in understanding the different peripherals on the DE1-SoC board. If you have any questions or concerns email:

aulich@ece.utoronto.ca