Framebuffer Driven LCD Display Design Example

Author: Devon Andrade

Date: 1/18/2016

1.0 – 8/25/2015	Devon Andrade	Initial Release
1.1 – 1/18/2016	Larry Landis	Added note on qip before SDC, adjust SW build flow

Table of Contents

Theory of Operation

This design example demonstrates how to draw complex graphics to the NEEK's LCD using a framebuffer driven approach. A framebuffer is a block of memory used to store pixel color values (red, green, and blue). In this design, the Nios II Processor writes to DDR3 memory where the framebuffer is located and programs in the colors of all of the pixels. The Framebuffer Reader module then reads outs that data from memory and performs a memory-mapped to streaming interface conversion. Lastly, the streaming pixel data is sent to the LCD Driver to get displayed to the screen. Below, you will find descriptions for how the LCD works as well as how all of the custom modules in this design were constructed (Framebuffer Reader and the LCD Driver).

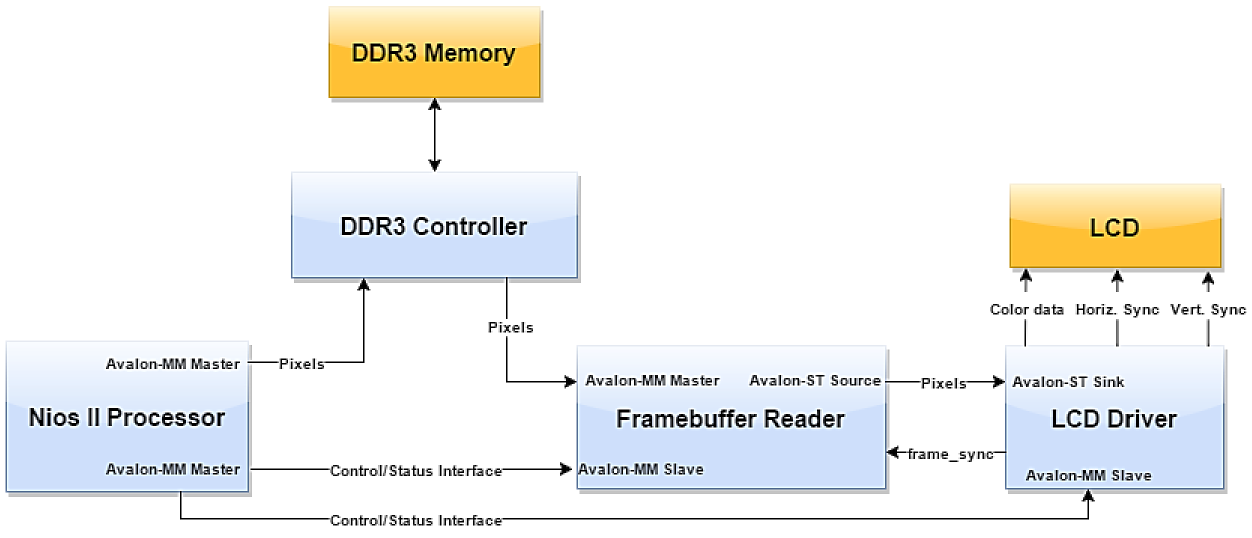


Figure 1: Block diagram of the entire design

LCD Driver

The LCD Driver is the module that is responsible for providing color and synchronization signals to the LCD at the appropriate times. The LCD used on the NEEK board accepts regular VGA signals that are most commonly used with desktop PC monitors. Beyond the hardware needed to generate the synchronization signals, this module also has an Avalon-MM Slave interface for reading back status information from a processor. This lets a processor know when the LCD Driver is inside of a vertical or horizontal blanking period (discussed in the next section).

Two Verilog files are used to create this component:

<Source Root>/IP/lcd_driver/hdl/lcd_driver.sv: Top-level file that contains the Avalon interfaces and instantiates the Sync Gen module.

<Source Root>/IP/lcd_driver/hdl/sync_gen.sv: Generates the synchronization signals required by VGA.

Video Graphics Array (VGA) Interface

Before you can understand how the sync generator is designed, you need to understand the ubiquitous Video Graphics Array (VGA) interface. The LCD driver on this board (the chip that takes in video signals and translates them into what the LCD needs) accepts standard VGA signals (the same signals that would be used to drive a monitor) for sending pixel data, as well as an extra signal for enabling and disabling the backlight (MTL2_BL_ON_n). The VGA interface consists of three 8-bit signals for each of the color values (red, green, and blue) along with two synchronization signals (which tell the monitor when to update the screen). The synchronization signals require exact timing to make sure the display is drawing the correct pixels at the correct time (in essence, determining the resolution of the display).

Figure 2 below shows the timing parameters for the common 640x480 resolution. Although this diagram doesn't show it, there's a certain "pixel clock" which is the frequency at which video data and the synchronization pulses are clocked out at. For the 640x480 resolution, the pixel clock would have a frequency of 25.175MHz (this is a part of the VESA standard, although LCD datasheets should give you this number as well). This would be the frequency you would run your VGA controller at within your FPGA. Most monitors don't require you to send out the actual pixel clock with the data—they'll assume the pixel clock you're using based on the timing of your synchronization pulses and the resolution they're set to. With that said, low-level LCD driver or video codec chips seem to be the exception and you will need to output a pixel clock alongside your color data on the NEEK board (MTL2_DCLK).

Let's start off by explaining the timing required for the horizontal synchronization pulse. Each line of pixels (640 pixels in this case) is ended with a horizontal synchronization pulse (active low in this case). There is a certain period of time before and after the synchronization pulse that is called the "blanking" period where the screen doesn't display any video data. This was used in older CRT monitors to give the electron beams time to move back to the beginning of the next line and has been carried over into modern electronics for backwards compatibility. The blanking time in front of the sync pulse is called the "front porch" while the blanking time after the pulse is called the "back porch." During the blanking period it is recommended to drive all of the color signals to zero (aka, black) to make sure nothing is displayed.

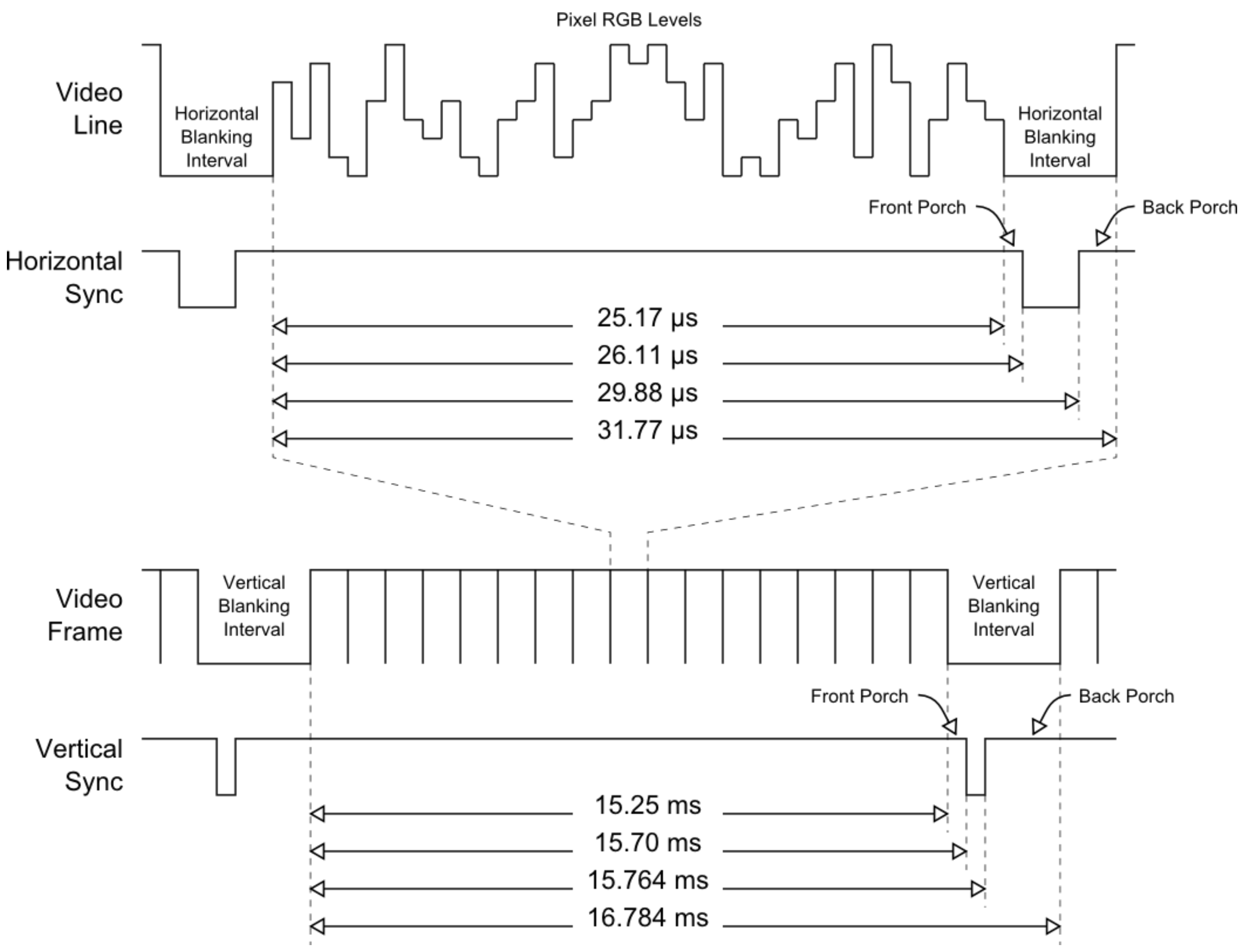


Figure 2: Timing parameters for the 640x480 resolution

As for actual video data, each pixel clock cycle after the blanking period a new pixel will be clocked "into" the screen with the screen displaying pixel 0 in the top left corner and moving to the right to fill the line. Once the line is complete, the blanking period begins and the video data is driven low and the cycle repeats itself. In essence, for each line of data, the following is occurring (all of the timing parameters below are values you can find in the VESA standard online):

1. Pixel data is shifted out over 640 clock cycles (each clock cycle determining the color for a different pixel in the line, starting on the left and moving right).
2. The blanking period begins (the front porch) and video data is driven low for 16 clock cycles
3. The horizontal synchronization pulse is driven low for 96 cycles. This is still part of the blanking period, so we continue to drive the video data low (aka, black).
4. The horizontal synchronization pulse goes back to being asserted, and we wait 48 clock cycles for the back porch before we repeat the process and start producing pixels for the next line (moving down the screen as it goes).

The process above repeats for 480 times to display an entire frame of video data. Just as there is a blanking period after displaying an entire line of video data, there is also a blanking period after displaying an entire frame. This was used by CRT monitors to move the electron beams back to the top left corner of the screen. And just as with the horizontal timing, the blanking period consists of a front porch, synchronization pulse, and back porch where each of these parameters is measured in “lines” (how many horizontal lines it takes to fulfill that part of the blanking period). And since this is a blanking period, the video data should be driven low. These blanking signals are checked within the top-level module to determine whether to drive the color signals low, or to display color data streaming in from the Framebuffer Reader. The process to display an entire frame of information is shown below:

1. Follow the process for displaying a single line of data above and repeat it 480 times.
2. The vertical blanking period begins (front porch) and the video data is driven low for 10 lines (each one of these lines is exactly the same as the lines produced before, except the video data is always low).
3. The vertical synchronization pulse is driven low for two lines.
4. The vertical synchronization pulse is re-asserted and we wait 33 lines for the back porch before we start repeating the process to display the next frame of data.

In essence, each line of video data actually takes up 800 clock cycles (640 for the visible area + 160 for the blanking period) and each entire frame of video data takes 525 lines (480 for the visible area + 45 for the blanking period). When updating the video data to be displayed onto the screen (for instance, if you’re storing the video data in memory), it is highly recommended to update only during the vertical blanking period (when the screen isn’t displaying anything). This ensures that you don’t try to update the screen while your VGA controller is halfway through drawing the screen. This can cause an effect called screen tearing which results in the screen showing half of the new video data, and half of the old video data. Many computer games have an option in their graphics settings called “Vsync” which tells the game to wait until the vertical blanking period before updating the screen (this reduces frame rate, but eliminates screen tearing).

LCD Timing Parameters

The NEEK board’s LCD follows the same VGA interface described above, albeit with different timing parameters than those described in the previous section. There are standard resolutions that most monitors conform to, and you can visit tinyvga.com to discover the exact timings (pixel clock frequency, and horizontal/vertical timings) for most standard resolutions. Unfortunately, the NEEK’s LCD doesn’t use a standard resolution; it instead opts for a non-standard 800x480 resolution. Below are the timing parameters required to display data on the NEEK’s LCD:

Pixel Clock Frequency: 33.33MHz (33MHz will suffice)

Table 1: LCD timing parameters for the NEEK board

Horizontal Scanline Part	Pixels		Vertical Frame part	Lines
Visible area	800		Visible area	480
Front Porch	210		Front Porch	22
Sync Pulse	30		Sync Pulse	13
Back Porch	16		Back Porch	10
Whole line	1056		Whole frame	525

Most LCD screens will take timing parameters that are "close enough" and still manage to display everything correctly. With that said, it is highly recommended that you use the exact values above to get the crispest display.

Sync Generator Module

This module is required to output the horizontal and vertical synchronizations pulses at the correct time. It also outputs two active high blanking signals that get asserted whenever the screen is displaying either the horizontal or vertical blanking periods (this is to tell the top-level module to pull the color data low and display black). This module is parameterized to the point where it's generic enough to work with any resolution. All you need to do is override the timing parameters when you instantiate the module, and make sure to pass in the correct pixel clock. In this design, we are using one of the Max 10's PLLs to take the 50MHz input clock and divide it down to 33MHz.

The design of the module itself is relatively simple. Using the passed in parameters (that have a 1:1 relationship with the parameters in table 1), the sync generator calculates at what time to display the sync pulses. To know what it's currently displaying, the sync gen uses a pair of counters: one representing the current x coordinate (which pixel in the line we're currently on) and one representing the current y coordinate (which line we're currently on). When the horizontal counter reaches the end of the line, it resets back to zero and increments the vertical counter. Once the vertical counter reaches the end of the frame, it resets back to zero and the process starts over.

Avalon Interfaces

The LCD Driver module contains an Avalon Memory-Mapped Slave interface to relay status information back to a processor as well as an Avalon Streaming Sink interface to receive color data to display. The memory-mapped slave consists of a single register at address "1" that returns back the current state of the horizontal and vertical blanking signals in the two least significant bits.

The Avalon-ST (streaming) interface follows a very simple handshaking protocol to pass data between the source and the sink. Whenever the source has data available for the sink, it asserts a "valid" signal. Whenever the sink is able to receive data, it outputs a "ready" signal. Whenever both of those signals are asserted at the same, data is passed from the source to the sink.

The LCD Driver is ready to receive data any time it's not blanking, and the asi_in0_ready signal is asserted accordingly. The data signal (asi_in0_data) is tied to the red, green, and blue output signals whenever blanking isn't occurring (otherwise the color signals are tied to ground).

For more information on exactly what every signal within these two interfaces does, please consult the Avalon Interface Specifications document on the Altera website.

Framebuffer Reader

This module is responsible for two tasks:

1. Reading out the framebuffer from memory and converting it into a stream of pixels that the LCD Driver module can understand.
2. Performing clock-domain crossing data synchronization between the 100MHz system clock and the LCD Driver's 33MHz pixel clock (described in the LCD Driver section).

A dual-clock FIFO primitive is used to perform both of these tasks at the same time. This dual-clock FIFO instantiates block-ram under the hood to store a buffer of pixels and also contains logic to synchronize the two clock domains together. The rest of the Framebuffer Reader module consists of a state machine that performs burst read transactions on the DDR3 memory (and stores the results in the FIFO) and some wires that translate the Avalon-Streaming interface into signals that perform read operations on the FIFO. This entire module is contained within a single Verilog file:

<Source Root>/IP/framebuffer_reader/framebuffer_reader.sv

Dual-Clock FIFO Primitive

Chapter 9 in the Max 10 Embedded Memory User Guide describes in great detail the FIFO primitive used within this design. As that chapter states, this FIFO will utilize M9K memory block resources within the Max 10 device. This dual-clock FIFO has been configured to have a depth of 64 words of 32-bits (size of a single pixel) along with an output signal called *wrusedw* that is used to check how full the FIFO is at any given time. That value is used within the burst read state machine to determine when to start a burst read so as to not overflow the FIFO (only start a burst read when the FIFO is half-full).

The *wrreq* and *wrclk* signals are used to write data into the FIFO at system clock speeds (100MHz) while the *rdreq* and *rdclk* signals are used to read from the FIFO at pixel clock speeds (33MHz). The *aclr* signal is used to asynchronously clear the FIFO of any pixel values at the end of the frame (when a *frame_sync* pulse has been detected). This ensures that no pixels from the previous frame are still inside of the FIFO during the next frame.

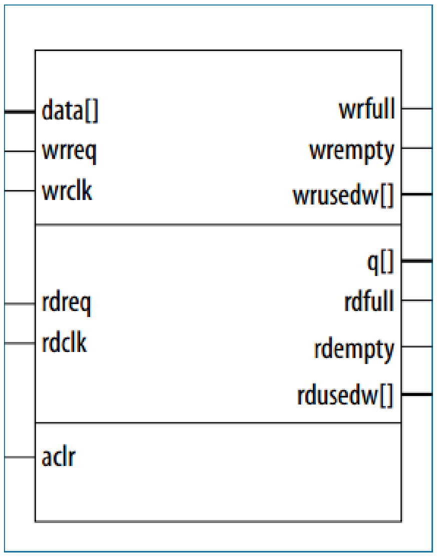


Figure 3: Block symbol for the Dual-Clock FIFO primitive

Frame Synchronization

After sending an entire frame of pixels to the screen, the framebuffer reader should clear out the contents of the FIFO and reset all of its counters back to zero to prepare for the next frame of video data. The frame synchronization logic takes in a "frame_sync" signal from the LCD Driver (this is just the vertical blanking signal) and synchronizes it to the system clock domain. When a positive edge of the frame_sync signal is detected, a soft reset signal is asserted. This reset forces the framebuffer reader to go back to pixel 0 and also clears out the FIFO of any leftover pixels. As you should recall from the VGA Interface section of this document, the vertical blanking period occurs right after a frame of data has been sent. By outputting a signal whenever the Vblank begins (the frame_sync signal), we can ensure the framebuffer reader is ready before the next frame needs to be sent.

Avalon Interfaces

The Framebuffer Reader module contains three Avalon interfaces:

1. **Avalon Memory-Mapped Master:** used to read pixel data from the framebuffer stored in DDR3.
2. **Avalon Memory-Mapped Slave:** allows the processor to set the base address of the framebuffer so the Avalon-MM Master knows where to get the pixel data from.
3. **Avalon Streaming Source:** sends the pixel data to the LCD Driver synchronized to the pixel clock.

For more information on exactly what every signal within these three interfaces does, please consult the Avalon Interface Specifications document on the Altera website.

Avalon Memory-Mapped Slave Interface

This slave interface allows the Nios II Processor to program in the base address of the framebuffer. This could also be turned into a fixed address that is known beforehand to both the framebuffer reader and Nios II processor, but making the value programmable gives more flexibility to the programmer. This allows a developer to dynamically allocate the block of memory needed to serve as a framebuffer (which generally isn't a fixed address) and to perform techniques like double-buffering.

Avalon Memory-Mapped Master Interface

This interface handles reading pixel data out of the framebuffer and storing it into the FIFO. To increase throughput from memory, 32-word burst transfers are used. A mealy state machine was created to handle performing burst transfers at the correct time (as to ensure the FIFO doesn't overflow).

The first state checks and waits for the FIFO to dip below half-full (at 64-words, half full would be 32-words, the same as our burst size) and then moves to the next state. The START_BURST state, as the name would suggest, outputs the signals required to begin a burst transfer. These are specifically the burstcount signal (used to tell the slave how many words we want to burst read), the address of the first pixel to read, and assertion of the read request line. After those signals are asserted, the state machine waits for the m0_waitrequest signal to go low, telling us that the slave (DDR3 memory) is about to start sending data.

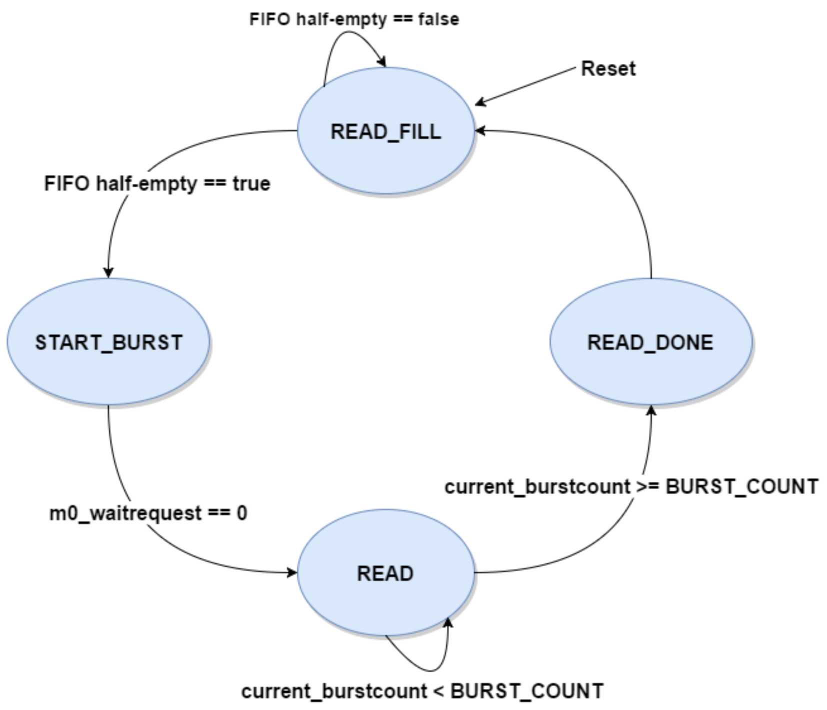


Figure 4: Burst read state machine diagram

The READ state is where the bulk of the work happens. The state machine stays inside of this state until it has read exactly 32 pixels. On any clock edge that the m0_readdatavalid is asserted (telling us that whatever data is on the m0_readdata line is valid), the write request line of the FIFO is asserted and the data is pushed into the FIFO. Once all of the pixels have been read, the state machine moves into its final state. The READ_DONE state just resets the current_burstcount variable (used to determine how many pixels have been read from memory any time during the burst transfer) and increases the current address offset by 128 bytes (32 pixels * 4 bytes/per pixel). These commands set up the state machine for the next burst transfer that needs to occur.

This state machine is split into three Verilog always blocks. The first is a sequential block that is used to clock in the next state logic and change which state the machine is currently in. The second always block is a combinational block that sets any outputs that don't need a clock to be set correctly. These outputs are control signals used on the Avalon bus, as well the next state the machine will move to. Lastly, another sequential block is used increment the current_burstcount variable whenever the state machine is inside of the READ state as well as to reset the offset and current_burstcount variables when the READ_DONE state is reached. This completely separates the combinational outputs of the machine and the support signals that change sequentially with the clock.

Avalon-Streaming Source

The transmit logic required to convert pixel requests from the LCD Driver into read requests on the FIFO is relatively simple. The Avalon-ST (streaming) interface follows a very simple handshaking protocol to pass data between the source and the sink. Whenever the source has data available for the sink, it asserts a “valid” signal. Whenever the sink is able to receive data, it outputs a “ready” signal. Whenever both of those signals are asserted at the same, data is passed from the source to the sink.

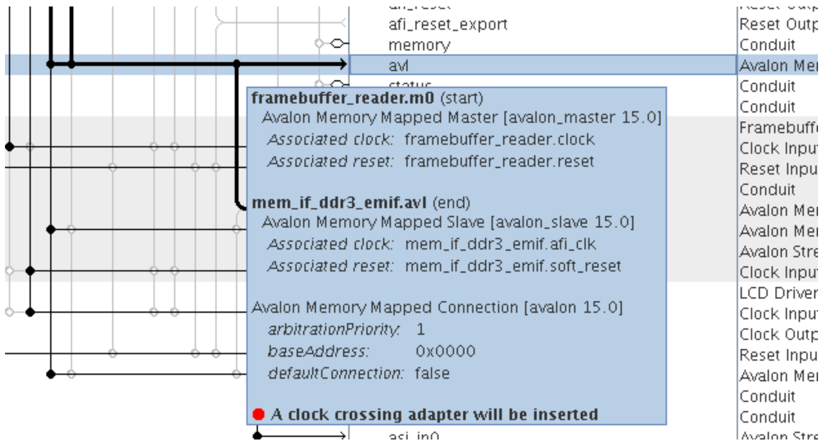
Whenever the FIFO isn’t empty, that means that we have data ready to send and we should output the valid signal (aso_out0_valid = ~fifo_rdempty). Whenever both the valid and the ready signal are asserted, we need to assert the FIFO’s read request line (fifo_rdreq = aso_out0_valid && aso_out0_ready). And lastly, the data signal for the streaming interface is wired directly to the read data interface of the FIFO (aso_out0_data = fifo_rddata). Nothing else needs to be done to conform to the streaming interface standard.

Memory Throughput

A major concern with any design that requires data in very concise time periods is whether or not data can be received and sent fast enough. In this design, we are storing the entire framebuffer within external memory. Naturally, there is a delay whenever we go through a DDR3 controller and off-chip to get to memory and it’s up to us as designers to make sure we can get data fast enough to not cause flickering on the LCD. There are two paths to the DDR3 memory: from Nios II to memory and from memory to Framebuffer Reader. Techniques were used to optimize both of those paths as described below.

Clock Domain Crossing

The system as a whole (Nios II, Framebuffer Reader, etc.) is running on a 100MHz clock being outputted from one of the Max 10’s PLLs. The DDR3 memory is required to run at 300MHz to meet spec, and the DDR3 controller exposes a half-speed (150MHz) Avalon control interface that we read and write to from within our Qsys system. When we connect modules up that have differing clocks, Qsys likes to be smart and automatically add in clock-crossing adapters for you without telling you. Unfortunately, by default, these adapters are only meant for low-throughput applications. They use a primitive handshaking system to pass data back and forth between the clock domains. For an LCD design, this will show up as flickering on the screen because the LCD Driver is displaying pixels faster than it’s receiving them.


```
framebuffer_reader.m0 (start)
  Avalon Memory Mapped Master [avalon_master 15.0]
   Associated clock:  framebuffer_reader.clock
   Associated reset:  framebuffer_reader.reset

mem_if_ddr3_emif.avl (end)
  Avalon Memory Mapped Slave [avalon_slave 15.0]
   Associated clock:  mem_if_ddr3_emif.afi_clk
   Associated reset:  mem_if_ddr3_emif.soft_reset

Avalon Memory Mapped Connection [avalon 15.0]
   arbitrationPriority:  1
   baseAddress:          0x0000
   defaultConnection:    false

● A clock crossing adapter will be inserted
```

Figure 5: If you hover over a connection, Qsys will tell you if it's inserting an adapter

To solve this problem requires changing a Qsys setting on the “Interconnect Requirements” tab called “Clock crossing adapter type.” This setting allows you to change how data is transferred between domains within an adapter. If you change the default setting of “Auto” to “FIFO” it will force Qsys to use a higher throughput dual-clock FIFO approach to clock-crossing. You could also manually instantiate “Avalon-MM Clock Crossing Bridge” components for finer control over the FIFOs that gets generated.

Nios II Code Optimizations

If you are using the Nios II processor to program pixel data individually into the external memory, then turning on code optimizations in the compiler settings can significantly increase the speed at which pixels get written into memory (and in turn, increase framerate). The reason for this is because most likely you are programming each individual pixel in a big loop. Meaning, a small set of assembly instructions are going to be ran nearly a millions times to fill up an entire framebuffer. By optimizing out even a tiny amount of those instructions, the entire operation of filling the framebuffer can be sped up tremendously. Remember to change the optimization settings for both the application project, and the BSP project. These settings can be found by right clicking the project->Properties->Nios II Application/BSP Properties. There will be an “Optimization level” setting on this page that you can set to “Level 3.” After that, all you need to do is recompile and download your software.

How to Compile the Hardware

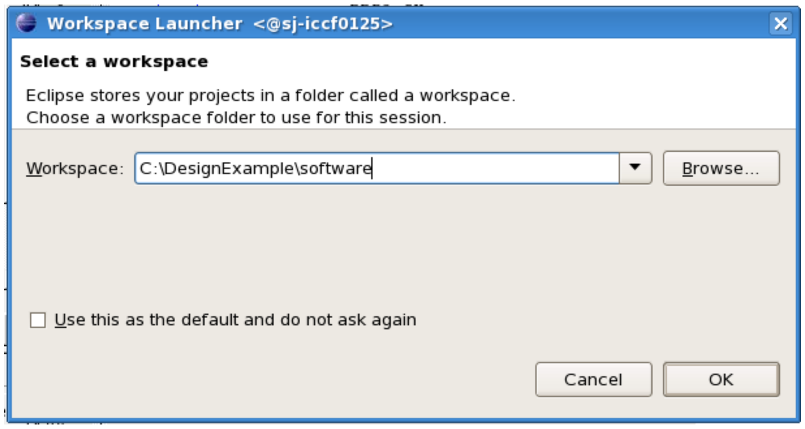
Follow the steps on the Design Store web page to extract and install the Frambuffer_LCD_Driver platform file. The following steps describe how to setup a project in the Quartus II software in order to program the MAX10 FPGA device with the design example.

1. Launch Quartus II and open the top.qpf project file using File->Open Project.
2. Note that for proper timing closure through TimeQuest, make sure the system.qip is listed prior to clock_constraints.SDC file or else the DDR3 clock does not get constrained properly.
3. Compile the project by clicking on Processing->Start Compilation.
4. Once the project is done compiling, launch the Quartus II Programmer from the tools menu (Tools->Programmer).
5. Plug in your development kit and make sure “USB-Blaster II” or “NEEK10” is shown next to the “Hardware Setup…” button. If it doesn’t, click on that button and select the correct programmer.
6. Once the programmer is selected, click on “Auto-Detect” in the programmer’s sidebar. If a dialog box appears asking for which device is on the JTAG chain, select “10M50DA”.
7. Double click in the File list where it says “<none>” and select the output_files/top.sof file.
8. Check the “Program/Configure” box then click “Start” to program the device.
9. After programming, follow the steps in the next section to program the software.

How to Compile the Software

The following steps will walk you creation of the software project and programming it to the device. The software directory contains the main.c and graphics.h source files.

1. Open up the Nios II Software Build Tools by clicking on Tools->Nios II Software Build Tools for Eclipse from within Quartus.
2. When the workspace selection window appears, browse to <design-example-installation-directory>\software and hit OK. For instance, if you extracted the example into C:\DesignExample, then the following image shows where your workspace will be located:



Workspace Launcher <@sj-iccf0125>

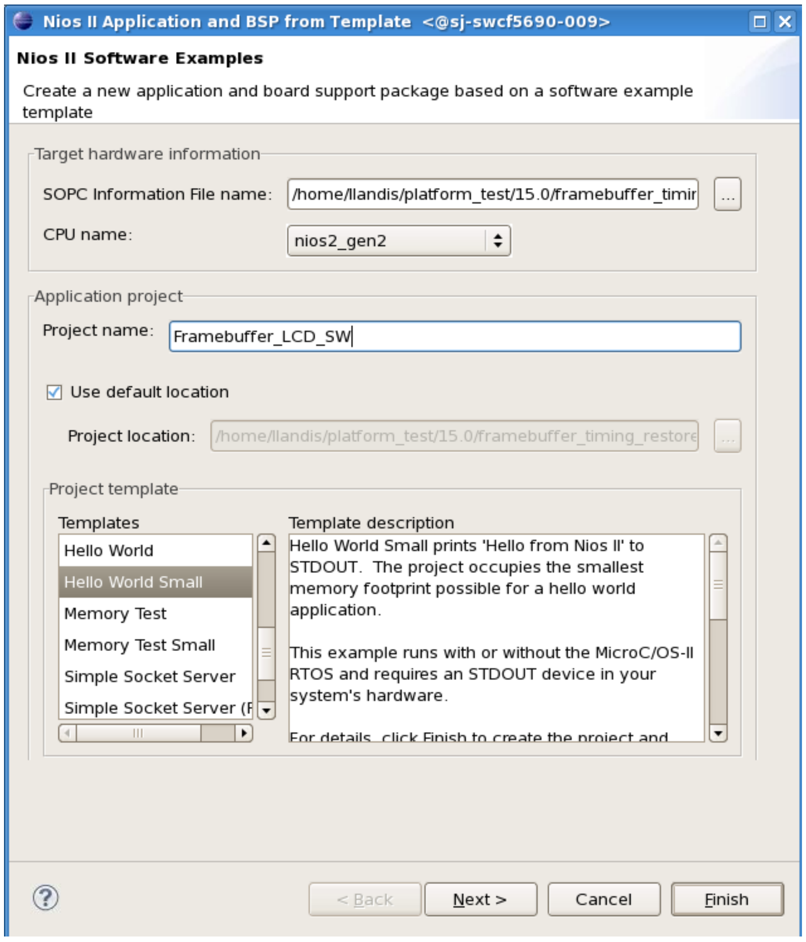
Select a workspace

Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\DesignExample\software

☐ Use this as the default and do not ask again

3. Next, go to Project Explorer, right click on New → Nios II Application and BSP from template.
4. Fill in the subsequent panel with your .sopcinfo file, a project name and use the template Hello World Small. Click Next and Finish.



Nios II Application and BSP from Template <@sj-swcf5690-009>

Nios II Software Examples

Create a new application and board support package based on a software example template

Target hardware information

SOPC Information File name: /home/llandis/platform_test/15.0/framebuffer_timir

CPU name: nios2_gen2

Application project

Project name: Framebuffer_LCD_SW

☑ Use default location

Project location: /home/llandis/platform_test/15.0/framebuffer_timing_restore

Project template

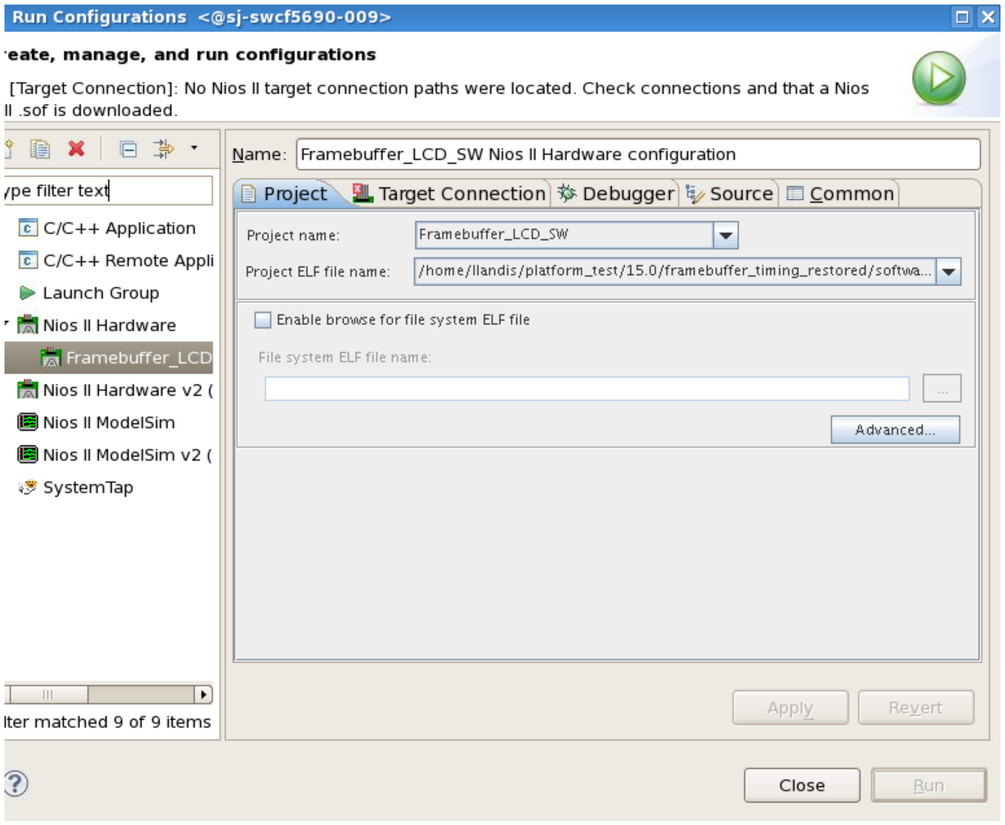
Templates: Hello World, Hello World Small (selected), Memory Test, Memory Test Small, Simple Socket Server, Simple Socket Server (F

Template description: Hello World Small prints 'Hello from Nios II' to STDOUT. The project occupies the smallest memory footprint possible for a hello world application.

This example runs with or without the MicroC/OS-II RTOS and requires an STDOUT device in your system's hardware.

For details, click Finish to create the project and

5. Now you need to remove the hello_world_small.c source file and add the main.c and graphics.h files to your project. Expand your Framebuffer_LCD_SW panel and right click on hello_world_small.c. Delete this file from your project.
6. Right click LCD_Framebuffer_SW. Click import. Navigate to General → Filesystem.
7. Browse to your software directory. Click ok.
8. Highlight graphics.h and main.c. Click ok.
9. Once the source files have been imported you can open up the example code by opening the main.c file within the Framebuffer_LCD_SW application project.
10. Build the project by going to Project->Build All. If you recently re-generated the Qsys project, you may need to re-generate the BSP by right clicking the BSP project->Nios II->Generate BSP.
11. To run the example, you need to set up a run configuration. Click on Run->Run Configurations. Next, right-click on "Nios II Hardware" in the sidebar and select "New." Under Project name, select the Framebuffer_LCD_SW project.


Run Configurations <@sj-swcf5690-009>

eate, manage, and run configurations

[Target Connection]: No Nios II target connection paths were located. Check connections and that a Nios II .sof is downloaded.

Name: Framebuffer_LCD_SW Nios II Hardware configuration

Project name: Framebuffer_LCD_SW

Project ELF file name: /home/llandis/platform_test/15.0/framebuffer_timing_restored/softwa...

Enable browse for file system ELF file

File system ELF file name:

12. Next, click on the "Target Connection" tab and hit "Refresh Connections" if the Max 10 device doesn't show up (you may have to scroll to the right to see that button). If no connections appear, make sure the board is both plugged in and already programmed with the .sof file as described in the previous section. Once that's connected, click on Apply and then Run.