Using Excalibur DMA Controllers for Video Imaging

February 2003, ver. 1.1 Application Note 287

Introduction

The Altera® Excalibur™ devices provide you with a complete system-on-a-programmable chip solution. Excalibur devices contain an embedded stripe subsystem comprising an ARM922T™ processor, on-chip SRAM, and an SDRAM controller, among other peripherals. In addition to the embedded stripe, the FPGA array provides you with the flexibility to meet the system requirements for today's high-end embedded systems.

This application note discusses the benefits of using direct memory access (DMA) controllers in Excalibur devices for implementing applications in video imaging. Different types of DMA solutions that complement the embedded stripe bus architecture are presented along with a discussion on how these solutions improve overall system performance in Excalibur devices. In addition, this application note discusses the implementation details of a DMA controller reference design that transfers video images stored in SDRAM to a video graphics array (VGA) driver implemented in the FPGA portion of the Excalibur device.

The DMA controller reference design implements a graphics display system in an Excalibur device, which you can use in applications ranging from automotive to gaming or any other application that requires a graphics display. The design implements VGA resolutions up to 640 x 480 without you making any modifications to the hardware. The reference design fits in any of the three Excalibur devices, uses minimal system resources, and runs in excess of 100 MHz.

This application note assumes that you are familiar with the Excalibur device architecture and that you are also familiar with the advanced microcontroller bus architecture (AMBA™) high-performance bus (AHB) specification.

For more information on the Excalibur devices, refer to the *Excalibur Devices Hardware Reference Manual.*

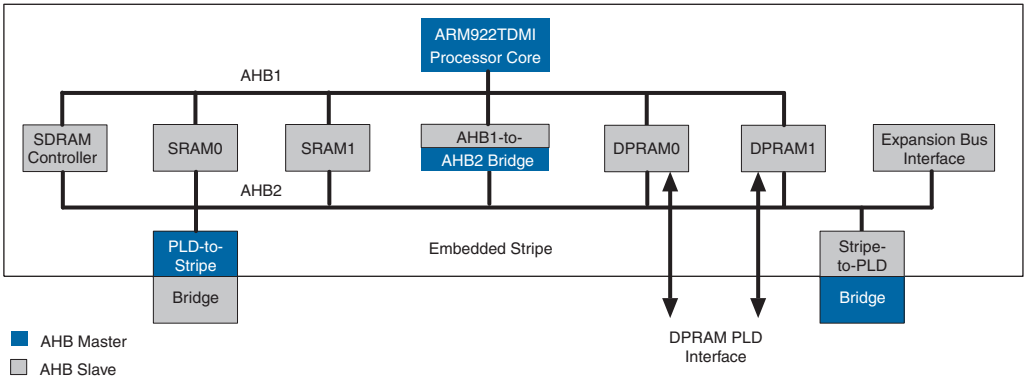
Excalibur Devices

This section discusses the Excalibur devices' embedded stripe bus architecture and DMA controllers.

Embedded Stripe Bus Architecture

The Excalibur embedded stripe bus architecture comprises two AHBs—AHB1 and AHB2. AHB1 is the main processor bus and is used to interface the ARM922T embedded processor to all of the memory components in the stripe. The AHB2 is clocked by a divide by 2 version of the AHB1 clock and lower speed peripherals reside on the AHB2. These two buses provide the embedded stripe peripherals with an efficient means of processing data. Figure 1 shows a block diagram of the Excalibur bus architecture.

Figure 1. Excalibur Bus Architecture



The embedded processor is the sole master on AHB1 and has direct access to all of the memory components in the embedded stripe. The embedded processor can access slaves implemented in the FPGA array via the stripe-to-PLD bridge and masters in the FPGA can access slaves on AHB2 via the PLD-to-stripe bridge.

For more information on the embedded stripe bus architecture, refer to the *Excalibur Devices Hardware Reference Manual*.

For information on FPGA bus implementations, refer to *AN181: Excalibur Solutions—Multi-Master Reference Design*.

To maximize the bus utilization and overall system performance of Excalibur-based designs, the embedded processor minimizes the number of accesses it makes to buses other than AHB1. Using a DMA controller in the FPGA array, to move data between system components, greatly reduces the number of accesses that the embedded processor makes outside of the AHB1, and it increases the utilization of both AHB1 and AHB2.

DMA Controllers

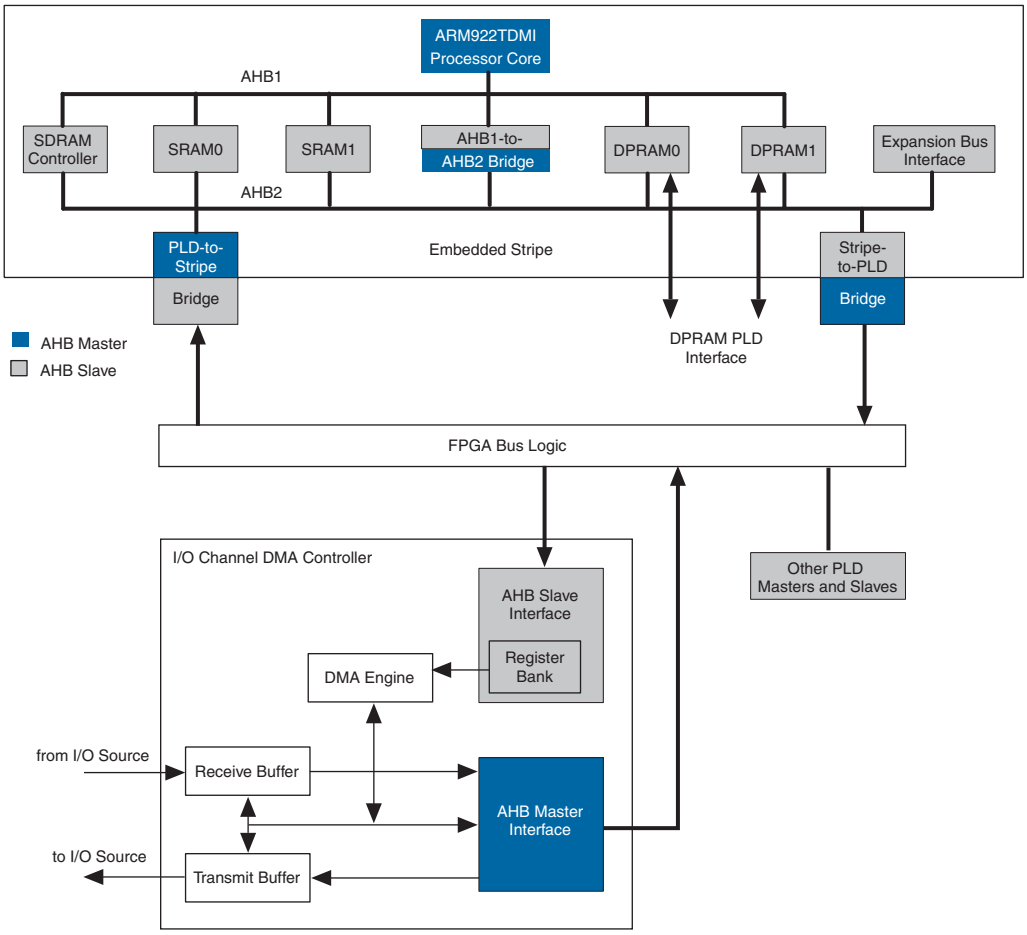
Because Excalibur devices are programmable, you can implement DMA controllers that are customized to maximize performance for individual applications. For example, some applications need to transfer blocks of data from I/O channels to system memory, while other applications require that data is moved from system memory to custom logic or from one memory location to another. You can implement each of these types of DMA controllers in Excalibur devices.

I/O Channel—Memory DMA and Memory-to-Memory DMA

Many applications transfer data between I/O channels and memory. Using a processor to handle the flow of data between I/O channels and memory typically wastes processor cycles, as the I/O channels tend to run much slower than the processor. In some applications, the total amount of memory space that is needed for a transfer from an I/O channel fits within the the embedded stripe dual-port RAM (DPRAM) blocks. In these cases, the I/O channel connects to the DPRAM with a small amount of control logic, which controls the addressing. Then the processor operates on the data that has been stored in the DPRAM directly from the I/O channel.

However, there are applications where the required memory space does not fit within the DPRAM. You can use DMA controllers in these situations, because they relieve the processor of the I/O handling workload and they can transfer data to any system peripheral, including SDRAM. I/O-to-memory type DMA controllers can be designed in the FPGA array and they require very little processor intervention. Figure 2 shows an I/O-to-memory controller implemented in an Excalibur device.

Figure 2. I/O to Memory DMA Controller



The I/O DMA-to-memory DMA controller (see Figure 2) has both an AHB master and slave interface. The slave interface allows the processor to configure the DMA controller's register bank, which provides information on the type of transfer to be executed, such as the destination address and total byte count. Processor accesses to AHB2 are greatly reduced by using the slave interface, because the processor does not have to perform any transactions on AHB2, apart from setting up the register bank once per DMA data transfer. All of the memory elements on the embedded stripe have paths to both AHB1 and AHB2. This allows data to be moved by the master interface of the DMA controller to any slave on the AHB2 bus; the processor then has direct access to that data via the faster AHB1 bus thus increasing the overall system performance.

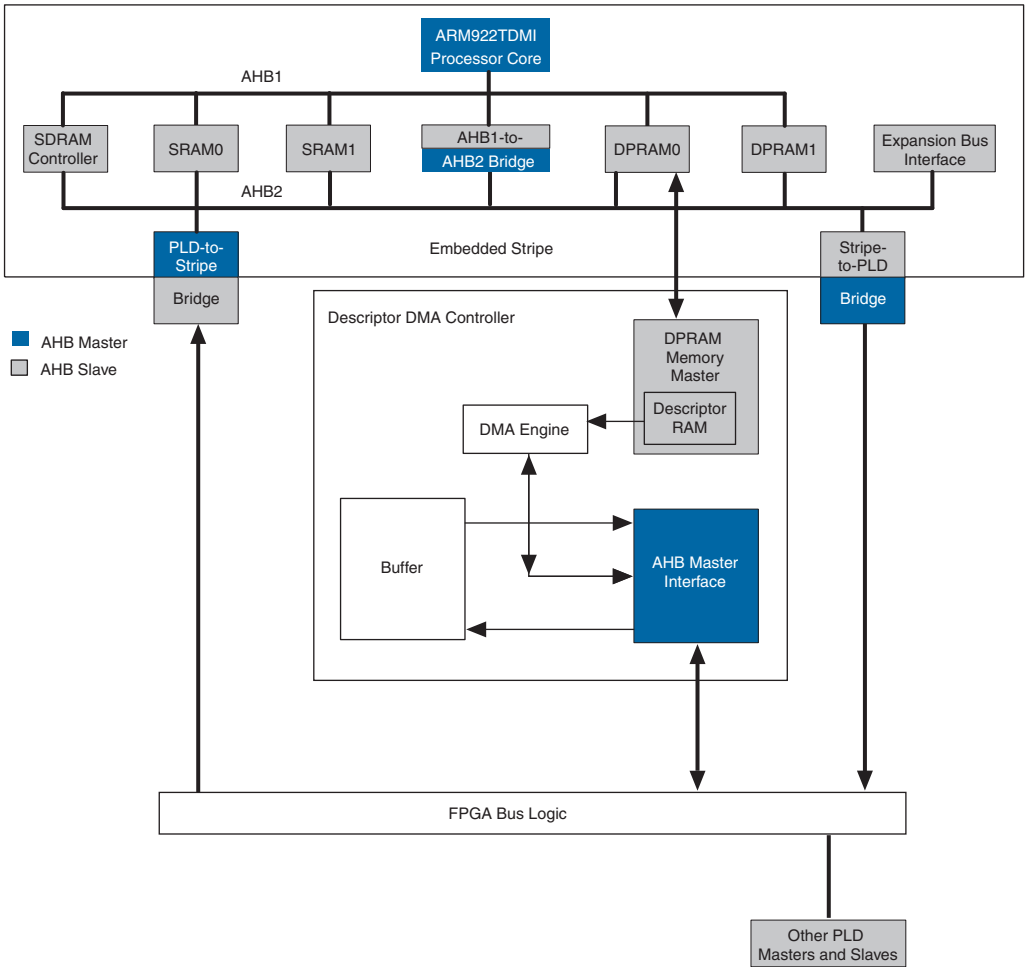
This same example can be reduced to a simple memory-to-memory DMA controller by removing the receive and transmit buffers (see Figure 2). In this case, rather than receiving data from off chip, the DMA controller is responsible for moving data to and from memory to other peripherals in the system. This memory-to-memory peripheral DMA controller is conceptually quite simple and can greatly improve performance in bulk data transfers between the stripe and the FPGA for some applications. However, it is possible to design even higher performance controllers that take further advantage of the architecture of the embedded stripe.

Descriptor Based DMA Controller

In the examples described above, the configuration details on the transfers that are executed by the DMA controller are stored in a register bank within the FPGA array. When doing multiple transfers the processor has to reload the register bank with the next configuration data every time it wants to issue a DMA transaction. In some cases this can cause system bottlenecks as the processor may have to wait for the DMA controller to relinquish ownership of the AHB2 bus before the next transaction configuration can be stored in the register bank. One way to alleviate this problem is to use a descriptor based DMA controller and store the descriptors in DPRAM thereby eliminating the need for the processor to communicate with a register bank over the AHB2 bus.

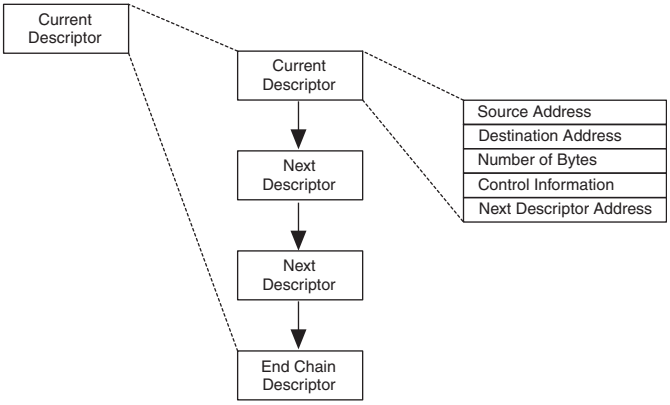
Figure 3 shows a block diagram of a DMA controller that stores descriptor chains in DPRAM.

Figure 3. Descriptor Based DMA Controller



Descriptor chains describe a sequence of transfers for the DMA controller to perform and are written to the DPRAM by the processor via the AHB1 Bus. Individual descriptors contain information pertaining to the size, source and destination of the transfer and a pointer to the next descriptor in the chain. A descriptor is read from the DPRAM by the DPRAM memory master component of the DMA controller. The DMA controller executes the transfer described by the current descriptor and upon completion of the transfer it writes a flag back to that descriptor, which indicates that the transfer is completed (see Figure 4). After completion of the current descriptor, the DMA controller reads the pointer to the next descriptor. The DMA controller executes the next descriptor and continues to do this until it reaches the end of the descriptor chain.

Figure 4. Descriptor Execution



Descriptor based DMA controllers significantly increase performance over previous implementations, as the processor does not have to send transactions to the FPGA via the bridges. More importantly, the processor sets up a number of transactions to be processed, proceeds to execute other instructions, and does not have to wait on the DMA unless the data transferred is needed.

DMA Controller Variant

Many DMA controller variants can be created to meet different system requirements due to the flexibility of the FPGA array and the high memory content present in Excalibur devices. Applications often need to buffer large amounts of ingress data then have the data operated on and be transmitted elsewhere in the system. Using similar DMAs (as previously described) and the DPRAM in Excalibur devices a system can be constructed (see Figure 5).

Figure 5. DMA Variant

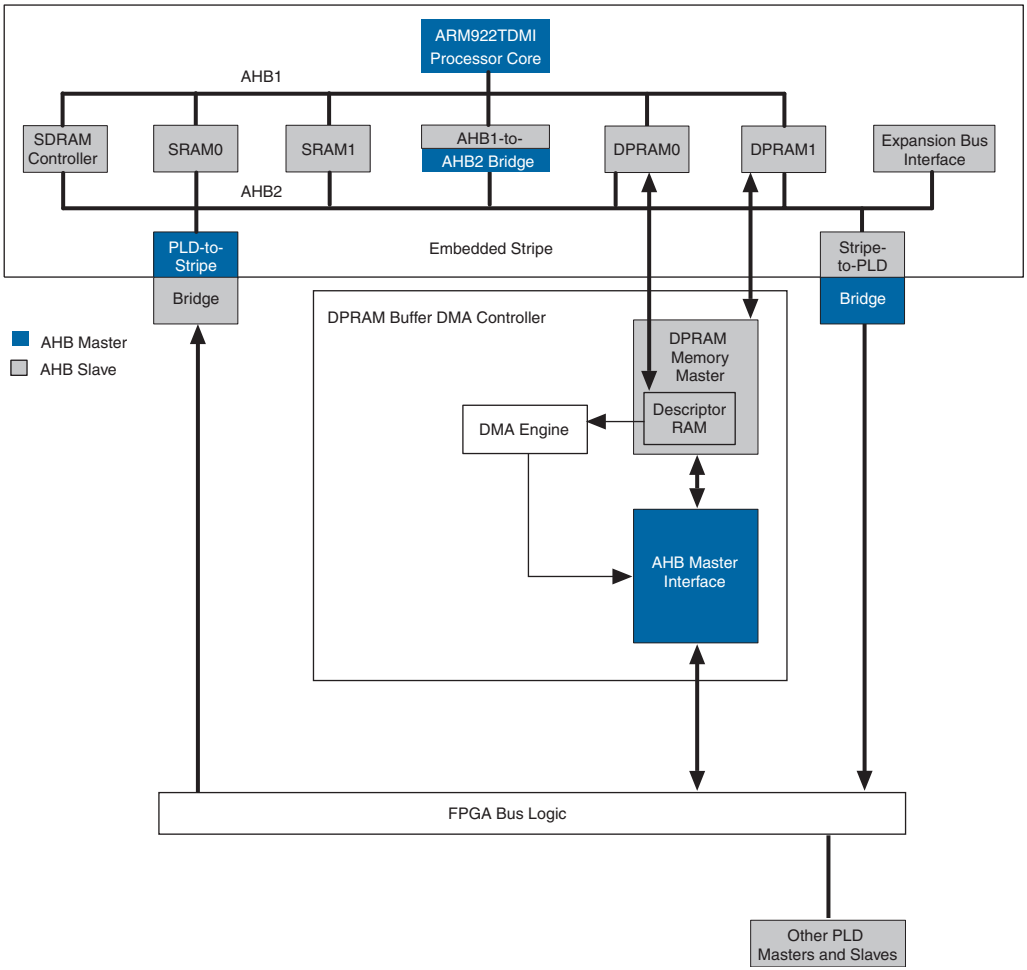


Figure 5 shows a DMA controller that makes use of the DPRAM as its back end storage space. You can use the DPRAM to store large amounts of data transmitted from serial I/O channels or buffer a set of data from elsewhere in the system. The data stored in the DPRAM can then be directly modified by the processor via the AHB1. After the processor has finished modifying the data the DMA controller can transfer it to its final destination. Excalibur devices contain two blocks of DPRAM, which provides up to 256 Kbytes of storage space on EPXA10 devices. You can use one block or both blocks of the DPRAM as data buffers and one block as a storage location for descriptors.

DMA Controller Reference Design

This section discusses the following areas of the DMA controller reference design:

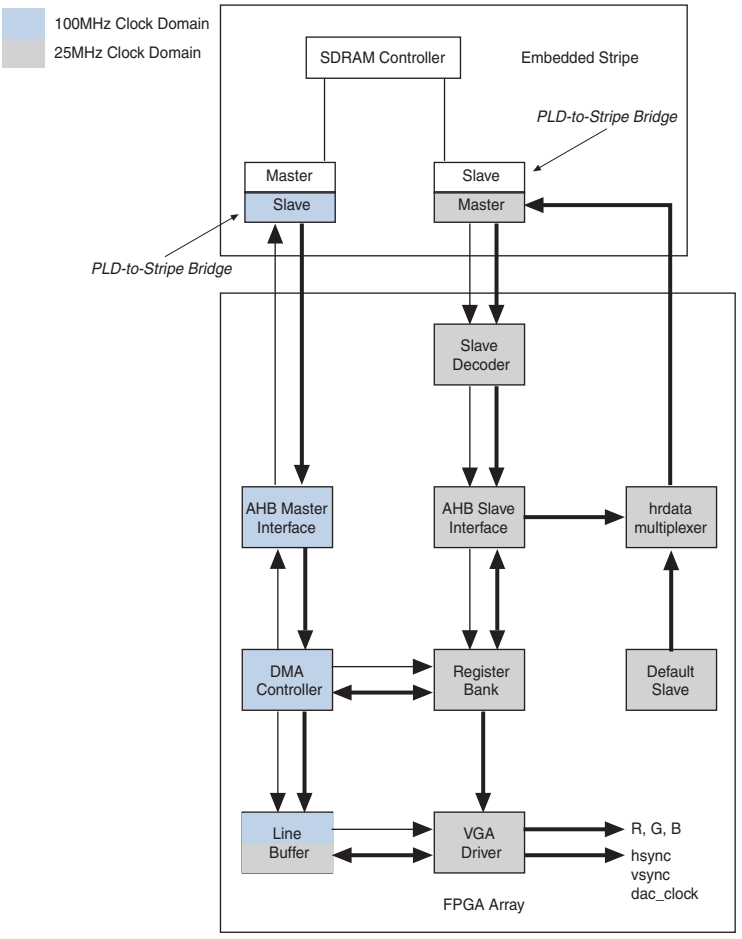
- Functional Description
- External Hardware
- Test Software
- Installation
- Using the Reference Design
- Simulate the Reference Design
- Convert .bmp Files to .hex Files for storage in SDRAM

Functional Description

The DMA controller implemented in this reference design is a variation on the memory-to-memory DMA controller. Its interface to the stripe is identical to the the memory-to-memory DMA controller. However, the control logic for this DMA controller has been designed to be synchronous with the operation of a VGA driver.

Figure 6 shows a block diagram that illustrates the various reference design components and component interconnections.

Figure 6. Video Driver Block Diagram



The AHB slave interface contains a register bank that is used to configure the operation of both the DMA controller and the VGA driver. The DMA controller reads images from SDRAM and transfers each line of the image to a line buffer, which feeds the VGA driver. The operation of this DMA controller is dependent on the values written into the slave interface and video timing signals generated by the VGA driver. The VGA driver reads images from the line buffer, generates horizontal and vertical timing signals, and outputs the pixels at the appropriate time to the video output signals.

Design Specifications

The reference design video driver perform the following actions:

- Generates the appropriate timing signals to display 640 × 480 images
- Supportes a double pixel size mode, where 1 pixel in memory translates to a 2 × 2 enlarged pixel on the monitor
- Displays variable size images with a maximum size of 640 × 480
- Converts 16-bit input pixels (65,536 colours) to 24-bit output pixels
- Outputs status information to the host processor
- Communicates with the embedded stripe via AHB master and slave interfaces
- Clocks video output data at 25 MHz clock

The video driver generates VGA timing signals based on a 25 MHz clock and the driver always outputs a 640 × 480 image to the screen. However, to conserve memory bandwidth, you can configure the driver to read smaller images from memory and place a black border around the images, which generates a full size 640 × 480 image.

This design implements a 16-bit resolution 640 × 480 VGA driver. Each 16-bit input pixel represents a lossily compressed pixel which is 1 s padded by the VGA driver to produce a 24-bit RGB output.

VGA Driver

The reference design VGA driver generates timing signals for a 640 × 480 image running at 60 frames per second. Set-up information for the driver, including the image size and output enable signal, is passed from the embedded processor to the VGA driver via the AHB slave interface. The VGA driver begins driving pixels to the screen, when it detects the processor has written an enable signal to the slave interface. Pixels are transferred from SDRAM to the VGA driver via a line-buffer that is filled by the DMA controller. Each word from the line buffer is 32-bits and contains 2 pixels. The pixels are in a 5-6-5 RGB format where the lower 3 bits of the red and blue spectrums and the lower 2 bits of green spectrum have been lopped off. These 16-bit compressed pixels must be converted to 24-bit pixels, to be processed by a video DAC. Each of the 3 colours are padded with 1s to assemble the 24-bit output pixels. Padding the lower order bits of the 16-bit inputs with 1s ensures that the output pixels are brighter than if the lower order bits where packed with 0s.

To display an image, CRT monitors typically require 5 signals: R, G, B, hsync, and vsync. The R, G, and B signals represent the weight of the red, green and blue signal elements that compose a pixel. The hsync signal provides the monitor with a horizontal synchronization signal; the vsync signal provides a vertical synchronization signal to the monitor. The VGA driver in this design also generates two additional synchronization signals that indicate when pixels are active. These signals can be used by logic elsewhere in the design. These two signals are the hblank and vblank signals. The hblank signal is asserted whenever pixels are being actively driven to the screen. Similarly, the vblank signal is asserted whenever active lines are being driven to the screen. Figures 7 and 8 shows the relationship between each of these signals.

Figure 7. 640 × 480 VGA Horizontal Timing

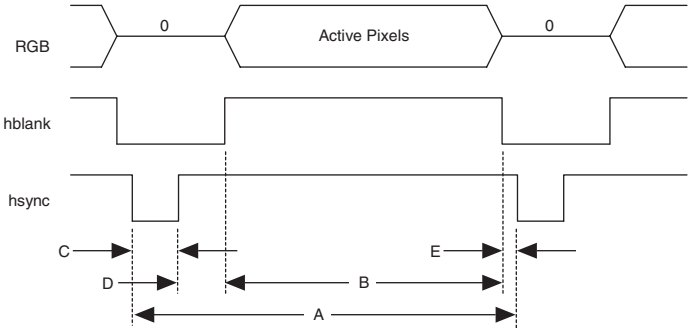


Table 1 shows the 640 × 480 VGA horizontal timing.

Table 1. 640 × 480 VGA Horizontal Timing

Parameter	Description	Time (µs)
A	Line scan period	31.77
B	Active video period	25.17
C	Sync period	3.77
D	Back porch	1.89
E	Front porch	0.94

Figure 8. 640 × 480 VGA Vertical Timing

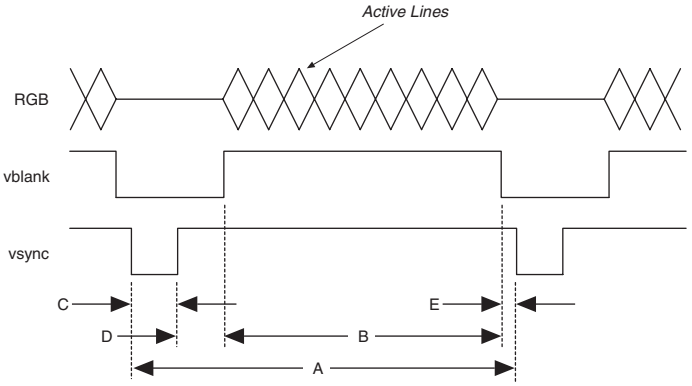


Table 2 shows the 640 × 480 VGA vertical timing.

Table 2. 640 × 480 VGA Vertical Timing

Parameter	Description	Time (ms)
A	Frame period	16.68
B	Active video period	15.25
C	Sync period	0.064
D	Back porch	1.02
E	Front porch	0.35

Many applications do not require full 640 × 480 screen resolution. Applications that need to display smaller images are supported through the use of the `image_dimensions` register that is included in the slave interface. The `image_dimensions` register is integrated into the timing generation circuitry of the VGA driver. The timing circuitry consists of a pixel counter and a line counter that keep track of the current position of a pixel on the screen. These counters determine the timing of the synchronization signals and they also synchronize the sending of the address signal to the line buffer. When the `image_dimensions` register indicates that the image size is smaller then 640 × 480 the synchronization logic modifies the duration of the `hblank` and `vblank` signals. Images that are smaller than 640 × 480 have shorter duration `hblank` and `vblank` signals, such that the image is centered in the middle of the display with a black border around the image.

Another feature related to driving smaller images is the VGA driver's capability of doubling the pixel size of smaller images. For example, when set to double pixel mode a 320 × 240 sized image in memory appears as a 640 × 480 image on the monitor. This picture enlargement is accomplished by reading a line stored in the line buffer twice and by also reading each pixel stored within a line twice.

The line buffer that feeds the VGA driver is filled by the DMA controller. This line buffer is filled whenever the `vblank` signal is asserted and a negative edge transition is detected on the `hblank` signal. Therefore, the `hblank` and `vblank` signals must be modified to have a shorter duration, when smaller images are being displayed. Otherwise the line buffer is filled with invalid information, when border pixels are driven and memory bandwidth is wasted. When the VGA driver is configured in double pixel mode, video lines are read twice from the line buffer but the `hblank` signal is only asserted for the first line read. This ensures that memory bandwidth is not wasted by writing the same video line twice into the line buffer. Figures 9 and 10 illustrate the horizontal and vertical timing of the VGA driver when a smaller image size is used.

Figure 9. VGA Horizontal Timing for Images Smaller than 640 × 480

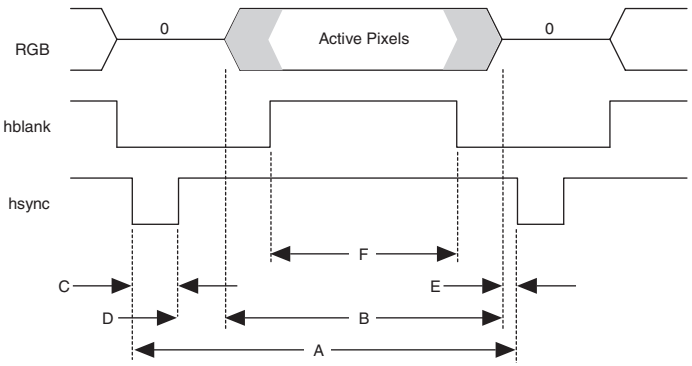


Table 3 shows the VGA horizontal timing for images smaller than 640 × 480.

Table 3. VGA Horizontal Timing for Images Smaller than 640 × 480

Parameter	Description	Time (µs)
A	Line scan period	31.77
B	Active video and border period	25.17
C	Sync period	3.77
D	Back porch	1.89
E	Front porch	0.94
F	Active video period	25.17 – t_{BORDER} *(1)*

Note to Table 3:
(1) t_{BORDER} = 25.17 µs/640 × (640 – *num_pixels_per_line*).

Figure 10. VGA Vertical Timing for Images Smaller than 640x480

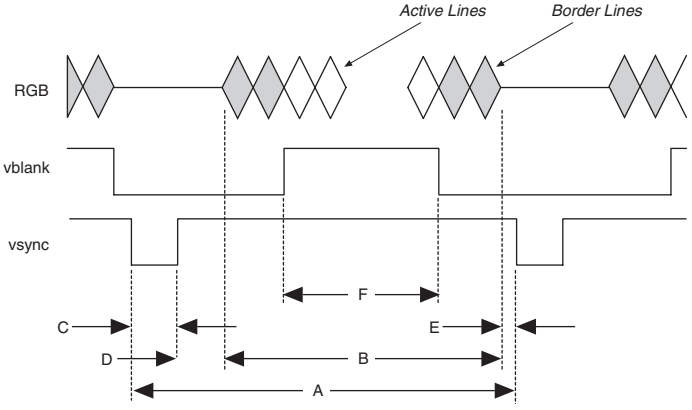


Table 4 shows the VGA horizontal timing for images smaller than 640 × 480.

Table 4. VGA Vertical Timing for Images Smaller than 640 × 480

Parameter	Description	Time (ms)
A	Frame period	16.88
B	Active video and border period	15.25
C	Sync period	0.064
D	Back porch	1.02
E	Front porch	0.35
F	Active video period	15.25 – t_{BORDER} *(1)*

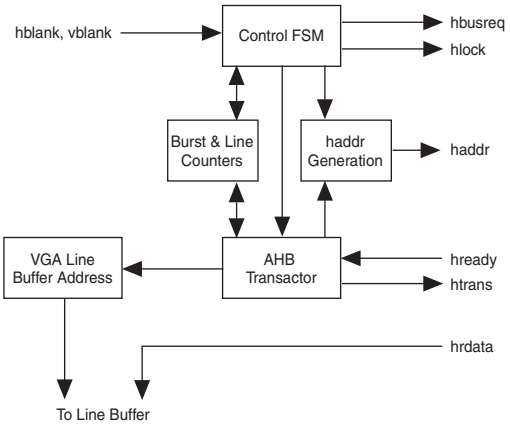
Note to Table 4:
(1) t_{BORDER} = 31.77 µs × (480 – *num_lines*)

DMA Controller

Transmitting video images from SDRAM to a VGA driver typically consumes a large amount of memory bandwidth. Memory bandwidth requirements are increased even more, if the processor has to update a video frame in memory, transfer a frame from memory to the VGA driver, and store and run application code all from a unified memory structure. Using a DMA controller to transfer data from SDRAM to the VGA driver greatly improves the performance of Excalibur-based systems that use graphics. The DMA controller allows the processor to focus on updating the next frame and running other program code, without having to take care of the bandwidth-intensive task of transferring images from SDRAM to the video driver.

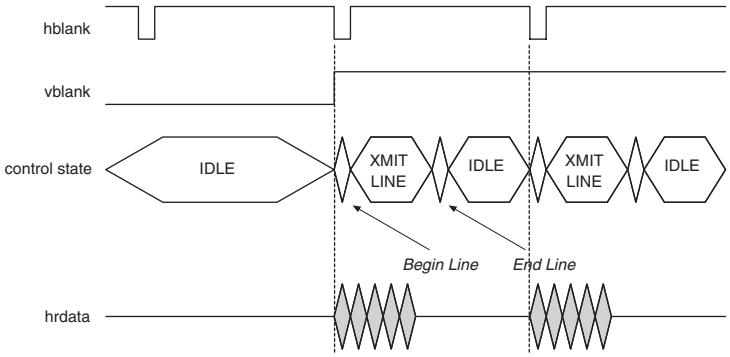
The reference design DMA controller runs at 100 MHz and is configured by the slave interface component. Configuration options for the DMA controller include the base addresses of the frame buffers, the size of the image to be transferred, and an enable signal for the DMA controller. All of the DMA controller's configuration registers power on to `0`, so the software must configure these registers to the appropriate settings, for the DMA controller to extract an image from memory. Figure 11 shows a block diagram for the DMA controller.

Figure 11. DMA Controller Block Diagram



Central to the operation of the DMA controller is a control logic finite state machine (FSM). The control FSM monitors the video enable input from the AHB slave interface and the VGA driver `hblank` and `vblank` signals. After the DMA controller has been enabled and the VGA driver is ready to output a line, the control FSM sequences into a transmit line state and remains in that state until an entire line has been read into the line buffer. Upon completion of a line transfer from SDRAM to the line buffer, the control FSM enters an idle state. When in the idle state the control FSM continues to monitor the VGA `hblank` and `vblank` signals and exits the idle state when a new line is starting. The control FSM also keeps track of the number of lines that have been transmitted in the current frame. When the correct number of lines has been transmitted to the line buffer, the control FSM asserts an interrupt signal and enters an extended idle state and remains idle until the VGA driver signals that the next frame is beginning. Figure 11 shows the signal transitions of a typical line transfer from SDRAM to the line buffer and the VGA driver.

Figure 12. Line Transfer Timing



The control FSM enters the transmit line state when the `vblank` signal is asserted and a falling edge transition is detected on the `hblank` signal. When the FSM detects that a line is starting it enters the begin line state, which requests access to the SDRAM over the PLD-to-stripe bridge. Assuming that the DMA controller gains access to the SDRAM, bursts of data begin to be transferred to the line buffer via the `hrdata` bus of the PLD-to-stripe bridge.

The PLD-to-stripe bridge facilitates communication between the FPGA and the embedded stripe using a standard AHB interface. The DMA controller portion of this reference design contains an AHB transactor FSM, which moves data across the PLD-to-stripe bridge. Upon receiving a start signal from the control FSM, the AHB transactor FSM initiates an unspecified length locked burst transfer across the PLD-to-stripe bridge. Unspecified burst length transfers are desirable for video applications, because they provide the best bandwidth across the PLD-to-stripe bridge—the bridge does not need to synchronize between clock domains many times during a single video line. The AHB transactor continues the unspecified length burst read until a 1K address boundary is encountered or the control FSM signals that a complete line has been written into the line buffer. Transactions that cross 1K address boundaries are not supported by the AHB specification; therefore, you must stop an unspecified burst on the last address location before a 1K boundary. To prevent bus lock ups due to crossing a 1K word boundary, the AHB transactor monitors the `haddr` bus, stops the current transfer, and initiates a new transfer, if it detects that the `haddr` bus holds a value of `0xXXXXX3FC`, `0xXXXXX7FC`, or `0xXXXXXBFC`. Each of these addresses represents the 32-bit address immediately preceding a 1K address boundary.

This reference design can run on any of the Excalibur devices. Some Excalibur devices support 16-bit SDRAM and others support 32-bit SDRAM. The SDRAM controller included in the embedded stripe reads memory from SDRAM in 8 native word bursts. Therefore, for 16-bit SDRAM, an unspecified length burst read across the PLD-stripe-bridge comprises a number of 4 32-bit word bursts. For 32-bit SDRAM an unspecified length burst read comprises a number of 8 32-bit word bursts. Either case is supported by the reference design, as the control FSM is sensitive to the total number of burst beats transferred across the bridge.

The DMA controller is also responsible for signalling the processor when a new frame is starting. The controller asserts an `irq` signal when the last pixel of the current frame has been sent. This allows the processor the maximum period of time to safely update the frame buffer in SDRAM before the DMA controller begins to read the next video frame. The `irq` signal is cleared by writing the next frame buffer address to the `buffer_address` register inside the slave interface.

Slave Interface

The AHB slave portion of the design configures the operation of the DMA controller and the image size of the VGA driver. Inside the AHB slave is a simple AHB interface that decodes AHB transactions from the stripe-to-PLD bridge and converts them into control and data signals that access an 8 word register bank. The AHB interface supports 32-bit single and burst transactions. The register bank contains control and status registers that are used to setup and maintain the operation of the DMA controller and the VGA driver. You can modify this AHB slave design for use with any other system designs that require an interface to the stripe-to-PLD bridge.

The reference design also incorporates a slave decoder and a default slave. Default slaves should be included in any AMBA AHB systems whose peripherals do not occupy the entire addressable memory space. Using a default slave can prevent bus lockups or reading of incorrect data as whenever an AHB transaction attempts to read or write data outside of the allowable address range the default slave signals an error response to the initiating master.

There are two other AHB components that are also needed to interface the register bank to the embedded processor: the slave decoder, and the data and response multiplexer. The slave decoder monitors the address signal coming from the stripe-to-PLD bridge. It also generates select signals for the AHB slave interface or for the default slave depending on the address presented to the decoder. The multiplexer creates the `hrdata`, `hresp` and `hready` signals that are sent to the stripe-to-PLD bridge. The data and response multiplexer receives its data inputs from the register bank peripheral and from the default slave.

Table 5 shows the registers for the slave interface. .

Table 5. Registers

Name	Mnemonic	Address (H)	Access
Buffer address	BUFFER_ADDRESS	00	Read/Write
Image dimensions	IMAGE_DIMENSIONS	04	Read/Write
Control	CONTROL	08	Read/Write
Current addresss	CURRENT_ADDRESS	0C	Read
Status	STATUS	10	Read
Reserved	–	14	–
Reserved	–	18	–
Reserved	–	1C	–

BUFFER_ADDRESS (peripheral base + 0x00)

Table 6 shows the buffer address register format.

Table 6. Buffer Address Register Format

Data Bit	Mnemonic	Description
31:0	HADDR	32-bit base address in the frame buffer memory.

IMAGE_DIMENSIONS (peripheral base + 0x04)

Table 6 shows the image dimensions register format.

Table 7. Image Dimensions Register Format

Data Bit	Mnemonic	Description
15:0	NUM_LINES	The total number of lines (max of 480 in normal mode, max of 240 in double mode).
31:16	NUM_PIXEL_PER_LINE	The number of pixels in each line (max of 640 in normal mode, max of 320 in double mode, must be divisible by 16).

CONTROL (peripheral base + 0x08)

Table 6 shows the control register format.

Table 8. Control Register Format

Data Bit	Mnemonic	Description
0	M	Video mode. 0 = normal mode, 1 = double pixel size mode.
1	E	Enable the VGA driver and the DMA controller: 1 = enable, 0 = disable.
31:2	Reserved	Write all 0s to ensure future compatibility.

CURRENT_ADDRESS (peripheral base + 0xC)

Table 6 shows the current address register format.

Table 9. Current Address Register Format

Data Bit	Mnemonic	Description
31:0	HADDR	32-bit base address in memory that is currently being read by the DMA controller.

STATUS (peripheral base + 0x10)

Table 6 shows the status register format.

Table 10. Status Register Format

Data Bit	Mnemonic	Description
0	HB	Horizontal blanking signal from the VGA driver. A 1 indicates that a line is being driven.
1	VB	Vertical blanking signal from the VGA driver. A 1 indicates that the image is being driven.
31:2	Reserved	Write all 0s to ensure future compatibility.

RESERVED

Table 6 shows the reserved registers format.

Table 11. Reserved Register Format

Data Bit	Mnemonic	Description
31:0	Reserved	Writes to this register have no effect. A read returns all 0s.

External Hardware

This reference design interfaces with the AleaREP Lancelot VGA card. The Lancelot card conforms to Altera's standard Nios daughter card header size, which allows it to be used with several of Altera's development boards. This VGA card contains a Texas Instruments THS8083A triple-channel 8-bit DAC. Also included on the card are two PS2 interfaces. which are useful for interfacing a mouse and keyboard to the Nios processor.

For more information on this video adapter card, visit www.fpga.nl.

The reference design can also be easily modified to work with other Video DAC chips. The output of the VGA driver conforms to 640 × 480 VGA standard timings and interfaces well with any other DAC that supports this timing.

Test Software

The reference design includes test software. The test software exercises the interface between the processor and the slave interface, and the interface between the DMA controller and the embedded stripe. The test software includes two primary functions. The first function is a main routine that sets up the VGA driver by configuring the size of the images to be driven and the base address in memory of the images. The second function is the interrupt service routine—when the VGA driver has been configured and enabled by the main routine it asserts its `irq` signal every time the last pixel in the current frame has been sent. This assertion triggers the interrupt service routine, which tells the VGA driver where the next image is located. This test software includes an animation sequence of 29 images, which are stored in SDRAM. The interrupt service routine increments the VGA driver's buffer address every time it receives an interrupt. Writing a new frame address to the slave interface causes the interrupt to be cleared and hands the processor back to the main routine until the next interrupt occurs.

Installation

To install the video DMA reference design unzip the **an287.zip** file. Figure 13 shows the reference design directory structure.

Figure 13. Directory Structure

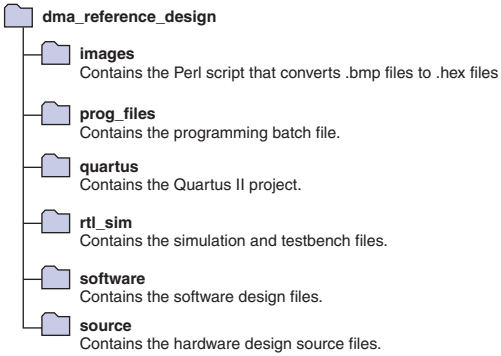


Table 12 describes some of the reference design files.

Table 12. Reference Design Files (Part 1 of 2)

Filename	Description
\source\video_system.vhd	Reference design top-level design file.
\source\video_dma.vhd	Video DMA driver top-level.
\source\video_dma_controller.vhd	DMA controller portion of the design.
\source\vga_driver.vhd	VGA driver.
\source\slave_interface.vhd	AHB Slave containing a register bank .
\source\default_slave.vhd	AHB default slave.
\source\slave_decoder.vhd	AHB Slave address decoder.
\source\response_and_data_mux.vhd	AHB Slave data and response multiplexer
\source\image_package.vhd	Package defining timing parameters for VGA.
\source\video_components.vhd	Package containing component declarations.
\software\main.c	The main C file used to test the design.
\software\irq.c	The interrupt service routine.
\software\uartcomm.c	UART driver.
\software\irq.h,uartcomm.h,uart00.h, int_ctrl00.h	Head files for software design.
\software\sim_test.s	A simple test program used to generate models for simulation in an HDL simulator.

Table 12. Reference Design Files (Part 2 of 2)

Filename	Description
\rtl_sim\testbench.vhd	VHDL testbench for the design.
\rtl_sim\mt48lc16m16a2.vhd	Simulation model of EPXA1 development board SDRAM.
\rtl_sim\simulate_video_system.do	Simulation script for the ModelSim simulator.
\rtl_sim\memory.dpram0, memory.dpram1, memory.regs, memory.sram0, memory.sram1	Simulator initialization files generated from the sim_test.s assembly file.
\images\bmp2perl.pl	Perl script used to convert .bmp files to .hex files.
\prog_files\prog_hw.bat	Batch file used to link software and download code to the EPXA1 development board.

The reference design requires the following software:

- Quartus® II version 2.0 or higher
- ARM Developer Suite for Altera (Altera ADS-Lite) software, version 1.1

Using the Reference Design

Using the reference design involves the following steps:

1. Compile the Hardware.
2. Compile the Software.
3. Link the Software Image with Still Images & Program the Board.
4. Simulate the Reference Design

Compile the Hardware

To compile the hardware, perform the following steps:

1. Open the **video_system.quartus** project in the Quartus II software.
2. To compile the hardware portion of the project, choose **Start Compilation** (Processing menu).

Compiling the hardware generates a slave binary image (SBI) file, which contains the hardware image that you want to configure into the FPGA portion of the Excalibur device.

Compile the Software

After the hardware has been compiled, you can also compile the software portion of the design using the Quartus II software. To compile the software, choose **Start Software Build** (Processing menu).

Compiling the software generates a software image that you can download to SDRAM.

Link the Software Image with Still Images & Program the Board

The reference design's software design cycles through a sequence of images stored in memory. The software converts the original Windows bitmap image files to raw data and stores them in **animate_image_data.hex**. This data file has an address offset of `0x00100000`.

The software image and the animation data must be combined to generate an object file by using the **makeprogfile** utility. This object file is linked in with the Altera bootloader by using the **armlink** utility, which generates a executable and linkable file (**.elf**). Finally the **.elf** file is converted to a **.hex** file by using the fromelf utility. The **.hex** file generated by the **fromelf** utility can be downloaded to the EPXA1 development board. Each of the command processes listed above are performed in the **prog_hw.bat** file, which is included with the design.

To program the board and link the software, perform the following steps:

Do not attach the video DAC daughter card to the board until you have successfully programmed and then powered down the board.

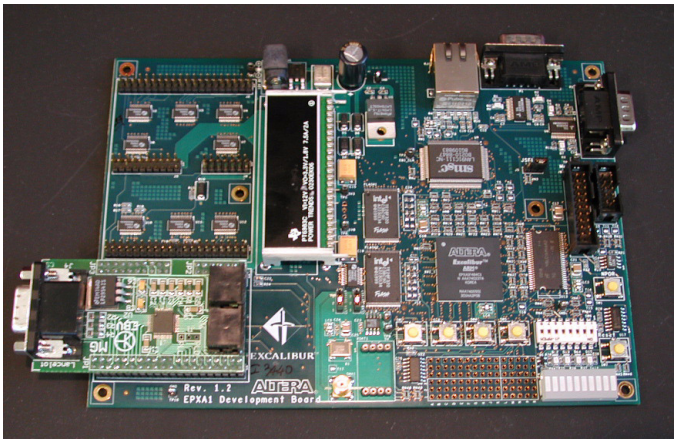
1. Open a DOS prompt and change directories to the \<*reference design installation directory*>**\prog_files** directory.

2. Ensure that the EPXA1 development board is powered on. Type `prog_hw` at the DOS prompt. Running this batch file first links the software image to the boot loader and then programs the XA1 board using the **exc_flash_programmer** utility.

3. Power off the board.

4. Attach the AleaREP video daughter card to the EPXA1 development board so that the daughter card occupies the JP15, JP11 and JP10 connectors. Arrange the daughter card so that the VGA connector points away from the board (see Figure 14). After you have attached the daughter card connect a monitor to the video connector on the daughter card. Power on the board and after the monitor warms up an animation sequence runs on the monitor.

Figure 14. Daughter Card attached to EPXA1 Development Board



Simulate the Reference Design

The simulation environment included with the reference design simulates the portions of the design that interface to the embedded stripe. Therefore, the testbench exercises the slave interface and the DMA controller. Simulating the entire design requires many more simulation cycles than simulating just the portions of the design that interface to the stripe. However, the other components in the design have been pre-verified as being functionally correct.

The reference design includes a simulation script, which is used to setup the simulation environment. Run the script by typing the following command from a ModelSim SE command prompt:

```
simulate_video_system.do
```

The script performs the following actions:

- Compiles the full-stripe model of the Excalibur device
- Compiles the necessary design components and loads and runs the design in the simulator
- Launches a waveform and runs a simulation, which is set to run long enough to transfer two lines of video across the PLD-to-stripe bridge

In this simulation, the embedded processor uses a simple assembly file to set up the driver. The file writes the size of the image and the base address of the image to the slave interface. An assembly file is used instead of the main test software, as the assembly file does not have the overheads compared to the C code—when running C code, the processor must setup up the stack and other C runtime features. It can be much faster to simulate assembly code, particularly if the intent of the simulation is test peripheral interfaces. To change the software loaded into the processor, compile the **sim_test.s** assembly code. Then convert the **.hex** file generated by the ADS or GNU compiler and generate **memory.dpram0**, **.dpram1**, **.regs**, **.sram0**, and **.sram1** model initialization files by using the **makeprogfile** utility.

For more information on generating model initialization files and using the full-stripe simulation model, refer to *AN240: Simulating Excalibur Systems*

Convert .bmp Files to .hex Files for storage in SDRAM

The reference design includes a Perl script that converts 24-bit Windows **.bmp** files into 16-bit **.hex** files. The **bmp2hex.pl** script generates a **.hex** file from a single or multiple images. To use the script you must have Perl installed on your system. To run the script, type the following command:

```
perl bmp2hex.pl image_name0.bmp image_name1.bmp
```

The script defaults offsetting the image data in memory by 0x00100000. Other address offsets can be used by modifying the hex_address_line sub-routine in the script.

For more details, see the comments in the script.

Summary

Excalibur devices offer the flexibility to allow many different types of DMA controllers to be implemented in system-on a-programmable-chip systems. You can use DMA controllers to remove system performance bottlenecks, which allows the embedded processor to concentrate on performing other tasks, while the DMA performs bulk data transfers. One example application benefited by a DMA controller is graphics displays. Designs requiring a graphics display are readily implemented in the Excalibur devices and can be designed in systems easily using the included reference design.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com
Applications Hotline:
(800) 800-EPLD
Literature Services:
lit_req@altera.com