The Express Configuration of SpartanXL FPGAs

XAPP 122 November 13, 1998 (Version 1.0)

Application Note by Kim Goldblatt

Summary

Express Mode uses an eight-bit-wide bus path for fast configuration of Xilinx FPGAs. This application note provides information on how to perform Express configuration specifically for the SpartanXL family. The Express mode signals and their associated timing are defined. The steps of Express configuration are described in detail, followed by detailed instructions that show how to implement the configuration circuit.

Xilinx Family

SpartanXL

Introduction

Express mode is a fast means for configuring the 3.3-Volt SpartanXL family. This mode is able to configure an FPGA quickly, since it uses an eight-bit bus to load one byte of data for every cycle of the configuration clock (CCLK), which is driven from an external source. In Express mode, the FPGA acts as a “slave”. The “master” to which it responds will typically be a processor, CPLD or some kind of intelligent interface.

Express mode is one of four different ways to configure the Volt SpartanXL family. The other methods are Slave Serial mode, Master Serial mode and configuration via the JTAG port.

Note that the XC4000XLA family also supports Express mode. However, the 5-Volt Spartan family does not.

When to Use Express Mode

Express mode is the fastest means for configuring the SpartanXL family and, therefore, should be used whenever the FPGA must go from power-up to user operation in the shortest possible time.

Express mode will configure a SpartanXL device eight times faster than the slave serial, master serial and JTAG methods, since Express mode transfers one byte of data per cycle compared to one bit per cycle for the other three modes.

The time it takes for the Express configuration of the largest SpartanXL device available, the XCS40XL, is 387,848 bits divided by 8 bits per cycle or 48,481 cycles, significantly less than the 330,696 cycles required for serial configuration. At the maximum allowable clock frequency of 8 MHz, Express configuration takes about 6.1 ms compared to the 41.3 ms required for serial configuration.

Express mode requires an 8-bit bus to carry the configuration data. If insufficient bandwidth is available (i.e. the bus needs to be free for other tasks at the time of power-up or board initialization), then one of the other three configuration methods, all of which require only a single data bit, will be preferable.

Express Mode Signals

An Express mode implementation can involve as many as 17 lines on the SpartanXL device including: M1, D0 through D7, CCLK, PROGRAM, INIT, DONE, CS1, DOUT, HDC and LDC. (The last three are not always used). The principal functions of the 17 lines are described in Table 1. Refer to the Spartan Series Data Sheet for more detailed information.

Steps in the Configuration Process

The Express mode consists of four steps:

1. Clearing Configuration Memory
2. Initialization
3. Configuration
4. Start-Up

Let’s have a look at each of these steps so that we may understand how the 17 configuration signals work together to program a SpartanXL device. Refer to the Spartan Series Data Sheet for more details.

Clearing Configuration Memory

On power-up, once V_{CC} reaches the Power-On-Reset threshold, the device automatically begins clearing the configuration memory. It is also possible to begin the clearing operation by applying a Low-level pulse to the PROGRAM input.

This line makes *reconfiguration* possible at any time during device operation. It is particularly useful when the controller needs to initiate configuration at a specific point in the power-up sequence.

As long as PROGRAM is Low, the device continues to cycle through the clearing step. After each pass through

Table 1: Signals for Express Configuration

Signal	Type	Direction	Description
M1	Mode selection	Input	Set Low for Express mode
D0 - D7	Data	Input	Write configuration data into the device
DOUT	Data	Output	Status output connected to the CS1 input of the next FPGA in a daisy-chain, enables loading of configuration data into the next FPGA
CCLK	Clock	Input	Synchronizes configuration data on the rising edge
$\overline{\text{PROGRAM}}$	Control	Input	Begin clearing the configuration memory
$\overline{\text{INIT}}$	Control	Open-drain output	A transition from Low to High indicates that the configuration memory is clear and ready to receive the bitstream
DONE	Status	Open-drain output	A High indicates that the configuration process is complete
HDC	Status	Output	High throughout configuration, until the I/Os go active
$\overline{\text{LDC}}$	Status	Output	Low throughout configuration, until the I/Os go active
CS1	Control	Input	A High enables loading of configuration data for an FPGA in a daisy-chain

Note: M0 is a Don't Care for Express mode.

the configuration memory, $\overline{\text{PROGRAM}}$ is sampled. If $\overline{\text{PROGRAM}}$ is High, then one last clearing pass takes place, which concludes with a Low-to-High transition on $\overline{\text{INIT}}$.

Do not hold $\overline{\text{PROGRAM}}$ Low for more than 500µs. Therefore, $\overline{\text{PROGRAM}}$ should not be used to delay the configuration process for periods of this magnitude. Hold $\overline{\text{INIT}}$ Low instead.

Initialization

Since $\overline{\text{INIT}}$ is an open-drain output, it requires a pull-up resistor to achieve a High level. Now that $\overline{\text{INIT}}$ has gone High, the internal memory is completely clear. At this point, the device identifies the selected configuration mode by sampling the level on the mode pins, after which it activates the appropriate configuration logic. The device is ready to begin the configuration step. Note that holding $\overline{\text{INIT}}$ Low can be used to delay the entry to the configuration step.

To select Express mode, the mode select input, M1, is tied Low. The other mode select input, M0 is a "don't care"; it makes no difference whether the pin is High or Low.

Configuration

After $\overline{\text{INIT}}$ goes High, it is necessary for the master (i.e., controller) to wait for a period of T_{IC} = 5 µs before driving the CCLK input on the device. CCLK is driven from an external source. The clock oscillator internal to the device is not used to transfer data; it is only used during Initialization.

The configuration data, in the form of bytes, enter the device via D0-D7. One byte is clocked on the rising edge of CCLK each cycle as shown in Figure 1. The data needs to be set up for a period T_{DC} before the rising edge and held for the period T_{CD} after the rising edge. Numbers for the set up and hold times can be found in the Spartan Series Data Sheet. Bytes are loaded until the configuration memory is full, at which point DONE goes High, marking the beginning of the Start-Up step.

Start-Up

The Start-Up step provides a smooth transition from configuration to user operation. Three major events occur during Start-Up: The DONE output goes High, the I/Os go active and the GSR (Global Set/Reset) net is released. Start-Up takes place over a period of four cycles labeled C1, C2, C3 and C4. Options in *BitGen*, the bitstream generation program in Xilinx development software, determine which event takes place in which cycle. The menu for these options can be located as follows:

1. Open the Design Manager.
2. Select Implement from under the **Design** menu.
3. Choose the **Options** button.
4. Click on **Edit Template** for Configuration.
5. Select the **Startup** tab. A menu will appear that permits the three events to be assigned to different cycles.

As an alternative, *BitGen* options can also be selected using **Template Manager**, which is found under the **Utilities** menu of the Design Manager.

The Start-Up sequence discussed in this application note uses CCLK for the purposes of synchronization. This option is known as "CCLK_SYNC". The customary default *BitGen* settings are the most practical ones, since DONE goes High in C1, disconnecting the data source to avoid any contention, after which the I/Os go active and the GSR is released in C2, ensuring stable internal conditions. The CCLK is used to measure out the four start-up cycles. This application note only considers the default option.

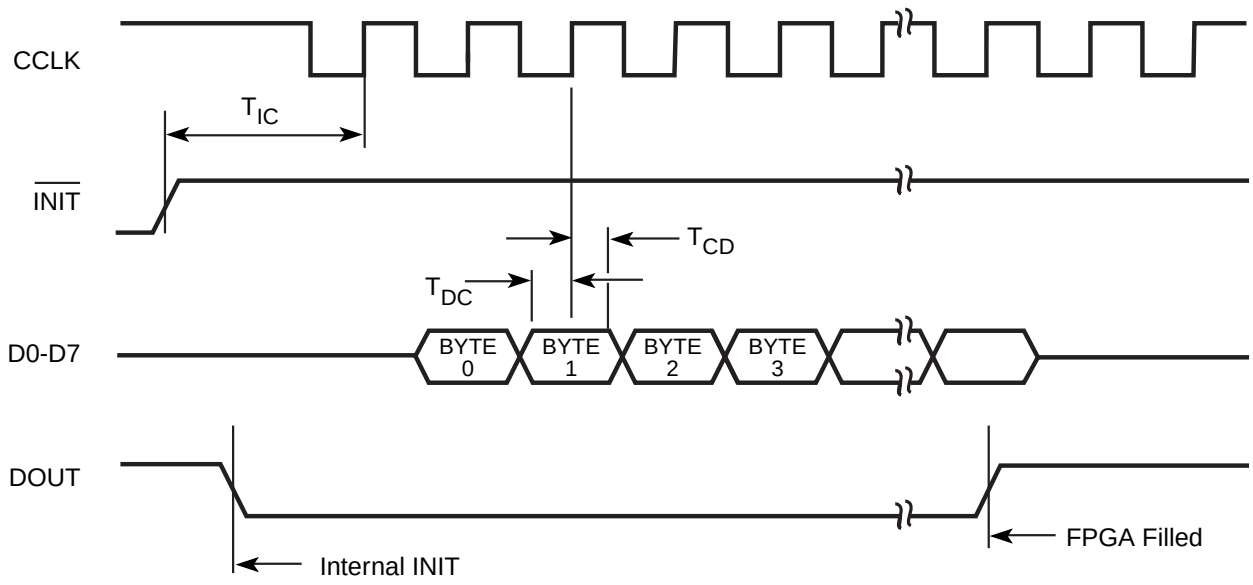

Figure 1: Loading the Bitstream in Express Mode

Note: CS1 is held High, enabling the device to receive configuration data.

In Express mode, full configuration memory is the one condition that determines when the Configuration step is finished. DONE goes High as a result of filling the configuration memory completely. This marks the beginning of the Start-Up step. DONE's failure to go High generally indicates a problem with configuration such as the incomplete loading of configuration data.

While a transition from Low to High on DONE indicates the completion of the configuration *step*, the configuration process, as a whole, ends with the last cycle of the Start-Up step, C4. It is important to provide CCLK rising edges for all four start-up cycles. This amounts to clocking the entire configuration file, from the first byte of the header to the last start-up byte.

Like DONE, the HDC and $\overline{\text{LDC}}$ outputs provide status on the device's progress to user operation. HDC is High during configuration and takes on whatever I/O function is assigned to it at the time when all I/Os go active, in the Start-up step. Similarly, $\overline{\text{LDC}}$ is Low during configuration and takes on its respective I/O function when the I/Os go active as well.

Configuring Multiple SpartanXL FPGAs

It is possible to configure any number of SpartanXL devices with a composite configuration data file. The devices are connected to form a "daisy-chain" by connecting the DOUT output of one device to the CS1 input of the next. M1 must be tied Low for all devices so that Express mode is used throughout the chain. The CS1 input of the first (left-most) device in the chain is tied to V_{CC}. The DOUT output of the last (right-most) device in the chain is left open. See Figure 2 for an example of how the devices are connected together. D0-D7 is connected in parallel to all the devices in the daisy chain.

The DONE output of all devices in the chain are tied together, as are the $\overline{\text{INIT}}$ outputs. Both these outputs are open-drain; therefore, they need to be pulled to Vcc for a High logic level. A 470Ω pull-up resistor is recommended.

A device can only accept configuration data if two conditions are met: CS1 is High and the configuration memory is not full. The High level on the CS1 input ensures that the first device is able to accept data. The $\overline{\text{INIT}}$ signal going High causes the DOUT of the first device to go LOW, disabling configuration of the rest of devices in the chain. When the configuration memory of the first device is full, its DOUT goes High, enabling the next device in the chain to receive configuration data from the parallel bus.

The line that connects the DONE outputs of all devices will not go High until all the configuration data has been loaded. The assertion of DONE marks the beginning of the Start-Up step.

For a daisy chain, the configuration data for the individual devices need to be combined into a single file. For details, see "Combining Files for a Daisy Chain" on page 8.

If the same configuration data file is to be loaded into more than one FPGA, then the devices can be connected with their configuration signals in parallel. See the Spartan Series Data Sheet for more information on daisy chain and parallel configuration.

The Controller Interface

One common way to implement Express configuration uses a controller to send the configuration data to the SpartanXL device(s) over the data bus. Aside from the FPGA(s), this application typically uses three resources:

1. ROM to store the configuration data file.
2. A controller for coordinating configuration.
3. A free register (e.g., in a CPLD or an I/O port) can be used as a synchronous interface between the controller and the SpartanXL device.

See Figure 2 for a schematic diagram of the controller interface.

Storing the Bitstream

A form of nonvolatile memory, such as ROM, is used to hold the configuration data. Generally, the data will be embedded in the processor's firmware. See "Embedding the Bitstream in Firmware" on page 8 for information on how to prepare configuration data for inclusion in C or assembly code.

As an alternative to the embedded approach, a free portion of ROM can be set aside to store the bitstream in a table that is independent from the firmware. During board initialization, the firmware can then instruct the processor to access the table.

Controlling Configuration

The controller supervises serial configuration by monitoring status signals, issuing control signals, manipulating the bitstream, and providing for synchronization to a clock.

If insufficient continuous processing time is available for configuration, then the task of writing the bitstream may be interrupted so the controller can attend to other tasks, only to be resumed at a later point in time. In this case, the task of writing the bitstream exists as firmware subroutine, to which an interrupt priority can be assigned.

In brief, the $\overline{\text{INIT}}$ line can be used to drive the interrupt line on the controller. A suitably low priority level can be assigned to this interrupt to ensure that the controller spends sufficient time servicing its primary tasks. As previously described, the processor initiates Express configuration by pulling $\overline{\text{PROGRAM}}$ Low. Once all the devices are clear, $\overline{\text{INIT}}$ goes High, requesting an interrupt of the controller. When the controller has no requests of higher priority than that of the SpartanXL device, it begins accessing configuration data from memory and writing them to the DIN input, bit-by-bit. While the controller will break away from configuration to attend to any higher priority requests, as soon as these are complete, it will continue with configuration until the DONE signal, monitored at the interface register, goes High.

When using interrupts, it is important to use a unique address for the SpartanXL device (or daisy chain). This avoids potential address conflicts when switching tasks. See the next section for how this is accomplished.

The Interface Register

A register is used to establish a synchronous interface between the controller and the SpartanXL device(s). The interface register is composed of two parts: the *output register* and the *input register,* which store the bit values of the configuration signals. In order to support the set of signals commonly used for the Express mode ($\overline{\text{PROGRAM}}$, D0-D7 and CCLK, $\overline{\text{INIT}}$ and DONE), the output register must be ten-bits wide for write operations and the input register must be two-bits wide for read operation. More bits can be added for other control signals. For example, when using the readback feature, add two bits for the READ_DATA and READ_TRIGGER signals.

The interface register should have a unique address which ensures that the configuration data on the processor's bus goes only to the register and nowhere else. It also ensures that data on the bus intended for other purposes cannot be written to the interface register.

A Practical Example

Let us look at a specific example of a processor configuring SpartanXL FPGAs in Express mode. Figure 2 shows a block diagram of a board with a processor. The µP Data Bus and the µP Address Bus permit access to the ROM, RAM and a CPLD with an interface register. The program for the processor resides in the ROM. On reset or power-up, the processor begins reading its instructions from here.

In addition to whatever primary tasks it may have to perform, the processor serves as the *master* for serial configuration. In this role, the processor accesses the configuration data, formats them, writes them to the interface register and, otherwise, coordinates the act of configuration. The three FPGAs serve as *slaves*, so the M1 pin on each is tied Low to select the Express configuration mode. Note that once M1 is Low, M0, the other mode input becomes a "don't care". M0 can be tied either High, Low or left open.

The configuration data are stored in the ROM. They may be embedded in the processor's program (expressed in assembly or C language) or they may exist as an independent table.

The CPLD contains the interface register that holds the bit values of the configuration signals. This example employs the minimum required number of signals: $\overline{\text{PROGRAM}}$, CCLK, D0-D7, $\overline{\text{INIT}}$ and DONE. The interface register consists of two parts, one called *Out-Reg* and the other called *In-Reg*. The processor writes bit values for $\overline{\text{PROGRAM}}$, D0-D7 and CCLK into Out-Reg, which, in turn, applies

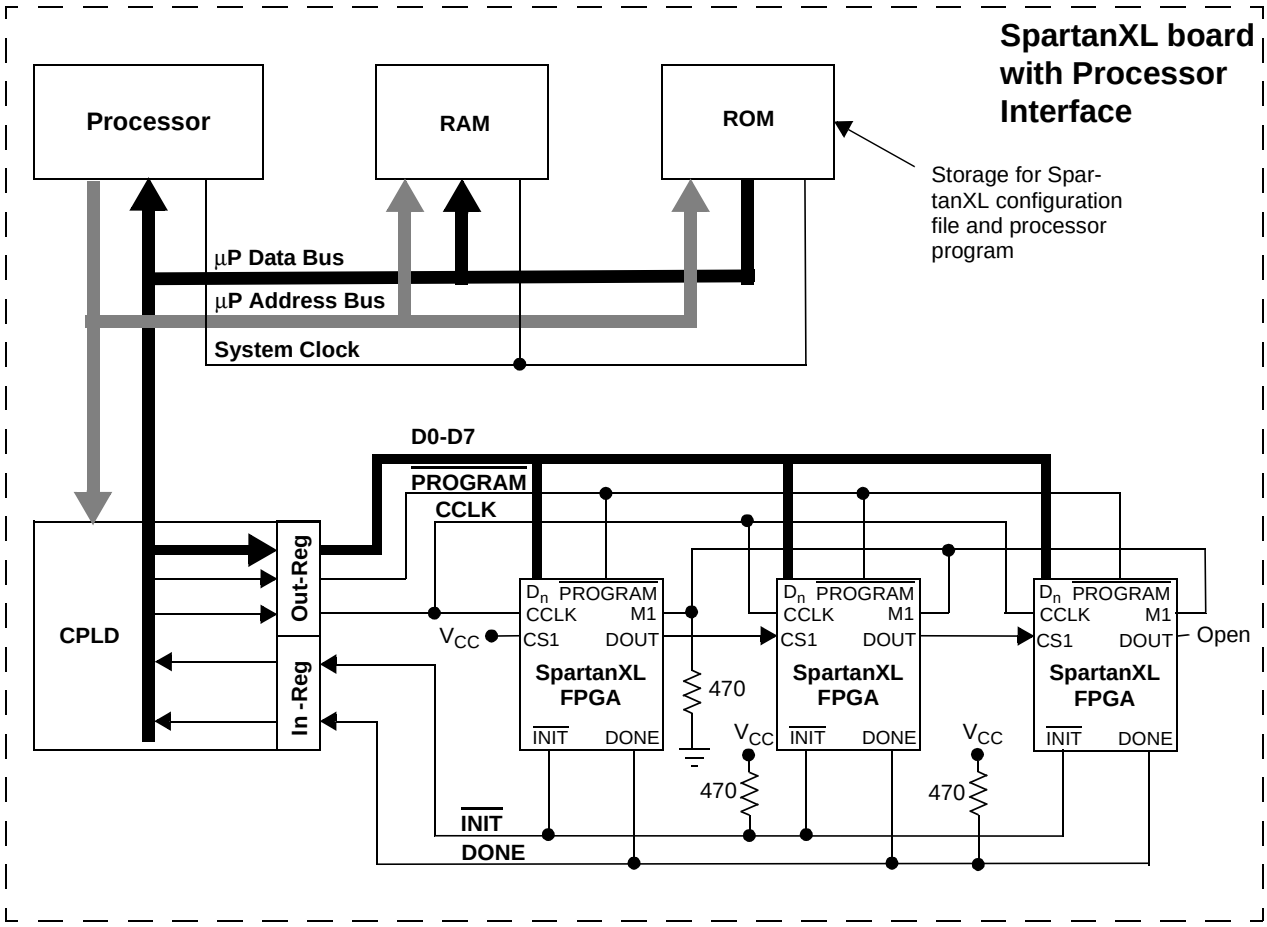

Figure 2: Express Configuration of SpartanXL FPGAs Using a Controller
Note: The M0 input on the SpartanXL devices is a “don’t care” and can be left open.

those values to the corresponding inputs of the SpartanXL device(s). Also, on a regular basis, In-Reg samples INIT and DONE from the device(s) and makes those bit values available on the Data Bus for monitoring by the processor. During Express configuration, the processor takes turns writing *control words* to Out-Reg one instruction cycle and reading *status bits* from In-Reg the next. The ten bit values contained in the control word provide the logic levels that drive Out-Reg’s PROGRAM, D0-D7 and CCLK signals. The two values in the status bits communicate to the processor the levels of In-Reg’s INIT and DONE signals.

A sample sequence of control words and status bits is shown in Table 2. Each row in the table shows the bit values in the interface register at a given point in time. This is just one of a number of different possible sequences. The full sequence, from start to finish, passes through the four steps of configuration: Memory Clear, Initialization, Configuration and Start-up. The processor initiates memory clearing by issuing a control word with PROGRAM set Low. If INIT is not already Low, it will go Low at this time. During this step, CCLK can be High or Low, so long as there’s no rising transition. Dummy bytes occupy the bit positions for D0 through D7. The processor monitors In-Reg until it detects INIT at a High level. At this point, the processor needs to wait a period from 55 µs to 275 µs, during which initialization takes place. With the beginning of the configuration step, the processor begins to write control words containing “real” data bytes while, at the same time, continues to monitor In-Reg.

Finally, the Start-up step readies the SpartanXL device for user operation over a series of four CCLK cycles according to the customary default settings in *Bitgen* (see “Start-Up” on page 2): In C1, DONE goes High. In C2, the I/Os become active. In C3, the GSR net is released. In C4, user operation begins. It is important that a rising transition on CCLK be provided for C1, C2, C3 and C4. The data clocked during those cycles, B_{n-3} through B_n, are dummy bytes.

Table 2: State Sequence for the Interface Register

Configuration Step	Contents of Interface Register				
	Control Word in Out-Reg			Status Nibble in In-Reg	
	PROGRAM	CCLK	D0-D7	INIT	DONE
Memory Clear	1	NRT[1]	X[2]	0[3]	0[4]
	0	NRT	X	0[3]	0[4]
	INIT goes Low (if not already Low).				
	1	NRT	X	0	0
	Wait for T_{IC} = 5 µs after INIT goes High				
Initialization	1	NRT	X	1	0
Configuration	1	0	B_0[5]	1	0
	1	1	B_0	1	0
	1	0	B_1	1	0
	1	1	B_1	1	0
	1	0	B_2	1	0
	1	1	B_2	1	0
	Continue writing bytes.				
	1	0	B_n	1	0
	1	1	B_n	1	0
	When the configuration memory is full, then the Start-Up step begins.				
Start-Up[6]	1	0	X	1	0
	1	1 (C1)	X	1	0
	DONE goes High.				
	1	0	X	1	1
	1	1 (C2)	X	1	1
	I/Os become active and GSR is released.				
	1	0	X	1	1
	1	1 (C3)	X	1	1
	1	0	X	1	1
	1	1 (C4)	X	1	1
	Begin User Operation				

Notes:
1. NRT means No Rising Transition.
2. X is a "don't care" byte.
3. The level shown is for configuration after power up. For configuration in mid-operation, prior to driving PROGRAM Low, DONE will be High.
4. The level shown is for configuration after power up. For configuration during operation, prior to driving PROGRAM Low, INIT may be an active I/O, in which case, it will be at a High or a Low.
5. B_i represents the sequence of configuration data bytes i = 1 through n, where B_1 is the first byte of the header and B_n is the last byte for extending write cycles.
6. This example shows the Start-up events ordered according to the default settings in Bitgen when CCLK is used to synchronize the Start-Up step.

Before the processor can send the configuration data file to the SpartanXL devices, it is necessary to format them into control words. The processor can accomplish this real-time by reading a byte of configuration data from ROM and positioning it within the control word so that it lines up with D0-D7 bits of the interface register. The processor also needs to provide the appropriate logic levels for the PROGRAM and CCLK bit positions in the control word.

The bit values for all ten signals need to be chosen in compliance with the protocol summarized in "Steps in the Configuration Process" on page 1 as well as the timing requirements described in the Spartan Series Data Sheet. For example, note in Table 2 that during the configuration step, each data byte is repeated for two consecutive writes to Out-Reg.is Low for the first occurrence, High for the second. This ensures that the setup times for D0-D7 with respect to CCLK are met. Note that the hold time for D0-D7 with respect to CCLK is zero for the SpartanXL family. Thus, it is unnecessary to continue holding the byte for a third control word.

It is important that the order of the control words, as written to the Out-Reg, preserve the byte order of the original configuration file. The first byte of the header (just after the title declaration) needs to be the first byte received by the FPGA. The first bit of each byte must line up with the D0 input on the SpartanXL device. Similarly, the eighth bit of each byte must line up with D7.

Figure 2 shows three SpartanXL devices connected in a daisy chain, though any number of Xilinx FPGAs can easily be accommodated in such a loop. See "Configuring Multiple SpartanXL FPGAs" on page 3 for an explanation of how the daisy chain signals work.

The Configuration Data File

Express mode requires a special bitstream file that is not compatible with any of the other configuration modes. This file is created by specifying a *BitGen* option in the Xilinx development software (version 1.5.25 or later). Note that presently, the software only supports this option using command line entry. The following command produces a configuration file for Express mode:

bitgen -g ExpressMode:Enable -g CRC:Disable -b filename

The "bitgen" command runs a program that produces configuration files. A -g option followed by "ExpressMode:Enable" instructs *BitGen* to produce an Express configuration file.

Since Express mode does not support error detection using CRC, it is necessary to disable this feature with the text: "-g CRC:Disable".

The data contained in the Express configuration file can be represented in a number of different forms, including the rawbits file (.RBT), the hex file (.HEX), the bit (.BIT) file. Table 3 summarizes the distinguishing characteristics of

Table 3: Configuration Data Files

File Format	File Extension	Title Declaration	Description
Rawbits	.RBT	Yes	Bitstream is coded in ASCII, one byte for each configuration data bit
Hex	.HEX	No	Each group of four consecutive configuration data bits is represented as one Hex digit (i.e., 0 through F) which, in turn, is coded as one ASCII byte
Binary	.BIT	Yes	Bitstream is coded in binary, one configuration bit after the next

these files. The default format for the command line shown is the bit file. Other options can be added to the command line to produce additional files in other formats. For example, the -b option specifies a rawbits file. The “filename” is the name of the .ncd file (i.e. the file that describes the mapped, placed and routed design).

The Anatomy of a Configuration File

A distinct benefit of the rawbits format is that the binary data can be easily viewed using a common text editor. Figure 3 shows the internal organization of an Express configuration file in the rawbits format. The same internal organization applies to Bit and Hex files as well, except that the latter does not have a title declaration. At the top of the file is a title declaration, which provides information about the configuration data such as:

- Configuration data file format
- Version of Xilinx development system in use
- The name of the design
- The target device
- The date the file was created
- The number of bits of actual configuration data

The title declaration is never loaded into the FPGA, only the header and the data frames that follow enter the device during configuration.

Following the title declaration, the actual bitstream begins with a header, which consists the following parts:

- Two dummy bytes, all ones
- A preamble code of 11110010 (shown in bold)
- A dummy length count consisting of 24 bits
- A field check code of 11010010 (shown in bold)

Following the header is the first data frame, which, like all data frames, begins with the start field 11111110 (shown in bold). Each data frame ends with the eight-bit constant field check code (11010010, shown in bold), followed by five bytes of extended write cycles. (The second of these is a repeat of the field check code, the other four consist of all ones.) Unlike the configuration file for the other modes, there is no post-amble code to terminate the configuration file for Express mode. The file ends with six or more dummy bytes, all ones. The first and second of these six are fill bytes. The third, fourth, fifth and sixth bytes correspond to the Start-Up cycles C1, C2, C3 and C4 (See Table 2). If D0-D7 are outputs during user operation, be sure to avoid any possible contention by disabling the configuration source before the I/Os go active (i.e., before C2).

Xilinx ASCII Bitstream
Created by Bitstream M1.5.25
Design name: s40xl.ncd
Architecture:spartanxl
Part: s40xlPQ208
Date: Wed Nov 11 11:37:02 1998
Bits: 387848
1111111111111111**11110010**00000010111101
0110000000**11010010111111110**111001111
1111111011111111111111111100110011111111
1110110011011111111110011101111110111 0
1111111111101101111111111111001100111
1111111011001101111111110011101111110
1110111111111110110111011111111100110
0111111111101100110111111111001110111
1110111011111111111111011101111111110
0110011111111110110011011111**11010010**
1111111111010010111111111111111111111
11

.
.
.

111111101110011111111111011111111111
1111100110111111111110110011011111111
1001110111111011101111111110011011111
1111111110011001111111111011001101111
1111100111011111101110111111111101101
1101111111111001100111111111011001101
1111111110011101111110111011111111111
1101110111111111100110011111111110110 0
1101111111**11010010**1111111110100101111
1111111111111111111111111111111111111
1111111111111111111111111111111111111
1111111111111111

Figure 3: SpartanXL Express Configuration File

The Rawbits File

Before the configuration data can be written to the device, it is necessary to first strip off the title declaration, then, convert the header and data frames from ASCII to binary. Then the binary version is segmented into bytes, starting with the first eight bits of the header and ending with the last eight bits for extending write cycles. An on-board controller can accomplish this processing.

As mentioned earlier, the rawbits file can be displayed using a text editor, though this advantage is offset by high storage requirements. A rawbits file takes up eight times the space of its binary version

The Hex File

The hex file is prepared for configuration in similar fashion. It has an advantage over the rawbits file in that, with each ASCII character representing four bits of binary data (one hex digit), it takes up a quarter of the storage space required for the rawbits version. On the other hand, the hex file requires more processing to convert it into the binary that is used to configure the device. Before loading the file into a SpartanXL device, it is necessary to first convert ASCII to hex (i.e., each ASCII byte becomes a single hex digit), and then from hex to binary (i.e., each hex digit becomes a nibble of binary). The Hex file does not have a text title declaration, so nothing needs to be stripped away from the file.

The Binary File

A binary format has two benefits over the other two file types. First, it is the most compact of all, taking up half the storage space of a hex file and one eighth the space of a rawbits file. As a result, the format is ideal for storage on board. Second, once in binary form, the header and data frames require no further translation and can be written directly to the device.

Because of the difficulty in identifying and removing the title declaration, the binary (.bit) file created by *BitGen* is not recommended for use. Instead, one should use *BitGen* to create a hex file, which, in turn, is converted to binary as described in the preceding section.

Embedding the Bitstream in Firmware

As an alternative to storing the configuration file in an dedicated segment of ROM space, it is also possible to embed it in the controller's firmware. To perform this task, use two utilities called *makesrc* and *pconfig* which are available for free downloading from the Xilinx web site, WebLINX. The files can be found as follows:

1. Visit WebLINX at *www.xilinx.com*.
2. Perform a Xilinx site search by selecting **Search**.
3. In the blank denoted by the words "**Search for:**", type the name of the utility you are looking for followed by an asterisk. (type *makesrc** or *pconfig**) The asterisk is a wild card character that will allow for any file extension the utility may have. Press enter to begin the search.
4. The browser will report the results of the search. Click on any of the hypertext links found. This takes you to a page from where the utility can be downloaded.
5. Click the file name (i.e., *makesrc.zip* for PCs and *makesrc.tar* for Unix-based computers*)* to download the file.
6. De-compress using the appropriate program (e.g., PKZip) and it is ready to use.

Since *makesrc* can only accept a file in the MCS PROM format as an input, it is necessary to convert the binary (or rawbits) file produced by *PromGen* (part of the Xilinx development software) first. The *pconfig* utility, which converts .bit, .rbt, and .hex files to MCS format, performs this task. The resulting MCS file is used as an input to *makesrc*, which produces a HEX file with formatting customized to suit the needs of different assemblers and compilers. For additional information on how to use *makesrc* and *PromGen*, consult the accompanying read-me files.

Combining Files for a Daisy Chain

The integrated bitstream used for configuring a SpartanXL daisy chain is not a simple concatenation of the configuration files for the individual devices. *PromGen* must be used to join the files. This utility only combines binary (.bit) files, which the *BitGen* utility readily supplies. *PromGen* takes the binary files for the different devices, strips off the title declarations and the headers, merges the data frames, and, finally, adds a new header at the top. The output file is a hex file (no title declaration). If a rawbits file is desired, then use the *hex2bits* utility. *Hex2bits* is available in both zip and tar versions and can be downloaded from WebLINX using the method that was recommended for *makesrc* and *pconfig* in the preceding section. Consult the accompanying read-me file for information on how to use *Hex2bits*.

Verifying Configuration

The successful loading of the bitstream into the device can be verified by reading back the configuration data in serial. This is accomplished by instantiating a readback symbol into the SpartanXL design. Refer to the Spartan Series Data Sheet and XAPP015 for directions on how to use this feature.

Bibliography

The Xilinx Spartan Series Data Sheet
(www.xilinx.com/partinfo/ds060.pdf)

The Xilinx Programmable Logic Data Book
(www.xilinx.com/partinfo/databook.htm)

XAPP015: Using the XC4000 Readback Capability
(www.xilinx.com/xapp/xapp015.pdf)

XILINX® The Programmable Logic Company℠

Headquarters

Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124
U.S.A.

Tel: 1 (800) 255-7778
or 1 (408) 559-7778
Fax: 1 (408) 559-7114

Net: hotline@xilinx.com
Web: http://www.xilinx.com

North America

Irvine, California
Tel: (949) 727-0780

Englewood, Colorado
Tel: (303) 220-7541

Sunnyvale, California
Tel: (408) 245-9850

Schaumburg, Illinois
Tel: (847) 605-1972

Nashua, New Hampshire
Tel: (603) 891-1098

Raleigh, North Carolina
Tel: (919) 846-3922

West Chester, Pennsylvania
Tel: (610) 430-3300

Dallas, Texas
Tel: (972) 960-1043

Europe

Xilinx Sarl
Jouy en Josas, France
Tel: (33) 1-34-63-01-01
Net: frhelp@xilinx.com

Xilinx GmbH
München, Germany
Tel: (49) 89-93088-0
Net: dlhelp@xilinx.com

Xilinx, Ltd.
Byfleet, United Kingdom
Tel: (44) 1-932-349403
Net: ukhelp@xilinx.com

Japan

Xilinx, K.K.
Tokyo, Japan
Tel: (81) 3-5321-7711
Net: jhotline@xilinx.com

Asia Pacific

Xilinx Asia Pacific
Hong Kong
Tel: (852) 2424-5200
Net: hongkong@xilinx.com