Speedster7t Configuration User Guide (UG094)

Speedster FPGAs

Preliminary Data

Achronix®
Data Acceleration

Copyrights, Trademarks and Disclaimers

Preliminary Data

This document contains preliminary information and is subject to change without notice. Information provided herein is based on internal engineering specifications and/or initial characterization data.

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

Website: www.achronix.com
E-mail : info@achronix.com

Table of Contents

Chapter - 1: Overview

At startup, Speedster7t FPGAs require configuration by the end user via a bitstream. This bitstream can be programmed through one of four available interfaces in the FPGA configuration unit (FCU).

The term FPGA Configuration Unit (FCU) refers to logic that controls the configuration process of the Speedster7t FPGA. This logic is responsible for the following:

- Receiving data on a variety of core interfaces (depending on the selected programming mode)
- Decoding instructions
- Sending configuration bit values to the appropriate destination (core configuration memory, the core's boundary ring configuration memory, FCU registers, etc.)
- Controls the startup and shutdown sequences that drive resets to the rest of the core
- CRC checks
- SEU mitigation
- Security
- Any core-level housekeeping that occurs on the de-assertion of reset (e.g., clearing of configuration memory)

Data from the configuration pins is brought into the FCU located in the core's boundary logic. Depending on the configuration mode, this data passes through one of four interfaces and is then provided to the control logic and state machines in the FCU. At this point, the data bus is standardized to a common interface (configuration mode independent). This data is processed and propagated to the configuration registers in the core's boundary ring, to the core's configuration memory, or to the hard IP blocks in the FPGA's I/O ring.

When all of the configuration bits are successfully loaded, the FCU transitions the Speedster7t FPGA into user mode, allowing the user to provide stimuli and enable operation.

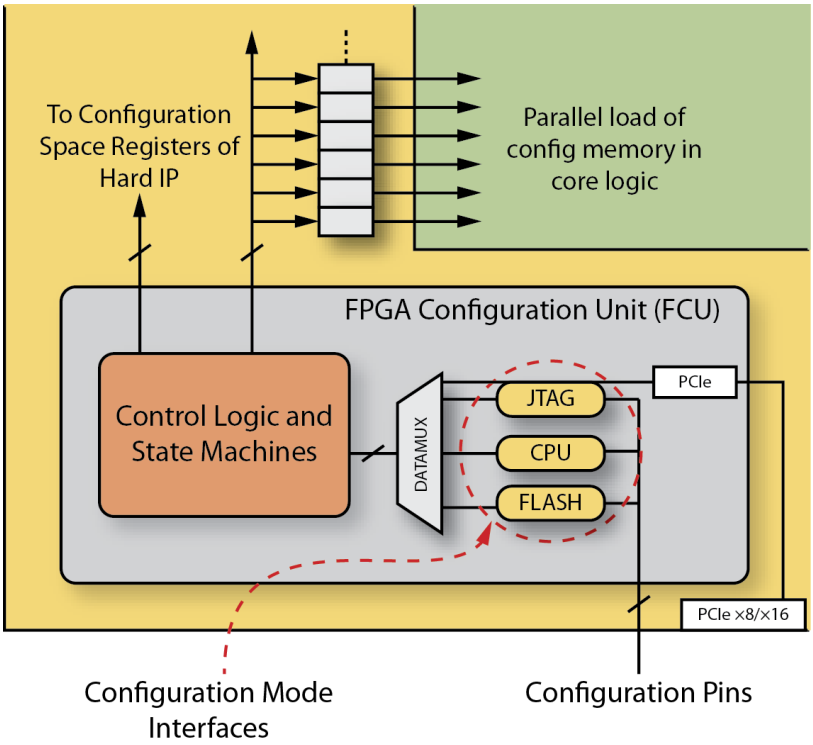

Figure 1: ***Speedster7t Configuration Block***

Chapter - 2: Interface Performance

The table below lists the various configuration interfaces supported by the Speedster7t FPGA and their corresponding maximum operating frequency.

Table 1: ***Configuration Modes and Maximum Frequencies***

Configuration Mode	Maximum Frequency
JTAG	50 MHz
CPU	250 MHz
Serial flash	62.5 MHz

All of the programming modes and interfaces are capable of running up to 250 MHz at the configuration pins. The FCU and all associated circuitry is also capable of running up to 250 MHz. Since the internal data bus in the FCU is 128 bits wide, and in most configuration modes, the data pin count is less than 128, the incoming data stream goes through a gearbox to reduce the throughput. This configuration ensures that the internal programming circuitry runs at less than 250 MHz to process the incoming data stream. In the widest data mode (CPU ×128), the gearbox is bypassed and the entire configuration interface can run at the full 250 MHz bandwidth. Depending on the mode and configuration data width, the total bandwidth varies, and the programming time changes accordingly.

Note

CPUx128 is primarily for ATE use and not a recommended mode for design configuration.

Chapter - 3: Configuration Modes for Speedster7t FPGAs

Speedster7t FPGAs support four configuration modes: Flash, JTAG, CPU and PCI Express. The selection between these modes is controlled by setting the FCU_CONFIG_MODESEL pins to the values shown in the table below. Both JTAG and PCIe modes are independent of the FCU_CONFIG_MODESEL pin setting and must be enabled by sending FCU commands that set the appropriate bits in FCU register space. The JTAG mode can be enabled by writing to the user data register of the JTAG TAP controller. The PCI Express mode is enabled by writing to the PCIe mode enable register in the FCU address space. The JTAG mode overrides all other configuration modes until disabled.

Note

PCIe mode is enabled by writing the PCIe mode enable register in the FCU address space, refer to Configuration via PCIe Express. The PCIE mode is set as highest priority over the other modes listed in the Table below.

Table 2: ***Pin Settings for Various Configuration Modes***

Configuration Mode	Data Width	FCU_CONFIG_MODESEL [3:0]	FCU_CONFIG_SYSCLK_BYPASS(3)	FCU_CONFIG_CLKSEL(3)
JTAG(1)	–	XXXX(2)	X	1
PCIe	–	XXXX	X	0
Flash single device (1D)	1 (SPI)	0001	0/1	0
	2 (Dual)	1000		
	4 (Quad)	1010		
	8 (Octa)	1100		
Flash four devices (4D)	1 (SPI)	0010		
	2 (Dual)	1001		
	4 (Quad)	1011		
	8 (Octa)	1101		
CPU	1	0011	1	0
	8	0100		
	16	0101		
	32	0110		
	128(4)	0111		

Configuration Mode	Data Width	FCU_CONFIG_MODESEL [3:0]	FCU_CONFIG_SYSCLK_BYPASS(3)	FCU_CONFIG_CLKSEL(3)

Table Notes

1. Always active. Enabled in the JTAG TAP controller.
2. If `FCU_CONFIG_MODELSEL[3:0]` pins are set such that flash or CPU configuration mode is selected, then the JTAG override should be issued after flash programming has completed or the CPU mode interface is inactive.
3. These straps select the configuration clock source.

FCU_CONFIG_SYSCLK_BYPASS	Clock Selected
0	On-chip oscillator clock
1	FCU_CPU_CLK

4. Speedster7t FPGAs have 32 dedicated data I/O pins for the CPU interface, which supports an up to ×32 interface. For ×128 mode, the upper 96 pins are shared with the DDR4 interface.
5. CPUx128 is primarily for ATE use and not a recommended mode for design configuration.

Configuration via CPU

In CPU configuration mode, an external CPU acts as the master controlling the programming operations to the Speedster7t FPGA and offers a high-speed method for loading configuration data. Depending on the setting of the FCU_CONFIG_MODESEL pins, the CPU mode can be either a 1-, 8-, 16-, 32- or 128-bit wide parallel interface, clocked using FCU_CPU_CLK, with chip select support to indicate valid data. This mode is the fastest programming mode as it provides the widest data interface and a maximum supported clock rate of 250 MHz.

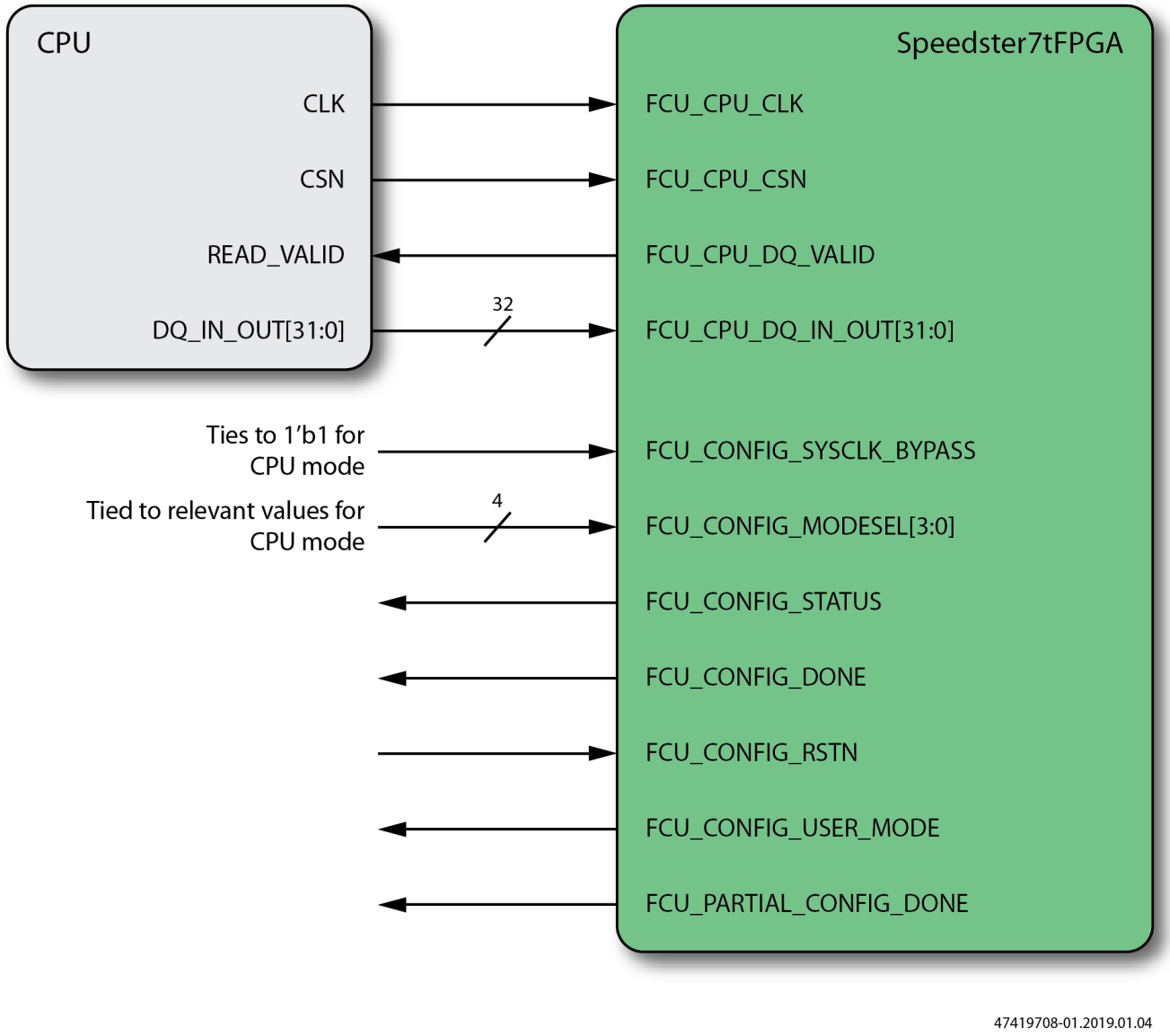

Figure 2: ***External CPU Connectivity to a Speedster7t FPGA***

> **Note**
>
> The CPU master needs only to connect to the first 1, 8, 16, 32 bits of `FCU_CPU_DQ_IN_OUT` depending on the CPU mode selected. All unused signals should be tied to ground.

As described in the Configuration Sequence and Power-up (see page 50) section, the configuration mode-specific operations occur between the release of `FCU_CONFIG_STATUS` (indicating that the configuration memory has been cleared and that the Speedster7t FPGA is ready to accept bitstream data) and the assertion of `FCU_CONFIG_DONE` (stating completion of configuration). The example waveform below for CPU×8 mode illustrates the sequence of events, clocking and control signal states needed for successful configuration in CPU mode:

1. After FCU_CPU_RSTN is de-asserted, FCU_CPU_CLK must continue to cycle to ensure that the FPGA cycles through the FCU states and the configuration memory is cleared. At that point, FCU_CONFIG_STATUS is driven high.
2. After at least 5 clock cycles of FCU_CONFIG_STATUS being driven high, FCU_CPU_CSN must be pulled low to begin writing the bitstream data into the Speedster FPGA. When the last set of data is written into the Speedster7t FPGA, FCU_CPU_CSN is pulled back high.
3. When FCU_CPU_CSN is pulled high, FCU_CPU_CLK needs to continue being clocked. When the FCU cycles through all of the configuration states, FCU_CONFIG_DONE is driven high to indicate that the Speedster7t FPGA was successfully programmed.
4. As the FCU_CPU_CLK toggles, the FCU cycles through its states to move the Speedster7t FPGA from programming mode into user mode, taking the fabric out of reset and performing operations to enable user-mode functionality for all parts of the core. The FCU_CONFIG_USER_MODE signal is asserted to indicate when the Speedster7t FPGA has successfully transitioned into user mode.

At any point during the configuration, if FCU_CPU_CSN is asserted low, then the bus FCU_CPU_DQ_IN_OUT should have valid data or NOPs, if FCU_CPU_CSN is high, the data on FCU_CPU_DQ_IN_OUT is ignored. When the bitstream is programmed, FCU_CPU_CSN can be held low while sending NOPs to the Speedster7t FPGA. This action will not affect the assertion of FCU_CONFIG_DONE or FCU_CONFIG_USER_MODE signals.

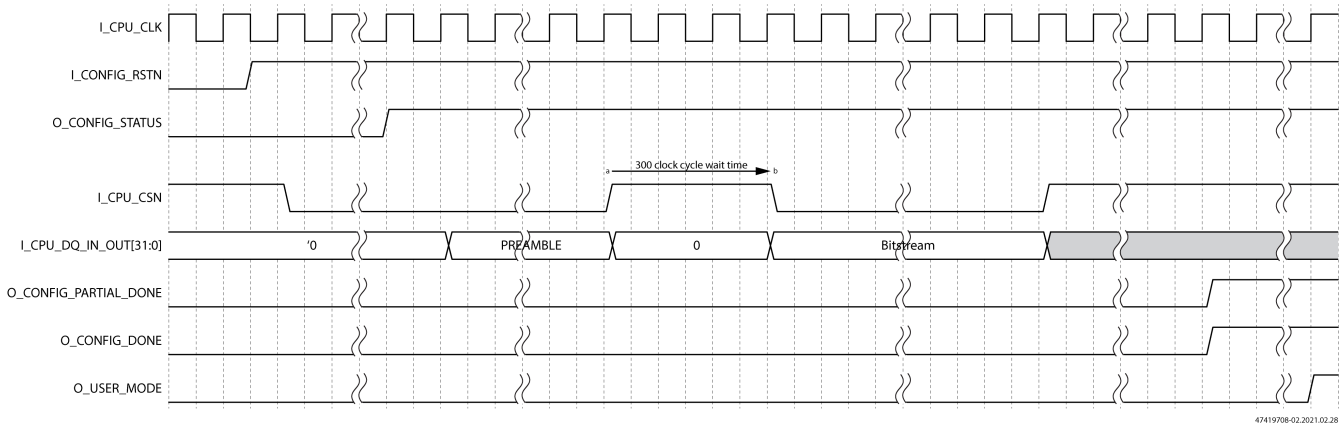

Figure 3: ***Clocking and Control Signals for Successful Configuration***

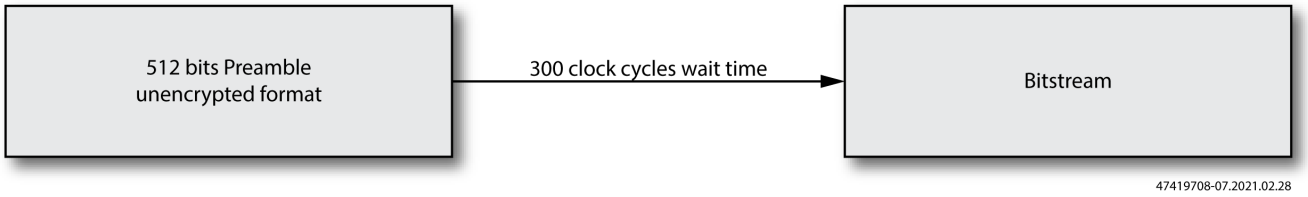

Figure 4: ***Bitstream Data Sequence for unencrypted bitstream***

> **Note**
>
> During the 100 clock cycle wait time, CPU_CSN will be pulled high.

Programming Data Ordering

In Speedster7t FPGAs, the configuration memory data bus is 128 bits wide, but the command and FCU register buses are 32 bits wide. Data transmission occurs MSB to LSB at both the byte and 32-bit packet levels. Commands are executed 32 bits at a time, but the data register is 128 bits wide and requires that four sets of 32-bit packets be transmitted. At the 128-bit full payload level, the data transmission needs to occur in the following order: i3, i2, i1, i0, where i*x* is a 32-bit packet. The sequence of instructions is i0, i1, i2 and then i3.

This structure makes the bitstream programming implementation very uniform for CPU×1, CPU×8, CPU×16 and CPU×32 modes. The various potential data orders are illustrated in the example waveforms below, each showing the transmission of the same bitstream contents in the five different CPU widths.

> **Note**
>
> The figures in this section are to show methodologies and generalized scenarios. For detailed waveforms for specific commands, refer to the respective section in FCU Command List. Also, the JTAG ID values in the waveforms below are indicative and not specific to a device.

CPU×32

As shown in the waveform below, a command is issued on each clock cycle in CPUx32 mode:

- The first 128-bit payload shows that the order of loading is NOP, Instance ID, JTAG ID and then Sync, with each 32-bit packet transmitted MSB to LSB. However, as indicated above, the sequence in which these are processed by the FCU are Sync, JTAG ID, Instance ID and finally NOP.
- The second 128-bit payload operates the same way where the 32 bit preamble data is transmitted first, followed by the next three preamble data, but the execution occurring in the reverse order with the 1st preamble data being executed last. Also, when a write or read command is issued, it needs to be the last 32-bit FCU command in the 128-bit sequence. This requirement is because the FCU expects data input or provides data output immediately following the write and read operations respectively.
- When the write command has been issued for a particular frame, subsequent clocks have CMEM frame data transmitted on every clock, again in 128-bit payload sets.

The signal FCU_CPU_CSN must be held low during the entire time when FCU commands are being issued for write operations. If FCU_CPU_CSN is asserted during the (128/CPU_data_width) continuous clock cycles of one request, that request is discarded. When the FCU_CPU_CSN signal returns low, the next request is handled normally.

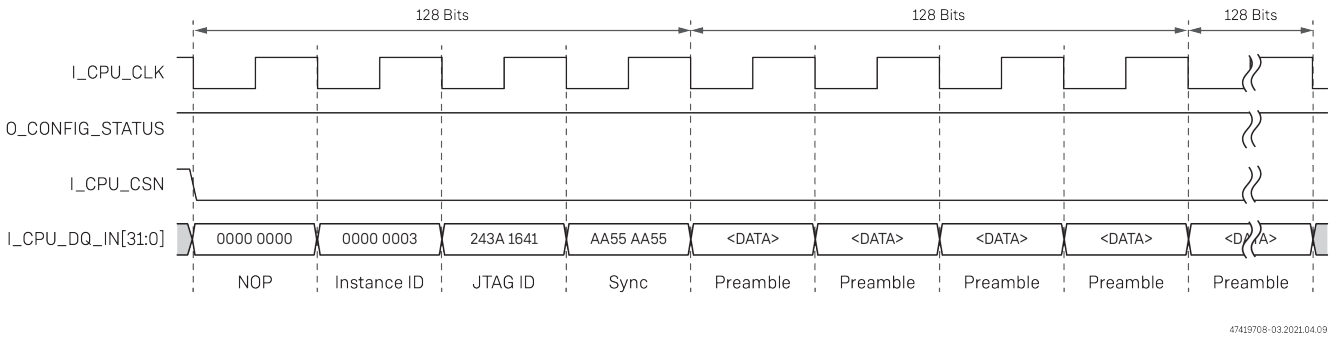

Figure 5: ***Bitstream Programming in CPU×32 Mode***

CPU×16

CPU×16 mode is very similar to CPU×32 mode. The only difference is that 16-bits of data are transmitted on each FCU clock cycle, i.e., each FCU command is transmitted over two FCU clock cycles, MSB to LSB (as shown in the waveform below).

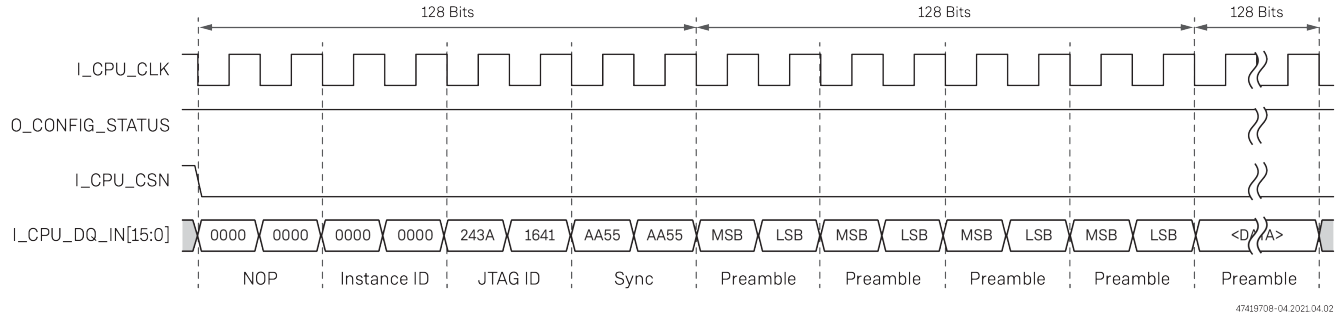

Figure 6: *Bitstream Programming in CPU×16 Mode*

CPU×8

CPU×8 mode follows along the lines of CPU×16 and CPU×32 modes, with each FCU command requiring four FCU clock cycles for transmission, MSB to LSB, as detailed in the waveform below.

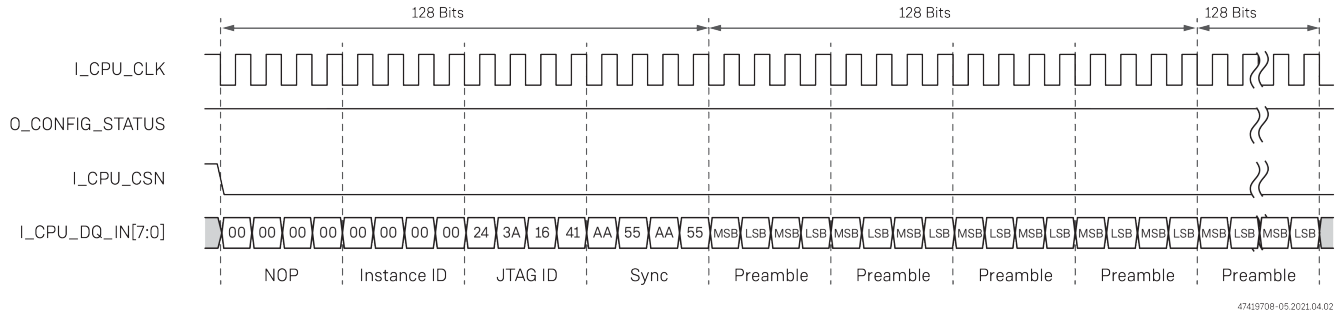

Figure 7: *Bitstream Programming in CPU×8 Mode*

CPU×1

In CPU×1 mode, a single bit of the FCU command (or write data) is transmitted on each FCU clock cycle, MSB to LSB, for a 32-bit packet, but in reverse order for the 128-bit payload as described in the other CPU width modes. The waveform below shows these details.

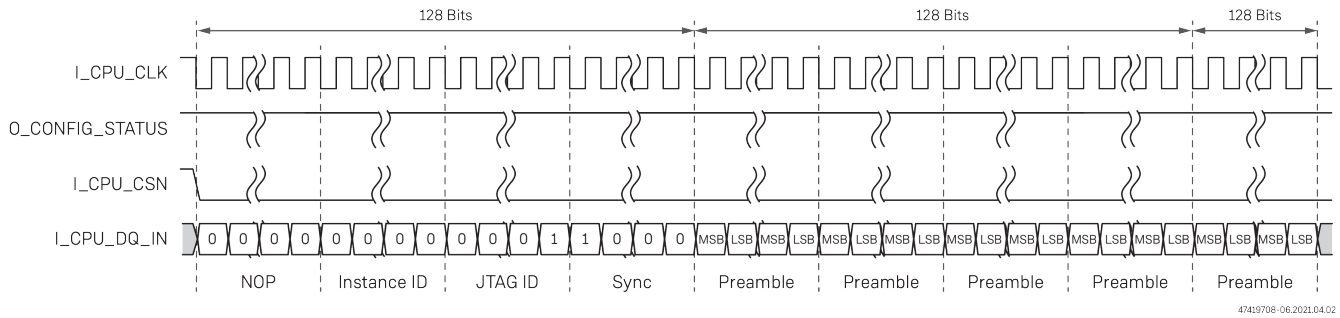

Figure 8: *Bitstream Programming in CPU×1 Mode*

Note

Contact Achronix Support for more details on the CPU x128 mode.

CPU×32 ACE Programming File Example

Preamble 512 bit:

NOP
Instance ID
JTAG ID
Sync
32-bit length preamble to indicate number of 128 bit blocks of unencrypted bitstream
64 bit preamble Information only valid for encrypted bitstream
32-bit length preamble to indicate number of 128 bit blocks of encrypted bitstream
256 bits of zeros that are reserved

NOP
Write Cmd
NOP
NOP
NOP
Write Data
...

Configuration via Flash Memories

Caution!

Speedster7t FPGAs can interface to serial NOR flash devices only. Parallel NOR, NAND or other flash variants are *not* supported.

Flash programming mode allows flash memories to be used to configure Speedster7t FPGAs. In this mode, the FPGA is the master and supplies the clock to the flash memory.

The clock supplied from the FPGA (on the FCU_FLASH_SCK pin) to the attached flash device(s) can be driven by the FCU_CPU_CLK or the on-chip oscillator clock depending on the configuration options selected as described in the Configuration Modes for Speedster7t FPGAs (see page 9) chapter. The frequency of this clock can be selected from one of four variants of the clock sources arriving at the FCU: the original (divide-by-1), divide-by-2, divide-by-4 or divide-by-8. This selection is configured using the 'Serial Flash Clock Divider' drop-down menu in the 'Bitstream Generation Implementation Options' section of the ACE GUI. This setting ensures that only the flash state machine runs at the slower frequency. All other FCU and ACB circuitry continues to operate at the original input clock frequency.

Note

At power-on, the device defaults to the divide-by-4 setting. The FCU then sets the appropriate configuration register to control the clock divider based on the user selection in ACE. The transition from a divide-by-4 clock to any other selected clock frequency is glitch free. Also, flash write is always SPI only while read can be in SPI, DUAL, QUAD or OCTA mode as summarized in the table below.

Flash Interface

The configuration block is equipped with a flash interface that supports flash programming and readback. A bitstream can be programmed into flash through JTAG or the PCIe interface. Flash registers (refer to Registers (see page 26)) within the configuration block assist with this process. The complete feature list of the flash interface is as described in the table below.

Table 3: *Flash Features*

Feature	Description
Programming interface	SPI – JTAG, PCIe (Remote update programming via FPGA).
Security mode	Double encryption.
Device mode	X1, X4.
I/O Mode	SPI, DUAL, QUAD, OCTA.
Flash write	SPI.
Flash Read	SPI, DUAL, QUAD, OCTA.
Header	Holds fallback, current address and read commands.
Two image support	Flash can store two images, golden and new image.
Remote update	Flash can be stored with new or updated images remotely.
Error fallback	If a CRC error occurs in the updated bitstream, the flash interface uses the fallback address specified in the page-0 header.

Flash Device Configurations

Speedster7t FPGAs support two flash device configurations, single flash device (1D) and four flash devices (4D).

1D Configuration

The 1D programming configuration is composed of a Speedster7t FPGA acting as the master and communicating with a single flash device. The `o_flash_sck` signal is used for clocking. The `o_flash_sdi` signal is the data output from the FPGA to communicate instructions to the flash device and `i_flash_sdo[0]` is the single-bit FPGA input pin which receives the bitstream from the flash in ×1 mode. The `o_flash_csn[0]` signal is pulled low as soon as communication between the FPGA and flash device begins, and stays low during the valid bitstream window.

The FPGA can communicate with the flash device in SPI, Dual, Quad or Octa modes in 1D configuration.

The figure below provides a block diagram of how a serial flash device can be connected to a Speedster7t FPGA in Octa (x8) mode.

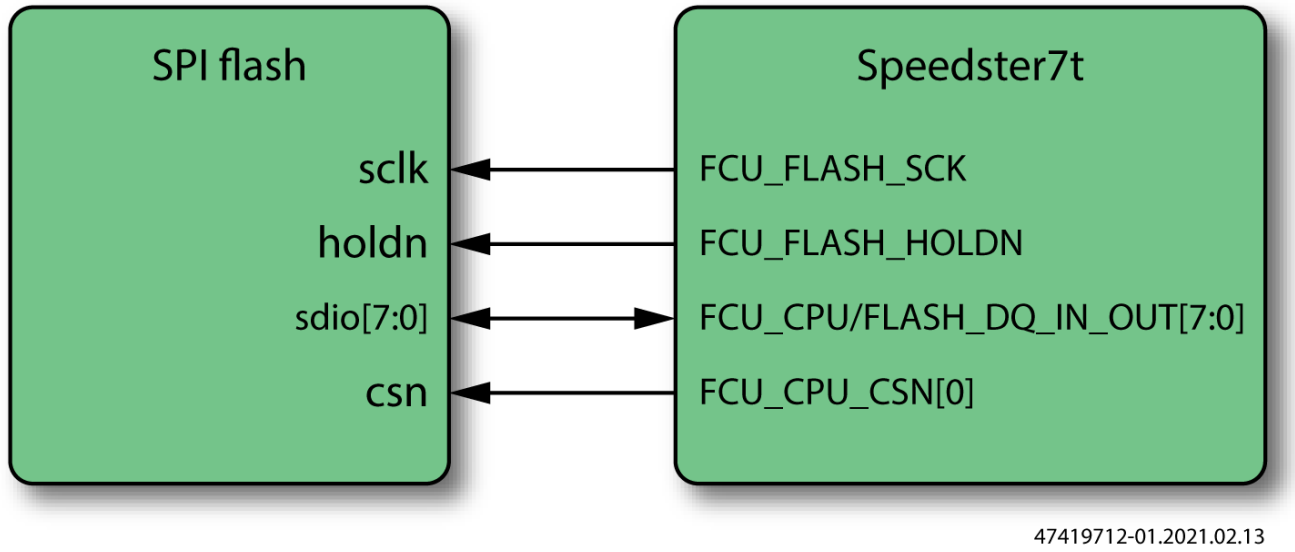

Figure 9: ***Speedster7t FPGA 1D Flash Programming Configuration***

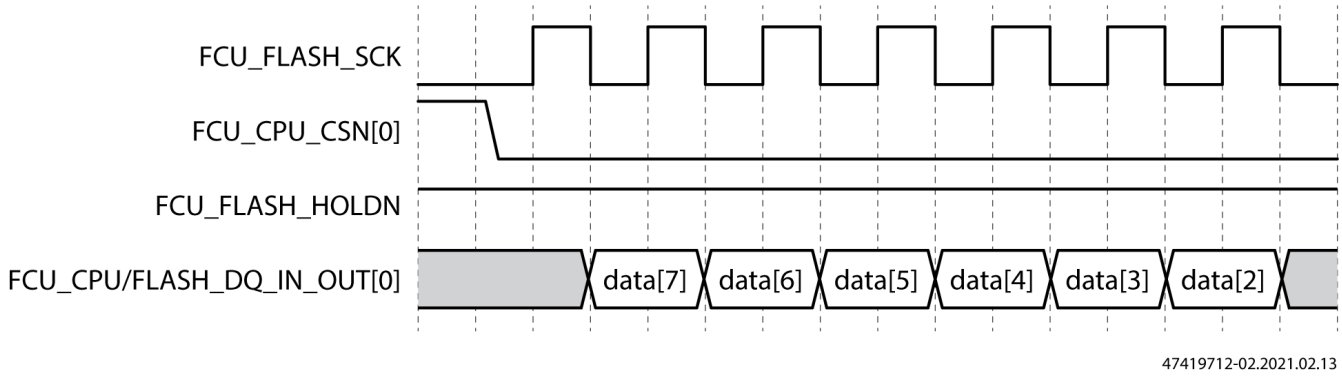

Figure 10: ***Data Ordering 1D Flash SPI Mode Read/Write***

4D Configuration

Serial 4D flash programming mode is essentially an enhanced and higher bandwidth implementation of the serial flash 1D configuration. The FPGA is again the master, and interfaces with not one but four flash memory devices to increase the data bandwidth four times.

When writing to the four flash memories, all four chip selects `o_flash_csn[3:0]` are pulled low simultaneously and 1-bit of bitsream data is sent to each flash device in SPI mode . When reading from the four flash memories, the FPGA pulls all of the `o_flash_csn[3:0]` signals low. Four-wide configuration data is read from the flash memories and transferred to the FPGA through the `i_flash_sdo` ports. When bitstream operations are complete (flash memory contents are read), transitioning from the end of the bitstream to user mode is performed as in CPU and flash 1D modes.

Each flash device can operate in SPI, Dual, Quad or Octa modes. The figure below provides a block diagram of how four serial flash devices (4D configuration) can be connected to a Speedster7t FPGA in Octa (x8) mode.

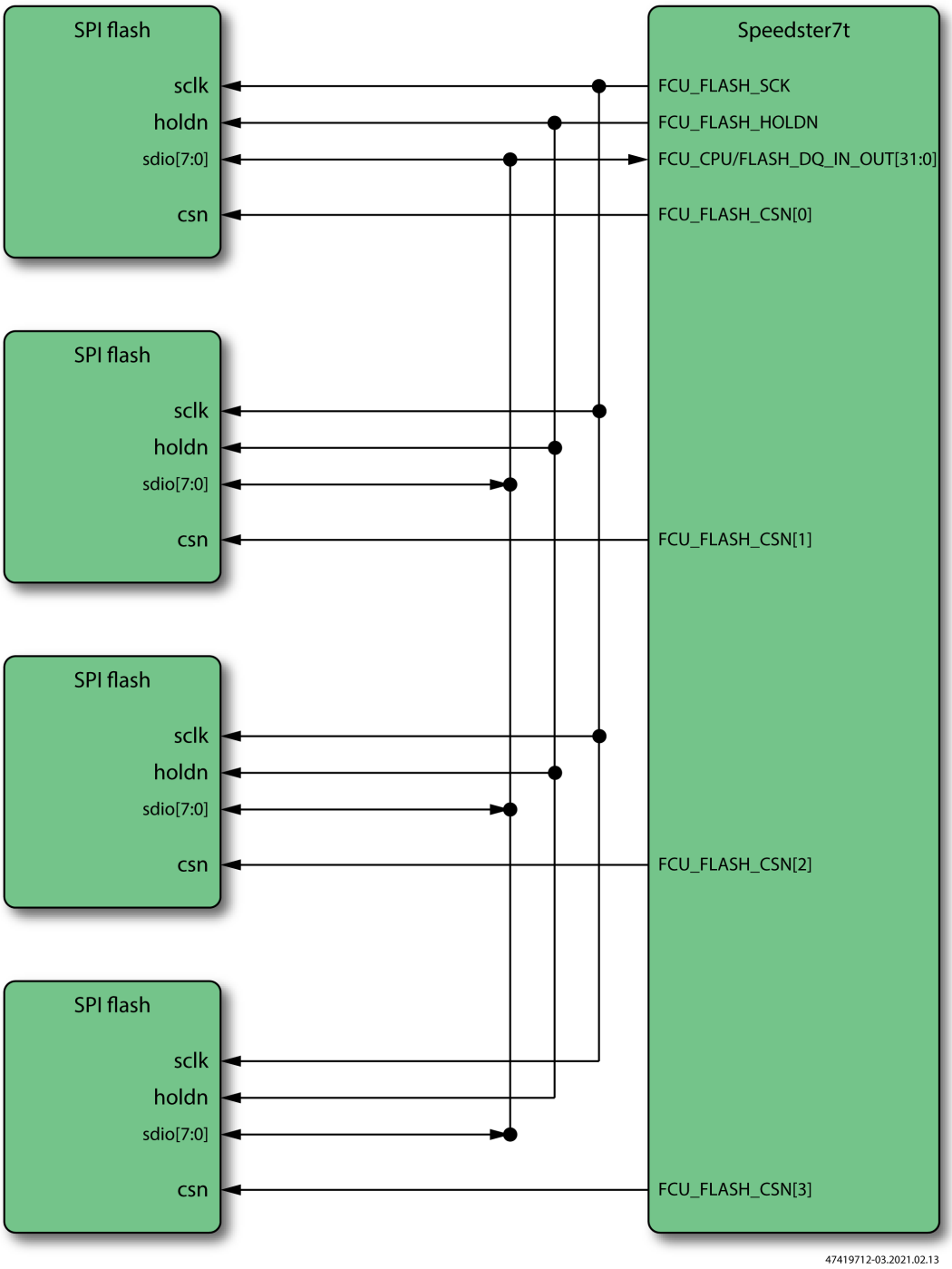

Figure 11: ***Speedster7t FPGA 4D Flash Programming Configuration***

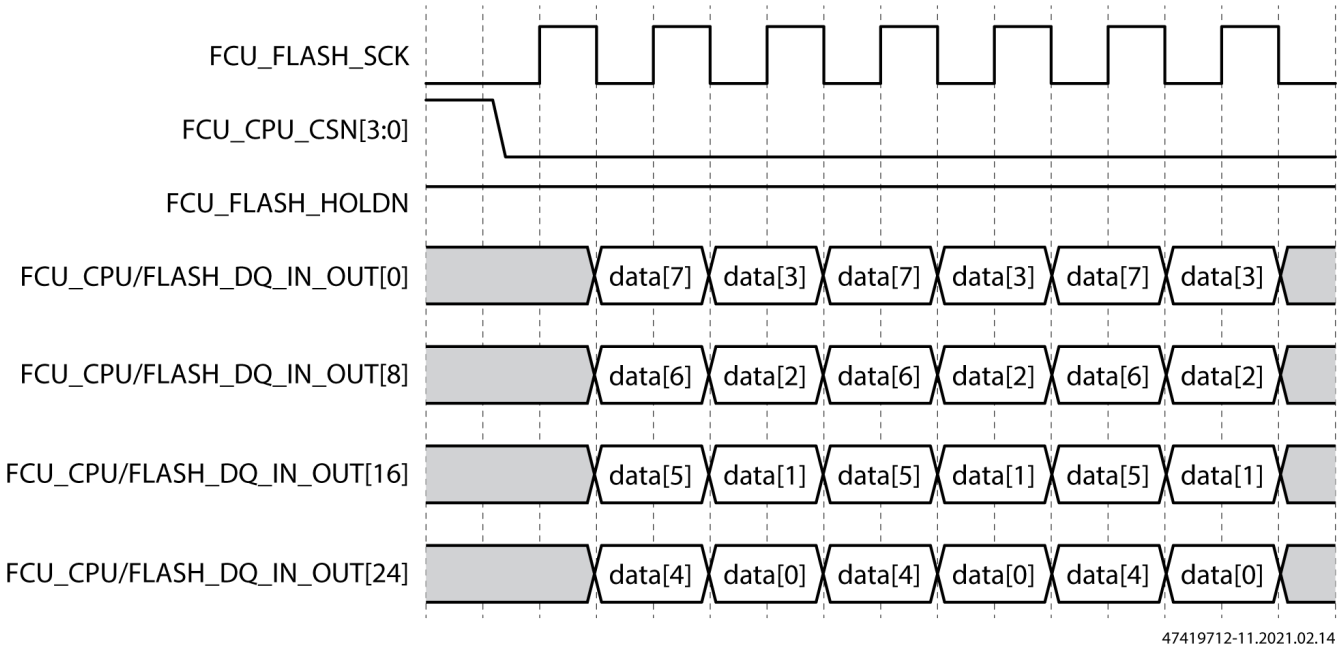

Figure 12: ***4D Flash SPI Mode Read/Write Data Ordering***

Addressing Modes and Memory Organization

Addressing modes for flash memory are based on the size of the device. A three-byte addressing mode is required for 128 Mb flash and smaller, and a four-byte addressing mode is required to support memory sizes above 128 Mb. Writes to the flash memory occur as pages, with each page consisting of 256 bytes. The figure below shows the memory organization:

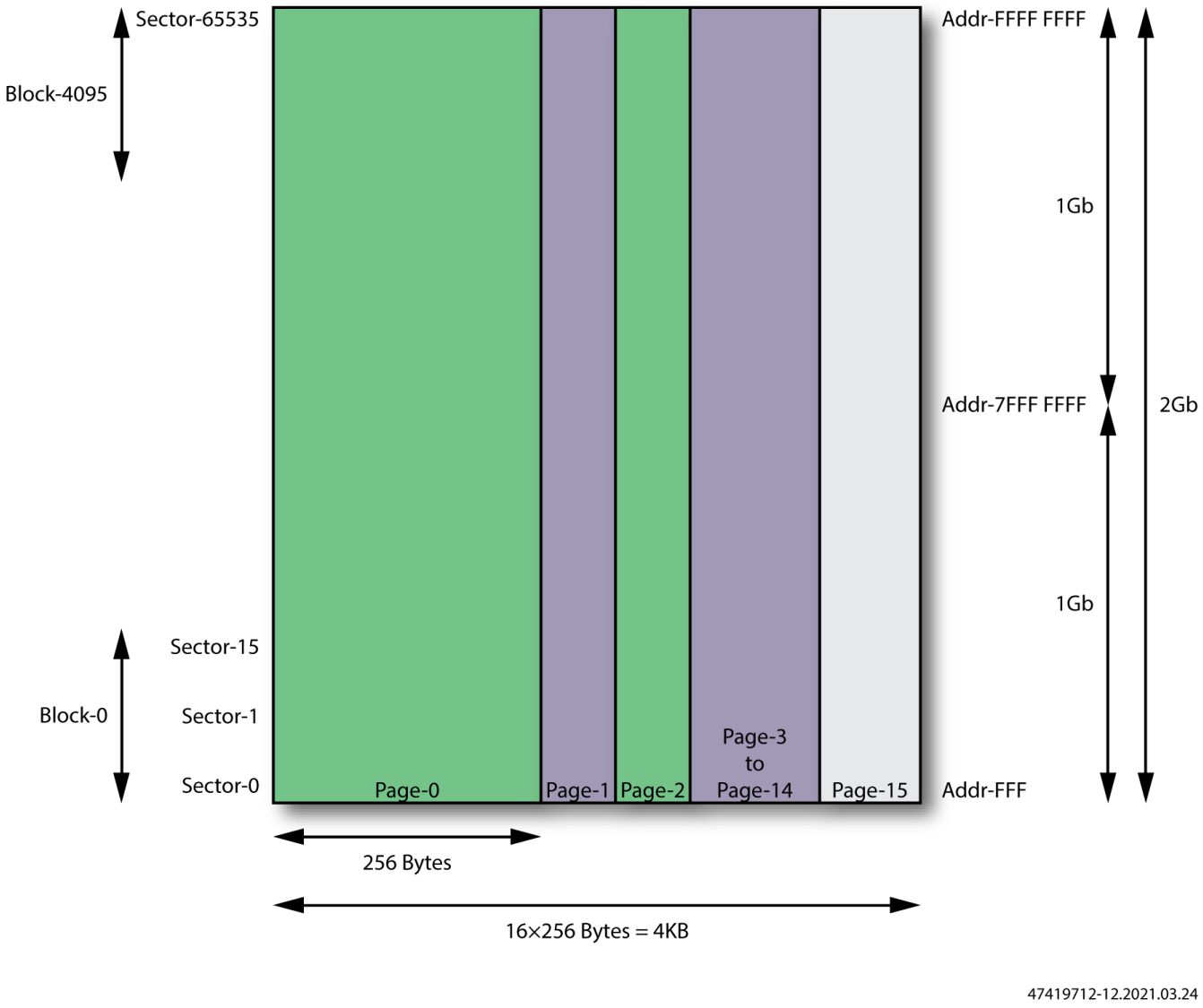

Figure 13: ***Speedster7t FPGA Flash Memory Organization***

Address Range

The following table shows the address ranges when two images are stored on a single flash device, assuming that each image is 1Gb in size.

Table 4: ***Address Ranges for Two Bitstream Images on a Single Device***

Address Range (32 bits)	Description	Configuration Details
0x0000_0000 to 0x0000_00FF	Page-0 address space. This range contains header information described in the flash configuration header section. This address range cannot be used for storing actual bitstreams.	These addresses are not configurable by the user.
0x0000_1000 to 0x07FF_FFFF	FPGA image 1 address space.	The start address can be configured by the user via the current/fallback address in page-0 header. This example assumes the address starts at 0x0000_1000 for a 1 Gb bitstream.
0x0800_0000 to 0x0FFF_FFFF	FPGA image 2 address space.	The start address can be configured by the user via the current/fallback address in page-0 header. This example assumes the bitstream starts at address 0x0800_0000.

Flash Configuration Header (Page-0 Header)

The first 256 bytes in the flash memory (page 0) store control information that describe how the subsequent bitstream should be read from the flash device. This information can be written to the flash device in two ways:

- Via the JTAG interface along with the bitstream.
- Pre-programmed into the device by the manufacturer.

This space is not used for storing the device bitstream.

Table 5: ***Page-0 Header Format***

Address	Bits	Value	Description
0xC to 0xF	32	Read command.	
0x8 to 0xB	32	Flash configuration header read count.	
0x4 to 0x7	32	Bitstream read control.	bit 0 – Flash read enable. bit 1 – Flash fall back enable. bit [5:2] – Retry count. bit [21:6] – Timeout count. bit 22 – Enable 4-byte addressing. bit [27:23] – Dummy read cycles. bit [31:28] – Flash SCK div count.
0x0 to 0x3	32	Bitstream read address (new image).	
0x14 to 0x17	32	Bitstream fallback address (golden image).	

Address	Bits	Value	Description
0x10 to 0x13	32	Fallback read command.	
0x18 to 0x20	24	Reserved.	

Flash Configuration Protocol

With the `FCU_CONFIG_MODESEL[3:0]` , `FCU_CONFIG_CLKSEL` and `FCU_CONFIG_SYSCLK_BYPASS` straps set for serial flash programming, operations begin as soon as the FPGA is powered up and the FCU receives the clock input. Immediately after reset is released, bitstream data is read out from the flash device through the flash interface (at this time the default is SPI (×1) mode). The bitstream read is performed in two stages as described below:

Stage 1 – Flash configuration header read from flash device:

- The FCU sends a default read command and address of 0x0000_0000 (32 bits) in SPI mode to the flash device and reads the flash configuration header.
- Internal registers are then updated, including the start address for the bitstream and flash read command.

Stage 2 – Bitstream read from flash device:

- Based on the read mode (×1/×2/×4/×8) obtained from the flash configuration header, the command and start address are sent to the flash device.
- The FCU reads the first 512 bits of bitstream data from the flash device and enters a wait state.
- If encryption is not enabled, the FCU reads the complete bitstream and configures the FPGA. If encryption is enabled and the efuse key is ready, the FCU reads the header segment0 data and sends it to the secure boot core. The flash read state machine then waits for 2.6 ms after which the FCU reads the complete bitstream and configures the FPGA.

Bitstream programming in all configuration modes is MSB to LSB. For transmitting a 32-bit FCU command, the ordering in the serial ×1 mode for 1D and 4D configuration is as follows:

- 1D flash configuration – The flash device transmits command bit 31 on the first clock and bits 30, 29, 28, etc. on subsequent clocks all the way down to bit 0 on the 32nd (last) clock.
- 4D flash configuration – The four flash devices transmit command bits [31:28] on the first clock, all the way down to bits [3:0] on the eighth (last) clock. The ordering within the 4-bit nibble corresponds to the flash device ordering. Specifically, on the first clock, flash[3] transmits bit 31, flash[2] transmits bit 30, flash[1] transmits bit 29 and flash[0] transmits bit 28

Flash Modes

The following section describes the various modes supported for read and write operations to/from an attached flash device. Read operations from the flash device can be configured either as SPI, Quad or Octa modes for both 1D and 4D configurations while write operations to the flash device always occur in SPI mode.

> **Note**
>
> A flash write can be performed by the user via either the JTAG or PCIe modes. The PCIe or JTAG port can access the data and command registers using an indirect addressing mode.

The following table describes the different combinations of the flash device configurations and modes supported in the Speedster7t FPGA.

Table 6: ***Flash Device Configurations and Modes***

Flash Programming Mode /Configuration	Flash Interface width	No. of Flash Devices	Write Width SO[0] Pin × No. of Flash Devices	Read Width SO[n:0] × No. of Flash Devices
SPI ×1 (1D)	1	1	1	1
SPI ×1 (4D)	1	4	4	4
Quad ×4(1D)	4	1	1	4
Quad ×4 (4D)	4	4	4	16
Octa ×8 (1D)	8	1	1	8
Octa ×8 (4D)	8	4	4	32

Read operation timing diagrams for each of the modes are described below:

SPI Mode (×1)- 1D flash

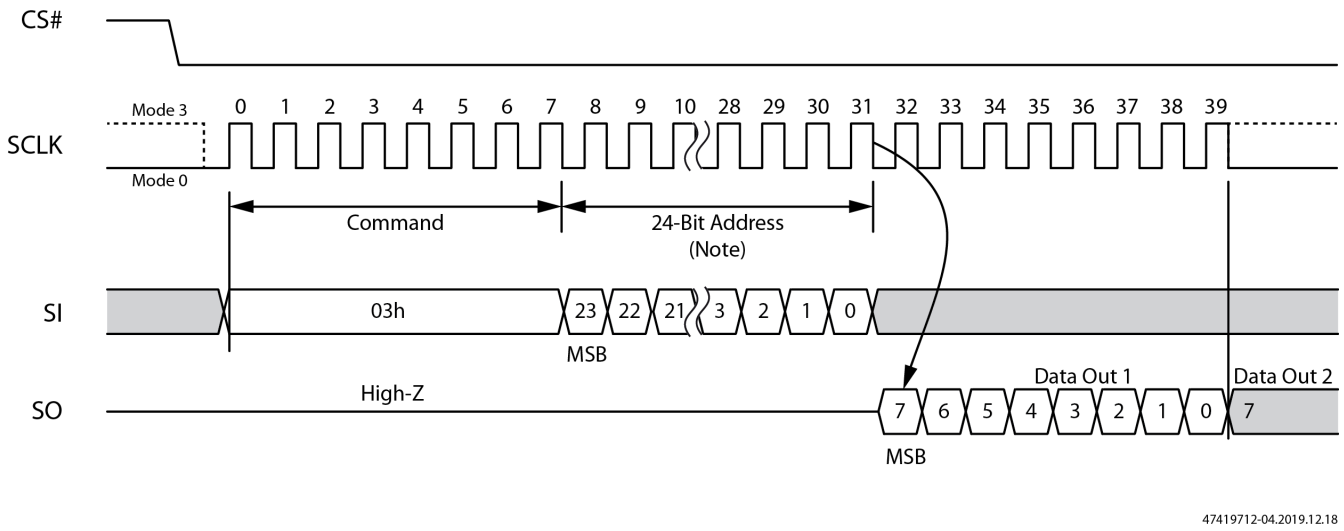

Figure 14: ***SPI Mode (×1) Read***

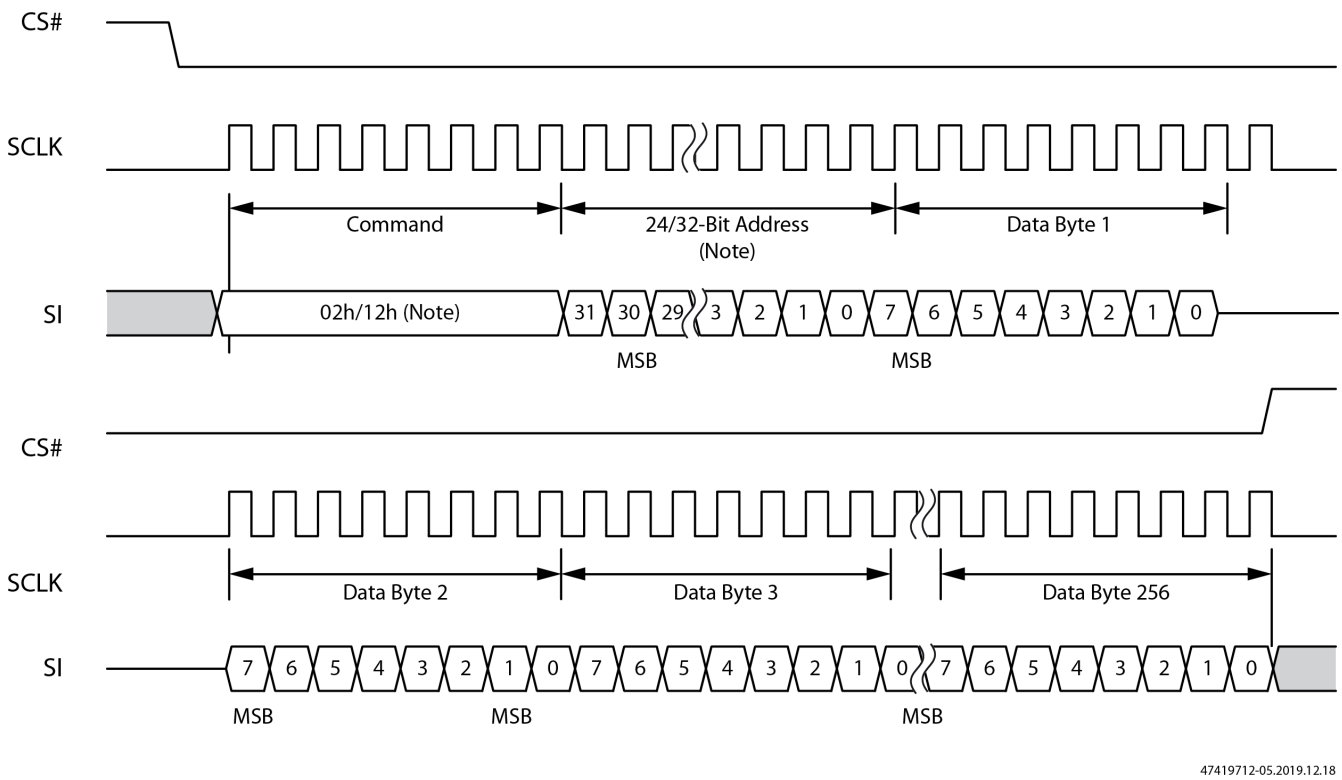

Figure 15: ***SPI Mode (×1) Write***

Quad Mode (×4)- 1D flash

Reads

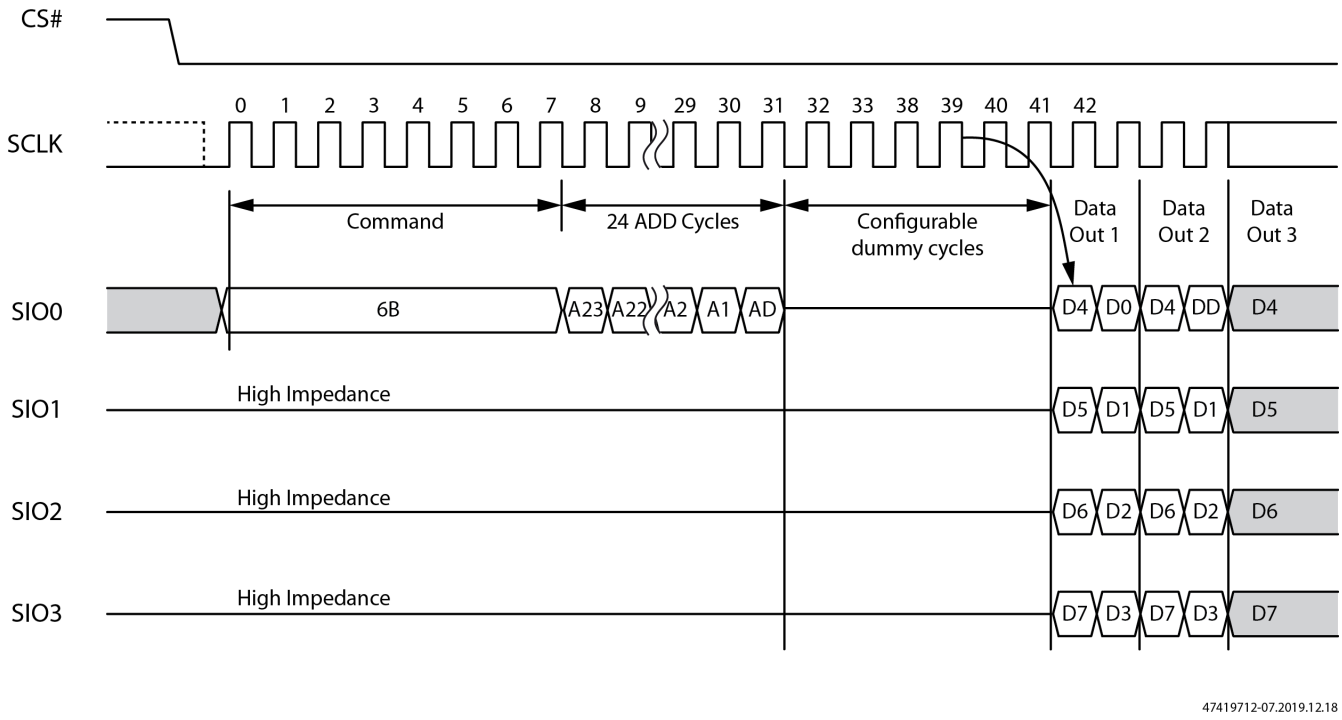

Figure 16: *Quad Read Mode (QREAD)*

Octa Mode (×8)- 1D flash

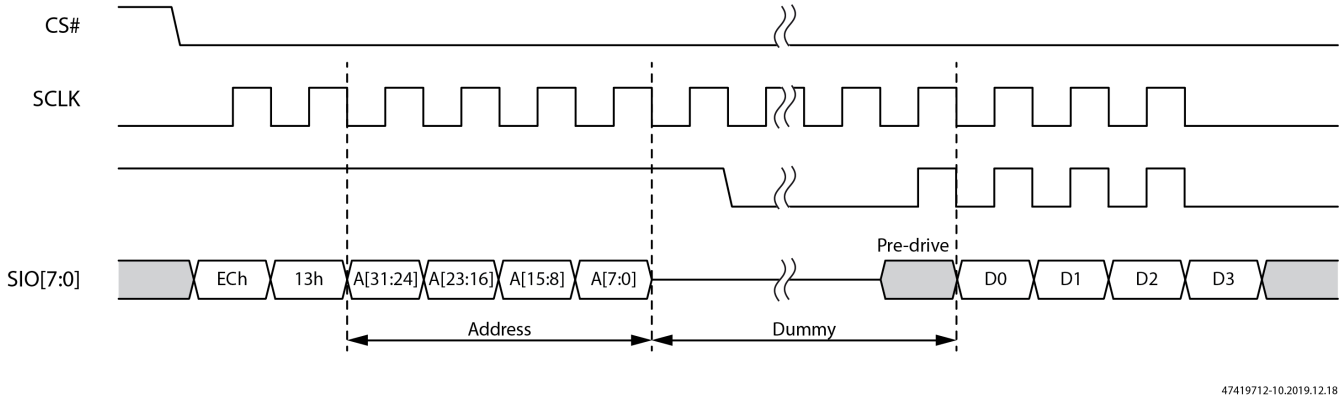

Figure 17: *OCTA Read Mode (8READ)*

Registers and Addressing

Table 7: ***Flash Controller Register Map***

Register Name	Address	Description
Flash write control register	0x1038	Flash write control register.
Flash write count	0x1048	Flash write count register.
Flash write configuration register	0x1050	Flash configuration register.
Flash write status	0x1060	Flash status register.
Flash write data1	0x1040	Flash write data register.
Flash write data2	0x11d4	Flash write data register.
Flash write data3	0x11d8	Flash write data register.
Flash write data4	0x1044	Flash write data register.
Flash bitstream current address	0x12bc	Flash bitstream read current address.
Flash fallback bitstream fallback address	0x12b8	Flash bitstream read fallback address.
Flash write command 1	0x103c	Flash command register.
Flash write command 2	0x104c	Flash command register.
Flash write command 3	0x1054	Flash command register.
Flash write command 4	0x1058	Flash command register.

Table 8: ***Flash Write Control Register***

Register Field	Bit Position	Type	Reset value	Description
Flash write enable	0	RW	0x0	Initiate the flash write operation.
Flash write clock div count	4:1	RW	0x1	Clock divider. Set to 4'b0001 default, divide by 2 clock which is required for JTAG mode.
Flash write Stop	5	RW	0x0	Suspend the current operation.
Flash write wait	6	RW	0x0	Flash wait operation.
Flash write ×1 mode	7	RW	0x0	Flash write in SPI ×1 device mode.

Register Field	Bit Position	Type	Reset value	Description
Flash write ×4 mode	8	RW	0x0	Flash write in SPI ×4 device mode.
Reserved	31:9	RW	0x0	Reserved.

Table 9: ***Flash Write Count***

Register Field	Bit Position	Type	Reset Value	Description
Flash write count	31:0	RW	'h0000_0000	Number of 128-bit blocks of data written to flash.

Table 10: ***Flash Write Data 1***

Register Field	Bit Position	Type	Reset Value	Description
Flash write data 1	31:0	RW	'h0000_0000	Write data to flash.

Table 11: ***Flash Write Data 2***

Register Field	Bit Position	Type	Reset Value	Description
Flash write data 2	31:0	RW	'h0000_0000	Write data to flash.

Table 12: ***Flash Write Data 3***

Register Field	Bit Position	Type	Reset Value	Description
Flash write data 3	31:0	RW	'h0000_0000	Write data to flash.

Table 13: ***Flash Write Data 4***

Register Field	Bit Position	Type	Reset Value	Description
Flash write data 4	31:0	RW	'h0000_0000	Write data to flash.

Table 14: ***Flash Write Configuration Register***

Register Field	Bit Position	Type	Reset Value	Description
Flash write data valid	0	RW [Write on clear]	0x0	Write data valid, Indicates to the flash interface when data is written to the flash write register. Cleared when the flash interface reads the data and writes it into the internal registers.

Register Field	Bit Position	Type	Reset Value	Description
Flash write command valid	1	RW	0x0	Flash write command valid, Indicates to the flash interface when the write command is written to the flash write register.
Flash write command count	8:2	RW	0x8	Write command count in number of bits.
Flash write data count	15:9	RW	0x127	Write data count in number of bits.
Flash write data request	16	R	0x1	Request write data, PCIe, should poll these bits, cleared when data is shifted to internal registers.

Table 15: *Flash Write Status*

Register Field	Bit Pposition	Type	Reset Value	Description
Flash write error	0	RO	0x0	Flash write error, flags flash device status.
Flash read error	1	RO	0x0	Flash read error, CRC error.
Flash write done	2	RO	0x0	Flash write is complete.
Flash read done	3	RO	0x0	Flash read is complete.
Flash state machine status	[8:4]	RO	0x0	Write state machine status.
Reserved	31:9	RO	0x0	Reserved.

Configuration via JTAG

The Speedster7t FPGA JTAG TAP controller is compliant to IEEE Std 1149.1 and is used for programming the bitstream and debug via Snapshot in ACE. The `JTAG_TMS` and `JTAG_TCK` inputs determine whether an instruction register scan or data register scan is performed. `JTAG_TMS` and `JTAG_TDI` are sampled on the rising edge of `JTAG_TCK`, while `JTAG_TDO` changes on the falling edge. JTAG configuration and operation mode is independent of `FCU_CONFIG_MODESEL` settings.

JTAG implementation in Speedster7t FPGAs, which allows for bitstream programming as well as real-time in-system control and observation, is composed of the blocks shown in the figure (see page 29) below.

The external interface is a standard 5-pin JTAG interface, connected directly to the JTAG TAP controller. The TAP controller operates independently from the Speedster7t FPGA FCU. It is always active and uses `JTAG_TCK`

for clocking. The TAP controller takes the data from the pins and converts it to DR instructions to communicate to the JTAG logic in the FCU. It also takes in data in the form of load/read requests, translating it to the appropriate signals to drive and expect on the JTAG pins.

The JTAG logic in the FCU interprets these DR instructions and generates input data in the standard 128-bit Speedster7t FPGA frame size format, along with a data valid indicator, to be forwarded to the FCU data mux and, ultimately, to the FCU state machine for configuration memory loading. The FCU data mux takes in 128-bit output data from the FCU, which also comes with a valid signal for debug and read-back operations. The mux also provides an acknowledge signal to indicate to downstream circuitry that the data transfer was successful.

The FCU data mux simply selects between the configuration mode specific data buses coming in to the FCU. This logic is controlled by the static FCU_CONFIG_MODESEL straps and the JTAG override circuitry from the JTAG TAP controller.

Finally, the FCU state machine takes incoming data and uses it for loading the configuration memory. Conversely, it also provides output data from the configuration memory or Snapshot to be forwarded upstream.

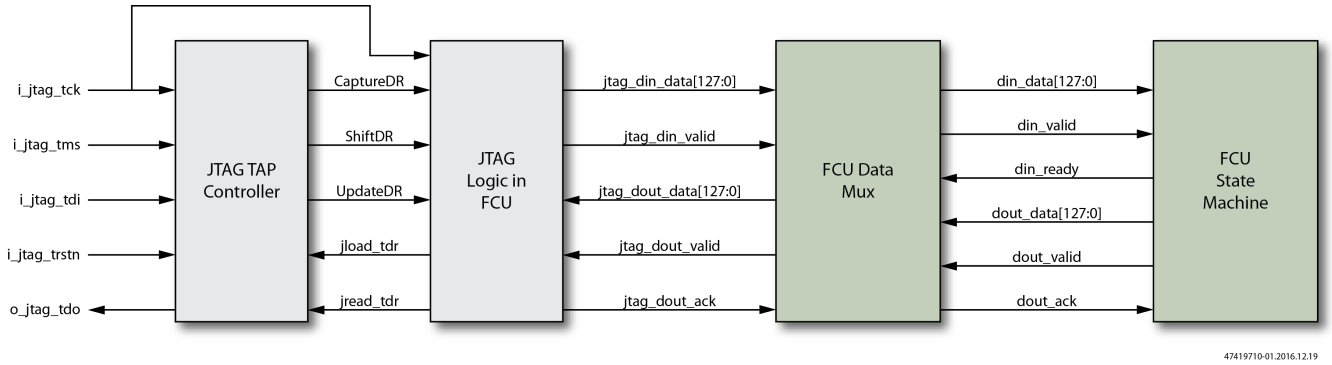

Figure 18: ***Block Diagram for JTAG Instruction Processing in FCU***

The JTAG programming sequence is highlighted in the following waveform (see page 30) to show the sequence of internal procedures that occur in the ACE generated jam file. An explanation of these steps follows:

1. DO_RESET_CHIP – In this step, an internal signal generates a pulse on the FCU reset circuitry to reset it, similar to providing a pulse on the FCU_CONFIG_RSTN input pin.
2. DO_ENTER_JTAG – This step provides a TAP command (override) to place the Speedster7t FPGA FCU in JTAG mode. After this step, regardless of the FCU_CONFIG_MODESEL settings, the FCU configuration mode (and the data muxes) are set to listen to the JTAG inputs, and the FCU clock is sourced from JTAG_TCK.
3. DO_ERASE – This step cycles through the FCU states to ensure that the configuration memory is cleared. After this step, FCU_CONFIG_STATUS is asserted.
4. DO_PROGRAM – In this step, the actual bitstream loading occurs. This operation consists of DRSCAN loops for every bit in the bitstream. Since the size of the bitstream is pre-determined, the loop count is set appropriately by ACE.
5. DO_ENTER_USER_MODE – In this step, IRSCAN and DRSCAN commands are provided to cycle through additional FCU states. Idle clocks are provided to ensure that the start-up state machine completes successfully and, in the process, o_config_done and FCU_CONFIG_USER_MODE are asserted. After this step, functions hosted within a Speedster7t FPGA are active.
6. DO_EXIT_JTAG – This step provides another TAP command, performed in parallel at the start of user-mode operations, to quickly provide additional instructions to remove the JTAG override on the FCU.

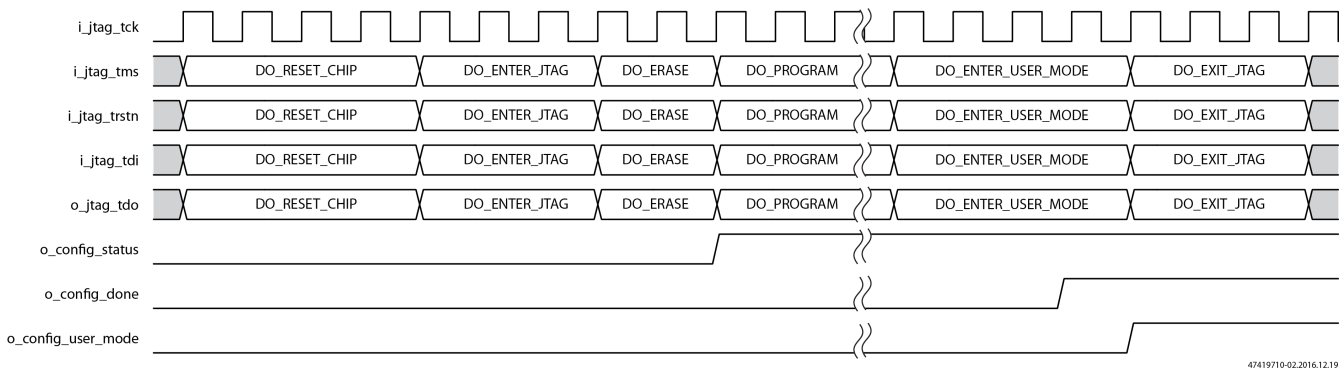

Figure 19: ***JTAG Bitstream Programming Sequence***

JTAG Instructions

The table below lists all JTAG instructions supported by Speedster7t FPGAs.

Table 16: ***JTAG Instructions***

Instruction	Opcode	DR Width	Function
BYPASS	23'b00000000000000000000000	1	The required BYPASS instruction allows a Speedster7t FPGA to remain in a functional mode and selects the bypass register to be connected between `JTAG_TDI` and `JTAG_TDO`. The BYPASS instruction allows serial data to be transferred through the FCU from `JTAG_TDI` to `JTAG_TDO` without affecting the operation of the Speedster7t FPGA.
EXTEST	23'b11111111111111111101000	–	The required EXTEST instruction places the Speedster7t FPGA into an external boundary-test mode and selects the boundary-scan register to be connected between `JTAG_TDI` and `JTAG_TDO`. Output pins operate in test mode, driven from the contents of the boundary-scan update latch. Input data are captured in boundary-scan latches prior to shift operation. During this instruction, the boundary-scan register is accessed to drive test data outside of the Speedster7t FPGA via the boundary outputs and receive test data from outside of the Speedster7t FPGA via the boundary inputs.
EXTEST_PULSE	23'b11111111111111111101001	–	The EXTEST_PULSE instruction generates a single pulse by entering and exiting the Run-Test /Idle state of the 1149.1 TAP controller.

Instruction	Opcode	DR Width	Function
EXTEST_TRAIN	23'b11111111111111111101010	–	EXTEST_TRAIN generates a stream of pulses while in the Run-Test/Idle state. A BSDL file for an 1149.6 device specifies the minimum number of pulses and the maximum time period allowed for pulse generation in the Run-Test/Idle state.
SAMPLE /PRELOAD	23'b11111111111111111111000	–	The required SAMPLE/PRELOAD instruction allows a Speedster7t FPGA to remain in its functional mode and selects the boundary-scan register to be connected between JTAG_TDI and JTAG_TDO. The output and input pins operate in normal mode. Input pin data and core logic output data are captured in the boundary-scan latches. During this instruction, the boundary-scan register can be accessed via a data scan operation to take a sample of the functional data entering and leaving the Speedster7t FPGA. This instruction is also used to preload test data into the boundary-scan register before loading an EXTEST instruction.
IDCODE	23'b11111111111111111111110	32	The optional IDCODE instruction allows a Speedster7t FPGA to remain in its functional mode and selects the optional device identification register to be connected between JTAG_TDI and JTAG_TDO. The IDCODE register appears between JTAG_TDI and JTAG_TDO after power-up, after the TAP has been reset using the optional TRST pin, or by otherwise moving to the Test-Logic-Reset state.
HIGHZ	23'b11111111111111111001111	–	The optional HIGHZ instruction sets all outputs (including two-state as well as three-state types) to a disabled (high-impedance) state and selects the bypass register to be connected between JTAG_TDI and JTAG_TDO.
CLAMP	23'b11111111111111111101111	–	The CLAMP instruction provides for “guarding” chip outputs during in-circuit test or boundary-scan functional test. Output pins operate in test mode, driven from the content of the boundary-scan update latch. The one-bit bypass register is selected for shifting.
INTDR	23'b00000000000000000111101	97	This instruction provides access to the test data register that is implemented internal to the TAP controller. This internal register is used for global configuration and monitoring of global status signals.

Instruction	Opcode	DR Width	Function
JLOAD	23'b00000100000001100111010	128	The JLOAD instruction enables the scan in of the configuration bitstream to the configuration logic (in this mode, the SHIFT-DR state is used to scan in the bitstream). For the read-back, the data register is read back. All of these operations are performed internally using a 128-bit parallel bus. Data is latched every 128 bits in the UPDATE-DR state.
JREAD	23'b00000100000001000111010	128	The JREAD instruction enables the data register for read-back. When this instruction is decoded and CAPTURE-DR is executed, the data from the configuration logic is sampled as 32-bit data plus a valid bit. Multiple words of the configuration memory can be read back by cycling through the CAPTURE-DR/SHIFT-DR states. The 33-bit status register is selected between `JTAG_TDI` and `JTAG_TDO`.
JUSR1	23'b00000100000000100111010	User defined	The JUSR1 instruction enables the USER1 TDR.(†)
JUSR2	23'b00000100000000000111010	User defined	The JUSR2 instruction enables the USER2 TDR.(†)
JASYNCERR	23'b00000000000001110111010	–	The JSYNCERR instruction enables the connection to the fabric error status scan register.

Table Note

† This TDR is implemented in the fabric and is used for supporting debug functionality in the fabric.

Chapter - 4: Configuration Pin Tables

Table 17: *Interface Pin Table*

<table>
<tr><th>Pin Name</th><th>Direction</th><th colspan="2">Usage</th></tr>
<tr><td colspan="4">Configuration Interface</td></tr>
<tr><td rowspan="13">FCU_CONFIG_MODESEL[3:0]</td><td rowspan="13">Input</td><td colspan="2">FPGA configuration unit (FCU) configuration mode selection inputs.</td></tr>
<tr><td>Configuration Mode</td><td>CFG_MODESEL[3:0]</td></tr>
<tr><td>CPU x1</td><td>0011</td></tr>
<tr><td>CPU x8</td><td>0100</td></tr>
<tr><td>CPU x16</td><td>0101</td></tr>
<tr><td>CPU x32</td><td>0110</td></tr>
<tr><td>CPU x128</td><td>0111</td></tr>
<tr><td>Flash SPI (x1)-1D</td><td>0001</td></tr>
<tr><td>Flash SPI (x1)-4D</td><td>0010</td></tr>
<tr><td>Flash Quad (x2)-1D</td><td>1000</td></tr>
<tr><td>Flash Quad (x2)-4D</td><td>1010</td></tr>
<tr><td>Flash Quad (x4)-4D</td><td>1011</td></tr>
<tr><td>Flash Octa (x8)-1D</td><td>1100</td></tr>
<tr><td>Flash Octa (x8)-4D</td><td>1101</td></tr>
<tr><td>JTAG</td><td>Always active mode</td></tr>
<tr><td>FCU_CONFIG_STATUS(4)</td><td>Inout(6)</td><td colspan="2">Active-high configuration status open-drain output signal indicating that the FCU has completed initial start-up and has cleared the CMEM and is awaiting FCU commands for bitstream programming. When high, it remains asserted until the FCU is power cycled or reset for a re-initialization sequence or a CRC error is seen during bitstream load.</td></tr>
<tr><td>FCU_CONFIG_DONE(4)</td><td>Inout(6)</td><td colspan="2">Active-high configuration done open-drain output signal indicating that bitstream loading completed successfully and that the device is ready to enter user mode. When high, it remains asserted until the FCU is power cycled or reset for a re-initialization sequence. If a device configuration error occurs, the CONFIG_DONE output will remain low. Holding this pin low on the board must be used as a method to synchronize the start-up of multiple devices.</td></tr>
<tr><td>FCU_CONFIG_RSTN(1)</td><td>Input</td><td colspan="2">Asynchronous active-low reset input clearing the configuration memory in the device and the logic in the FPGA configuration unit (FCU).</td></tr>
</table>

<table>
<tr><th>Pin Name</th><th>Direction</th><th>Usage</th></tr>
<tr><td>FCU_CONFIG_USER_MODE(4)</td><td>Output</td><td>Active-high output indicating that the device has transitioned into user mode. When high, it remains asserted until the FCU is power cycled or reset for a re-initialization sequence.</td></tr>
<tr><td>FCU_CONFIG_SYSCLK_BYPASS [FCU_CONFIG_CLKSEL</td><td>Input</td><td>Active-high bypass configuration system clock setting. Along with CFG_CLKSEL, this setting allows for clock selection during programming.
<table>
<tr><th>SYSCLK_BYPASS</th><th>CFG_CLKSEL</th><th>CFG_MODESEL[3:0]</th><th>Configuratio Clock</th></tr>
<tr><td>0</td><td>0</td><td>0000, 0001, 0010, 1000 to 1101</td><td>On-chip Oscillator</td></tr>
<tr><td>1</td><td>0</td><td>0000, 0001, 0010,1000 to 1101</td><td>CPU Clock</td></tr>
<tr><td>X</td><td>0</td><td>0011, 01XX</td><td>CPU Clock</td></tr>
<tr><td>X</td><td>1</td><td>XXXX</td><td>JTAG TCK</td></tr>
</table></td></tr>
<tr><td>FCU_CONFIG_BYPASS_CLEAR</td><td>Input</td><td>Active-high input pin to bypass configuration memory clear during device initialization.</td></tr>
<tr><td>FCU_CONFIG_ERR_ENC[2:0]</td><td>Output</td><td>Active-high bypass configuration system clock setting. Along with CFG_CLKSEL, this setting allows for clock selection during programming.
<table>
<tr><th>FCU_CONFIG_ERR_ENC[2:0]</th><th>Status</th><th>Priority</th></tr>
<tr><td>001</td><td>CRC Error.</td><td>0 (Lowest)</td></tr>
<tr><td>010</td><td>Single-bit/multiple-bit scrubbing error.</td><td>1</td></tr>
<tr><td>011</td><td>Secure Boot Failure OR Security error.</td><td>2</td></tr>
<tr><td>100</td><td>Efuse PUF enrollment error.</td><td>3</td></tr>
<tr><td>101</td><td>Asserted when the AXI interface of the IP configuration space register block does not receive a ready from the master.</td><td>4 (Highest)</td></tr>
<tr><td>Other</td><td>Undefined.</td><td></td></tr>
</table></td></tr>
<tr><td>FCU_LOCK</td><td>Output</td><td>Active-high status bit to indicate the FCU lock/unlock status.</td></tr>
<tr><td>FCU_OSC_CLK</td><td>Output</td><td>This clock is internally generated from a ring oscillator. For debug purposes, it can be bypassed and the external clock, CPU_CLK, can be used.</td></tr>
</table>

<table>
<tr><th>Pin Name</th><th>Direction</th><th>Usage</th></tr>
<tr><td>FCU_PARTIAL_CONFIG_DONE</td><td>Inout(6)</td><td>Active-high configuration done open-drain output signal indicating that bitstream loading completed successfully for partial reconfiguration of the device and that the device is ready to enter user mode.</td></tr>
<tr><td>FCU_STAP_SEL</td><td>Input</td><td>When asserted high, this signal enables the JTAG interface pins to be directly connected to the JTAG controller in the SerDes PMA blocks allowing SerDes configuration, debug and performance monitoring directly from the JTAG interface. For bitstream download and design debug using the JTAG interface, this pin must be held low. For SerDes PMA debug only mode, this pin must be held high.</td></tr>
<tr><td rowspan="6">FCU_STATUS[1:0]</td><td rowspan="6">Output</td><td colspan="2">Status bits showing the FCU state.</td></tr>
<tr><th>FCU_STATUS</th><th>State</th></tr>
<tr><td>11</td><td>fcu_locked</td></tr>
<tr><td>10</td><td>sync_found</td></tr>
<tr><td>01</td><td>ID found</td></tr>
<tr><td>00</td><td>instance ID found / FCU unlocked</td></tr>
<tr><td>FCU_STRAP[2:0]</td><td>Output</td><td>Unconnected spare outputs.</td></tr>
<tr><td colspan="3">JTAG Interface</td></tr>
<tr><td>JTAG_TCK</td><td>Input</td><td>Clock input to the FCU JTAG controller.</td></tr>
<tr><td>JTAG_TRSTN</td><td>Input</td><td>Active-low reset input to the FCU JTAG controller.</td></tr>
<tr><td>JTAG_TDI</td><td>Input</td><td>Serial data input to the FCU JTAG controller. Synchronous to JTAG_TCK.</td></tr>
<tr><td>JTAG_TDO</td><td>Output</td><td>Serial data output from the FCU JTAG controller. Synchronous to JTAG_TCK.</td></tr>
<tr><td>JTAG_TMS</td><td>Input</td><td>Mode select input to the FCU JTAG controller. Synchronous to JTAG_TCK.</td></tr>
<tr><td colspan="3">Flash Memeory Interface</td></tr>
<tr><td>FCU_FLASH_SCK</td><td>Output</td><td>Clock output from FCU to flash memory device(s).</td></tr>
<tr><td>FCU_FLASH_HOLDN</td><td>Output</td><td>Active-low hold output to flash memory device(s). This signal is used to pause serial communications between the Speedster7t FPGA and the flash device without deselecting the device or stopping the serial clock. Synchronous to FLASH_SCK.</td></tr>
</table>

Pin Name	Direction	Usage
FCU_FLASH_CSN[3:0]	Output	Active-low chip select to enable/disable one or more of the attached flash memory devices. For x1 mode, only CSN[0] is used. For x4 mode, connect each CSN[3:0] to a flash device.
CPU Interface		
FCU_CPU_CLK(5)	Input	Input clock from external CPU. The data/address bus is synchronous to this clock.
FCU_CPU_CSN(2)	Input	Active-low CPU mode chip select.
FCU_CPU_DQ_IN_OUT[31:0]	Input/Output	Data Input/Output pins shared between the CPU and Flash interfaces. The CPU interface is inaccessible when the Flash mode is in use and vice-versa.
FCU_CPU_DQ_VALID	Output	Active-high control bit to indicate to the CPU the clock cycles when the CPU_DQ bus has valid read-back data. Synchronous to FCU_CPU_CLK.

Table Notes

1. FCU_CONFIG_RSTN must be held low, and cannot glitch during device power-up. All other input pins need only be stable when i_config_rstn is ready to be released after power-up.
2. Refer to the FCU_CPU_CSN Behavior and Implementation Details section of the user guide for details.
3. All configuration status related output signals are driven from registers. The reset value for these registers is '0', and the transition from '0' to '1' is glitch free after reset de-assertion and when reaching the appropriate FCU states.
4. Refer to the Power-Up and Configuration Sequence section of the user guide for details.
5. FCU_CPU_CLK can either start with a rising or falling edge.
6. This output is an open-drain signal. In the default mode of operation, it is recommended that this signal be connected to an LED as an indicator on the board. In this case, use an external 10 kΩ ±5% pull-up resistor to 3.3V and drive a 1 kΩ resistor to the input of a FET to turn on the LED. If LED usage is not desired, this signal must be pulled-up to 1.8V (FCU_CB_VDDIO) instead using the same 10 kΩ pull-up resistor.

Chapter - 5: FPGA Configuration Unit (FCU)

The term FPGA Configuration Unit (FCU) refers to logic that controls the configuration process of the Speedster7t FPGA. This logic is responsible for the following:

- Receiving data on a variety of core interfaces (depending on the selected programming mode)
- Decoding instructions
- Sending configuration bit values to the appropriate destination (core configuration memory, the core's boundary ring configuration memory, FCU registers, etc.)
- Controls the startup and shutdown sequences that drive resets to the rest of the core
- CRC checks
- SEU mitigation
- Security
- Any core-level housekeeping that occurs on the de-assertion of reset (e.g., clearing of configuration memory)

Overview

The following features are supported by the FCU:

- Multiple configuration modes (see Configuration Modes for Speedster7t FPGAs (see page 9))
- Bitstream CRC (see page 54)
- AES encryption/decryption and bitstream security (Design Security for Speedster 7t FPGA (see page 70))
- Configuration memory scrubbing and SEU mitigation (single-bit error correction, dual-bit error detection) (Configuration Error Correction and SEU Mitigation)
- Read-back(Configuration Memory Read (see page 42))

The FCU has two operating modes:

- **Power-on** – Triggered after the input signal `FCU_CONFIG_RSTN` is driven high. When the FCU state machine starts, it progresses through a number of housekeeping activities, including the clearing of the configuration memory if needed. This housekeeping happens without any additional inputs from the user; all instructions sent via one of the programming interfaces during this time are ignored. At the end of this mode, the output pin `FCU_CONFIG_STATUS` is driven high (it was driven low earlier), and the FCU returns to the instruction processing mode.
- **Instruction processing** – The main mode of operation for the state machine. In this mode, the FCU functions as a simple CPU, processing incoming instructions and sending control signals downstream as directed. Instructions are received on 128-bit boundaries but processed 32 bits per clock cycle. The FCU can request data from the host or stall when it is processing the previous instruction. Depending on the programming interface being used, a set of output status signals generated by the FCU are used to determine how to proceed. Refer to Configuration Modes (see page 9)and FCU Command List (see page 38)for additional details.

Speedster 7t1500 FCU Command List

Bistream programming involves instruction processing and the following commands are supported by the Speedster7t FPGA FCU:

- FCU write and read transactions to registers within the FCU which enable/disable Speedster7t FPGA features and modes.
- ACB writes and reads to configure registers within Interface clusters.
- AXI read and write to IP CSR register space.
- Configuration memory read-back transactions to read contents of frames that are programmed in the fabric.

Table 18: *Configuration Commands*

Command	Shorthand	setting
CONFIG_CMD_NOP	-	4'h0
CONFIG_CMD_REG_WRITE	CMD_W	4'h2
CONFIG_CMD_REG_READ	CMD_R	4'h3
CONFIG_CMD_ACB_WRITE	CMD_AW	4'h4
CONFIG_CMD_ACB_READ	CMD_AR	4'h5
CONFIG_CMD_MEM_READ	CMD_MR	4'h7
CONFIG_CMD_AXI_WRITE	CMD_AXW	4'hC
CONFIG_CMD_AXI_READ	CMD_AXR	4'hD
CONFIG_CMD_CLK_ENABLE	CMD_CE	4'h8
CONFIG_CMD_CLK_DISABLE	CMD_CD	4'h9

Command Formats and Details

Command sets are ordered in 128-bit payload groups at 32-bit packet boundaries. The examples below highlight the ordering and data format for the seven sets of commands available to customers. Each example is illustrated as a 128-bit payload with the color coding indicating the 32-bit boundaries. The bit numbering is used to identify the various 32-bit packets inside the 128-bit data register.

FCU Register Write

An FCU register write consists of two parts, each contained in one 32-bit packet:

- The first packet has the 4-bit opcode for an FCU register write, dummy bits (12'b0) and the 16-bit FCU register address for the write.
- The second packet contains the 32-bit data to be written to that FCU register address.

The FCU register addresses and data configurations are pre-determined by Achronix.

The figures below provide representations of an example 128-bit payload consisting of two writes to FCU registers. There is no requirement for the payload to contain two writes to FCU registers or that there should even be two writes. The sequence could be a write to an FCU register followed by an ACB write or vice versa. If there is a single FCU write command, the other 64-bits can simply be set to all zeros (NOPs).

The two figures represent the same command implementation. The first shows the two FCU writes as they are stored in the data register in the correct bit-order (127 through 0). This is also the order in which the 32-bit packets are transmitted (left-to-right) across FCU clock cycles. The second figure depicts the order in which the 32-bit packets are accessed for execution after they have been loaded.

After the 128-bit payload is loaded, there is one clock to capture the configuration in the data register, four clocks for the FCU to execute each of the 32-bit packets, two clocks to do the actual writing of the data bits to the FCU registers (for two FCU writes), and one final clock for those to take effect. Cycle-accuracy is generally not required here since these commands control modes and features for the Speedster7t FPGA instance.

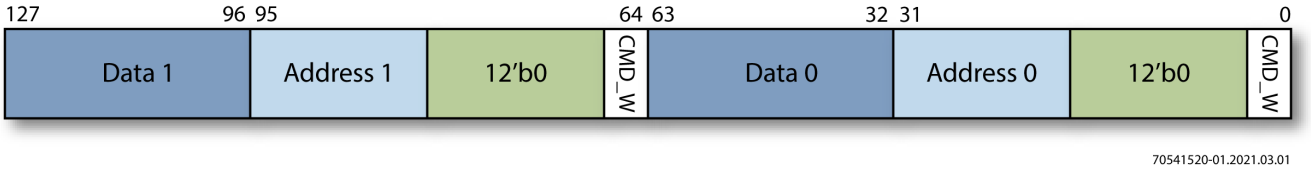

Figure 20: ***Write to Two FCU Registers (Storage/Transmission Order)***

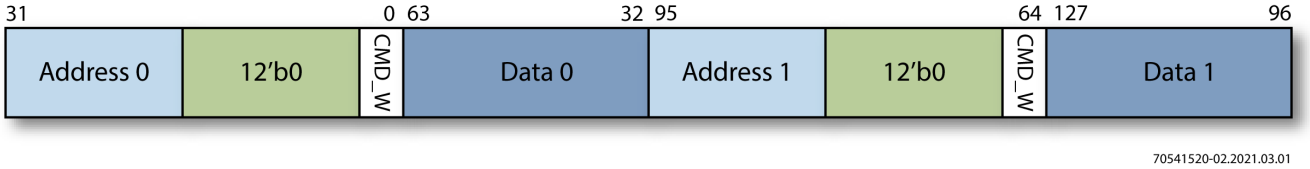

Figure 21: ***Write to Two FCU Registers (Execution Order)***

ACB Register Write

Similar to the FCU register write case, an ACB register write consists of two parts, each contained in one 32-bit packet:

- The first packet has the 4-bit opcode for an ACB register write, dummy bits (4'b0) and the 24-bit ACB address for the write.
- The second packet contains the 32-bit data to be written to that ACB address.

The format for the 24-bit ACB address is specified in the ACB Address Space (see page 60) section of the Achronix Configuration Bus (ACB) Interface (see page 59) chapter.

The figure below provides an example of a 128-bit payload consisting of two writes to ACB registers in execution order. The latency for this write is highly variable, and the selection of blocking vs non-blocking mode also dictates whether or not additional ACB commands can be accommodated before an acknowledgement is received for that write. Refer to the ACB Write and Read Protocols (see page 61) section of the Achronix Configuration Bus (ACB) Interface (see page 59) chapter for further details.

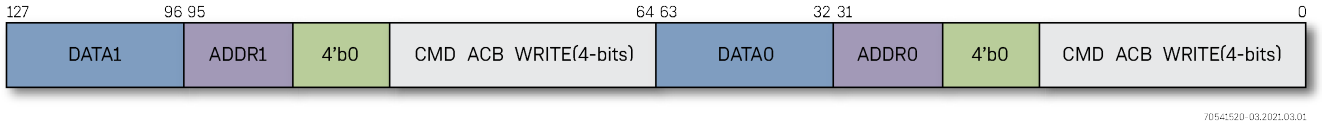

Figure 22: ***Write to Two ACB Registers (Transmission Order)***

FCU/ACB Register Write Waveforms

The two waveforms below detail the sequence of operations in CPU×32 and CPU×128 modes for single and continuous FCU/ACB register write operations. The figures assume that the ACB register writes use the default non-blocking mode of operation. For narrower CPU widths, refer to Programming Data Ordering. (see page 13)

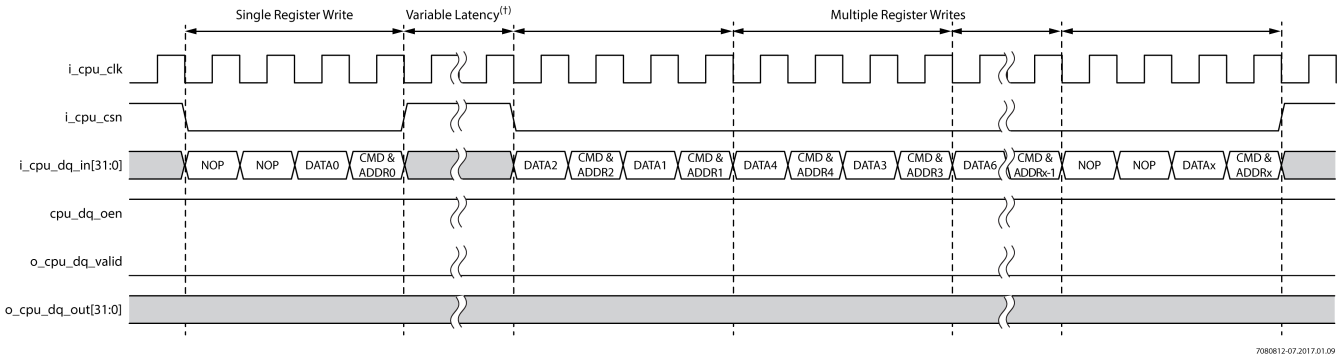

Figure 23: ***Single and Multiple Register Write Operation in CPU×32 Mode***

> **Note**
>
> During bitstream programming in CPUx128 mode, CMEM frame data can be processed by transferring 128 bits of data to the CMEM blocks at every clock cycle. However, FCU and ACB register writes are processed entirely by the FCU, 32 bits at every clock cycle. Therefore, unlike CMEM write data, with FCU and ACB register writes placing a 128-bit payload into the FCU in a single clock cycle, three additional NOP idle clocks are needed to provide time for the FCU to process all four sets of 32-bit packets as shown in the figure above.
>
> In the figure above, denoted by(†), the latency between subsequent sets of FCU commands, in which `i_cpu_csn` is high, can be variable. In fact, if the user wishes to perform back-to-back writes, `i_cpu_csn` need not even be pulled high. The signal can remain low the entire time between these sets of commands. If `i_cpu_csn` is driven high, it can be kept high as long as needed without restriction. When the signal is returned low, the next command is immediately processed by the FCU.

FCU Register Read

An FCU register read command is provided in a single 32-bit packet. However, it *must* be provided as the *last* command in a 128-bit payload, as the execution begins immediately afterward.

Similar to the FCU register write command, the read command consists of the 4-bit FCU register read command opcode, dummy bits (12'b0), and the 16-bit FCU register address to be read. The payload below, shown in execution order, illustrates the sets of NOPs, followed by the actual FCU register read command. As is the case with the FCU register writes, it takes six clock cycles for the packets to be processed and the read command to take effect, after which the read data is returned.

The user should rely on the mode-specific read-valid signal to ensure that the correct data is captured and not try to rely on any cycle-accurate implementation.

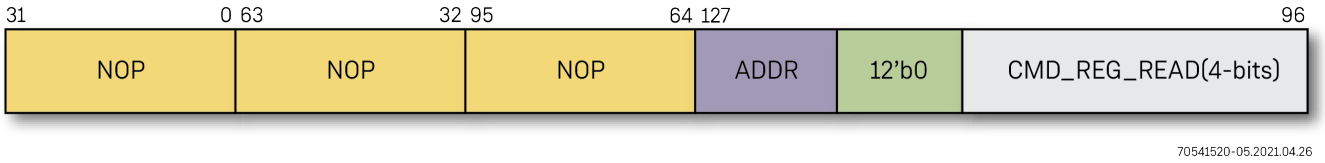

Figure 24: ***Read from FCU Register (Execution Order)***

ACB Register Read

An ACB register read command is provided in a single 32-bit packet. Like the FCU register read, it *must* be provided as the *last* command in a 128-bit payload, as the execution begins immediately afterward.

The ACB register read consists of the 4-bit ACB register read command opcode, dummy bits (4'b0), and the 24-bit FCU register address to be read. The payload below, shown in execution order, illustrates the three sets of NOPs, followed by the actual ACB register read command. Refer to the ACB Write and Read Protocols (see page 61) section of the Achronix Configuration Bus (ACB) Interface (see page 59) chapter for details of the ACB register read operation.

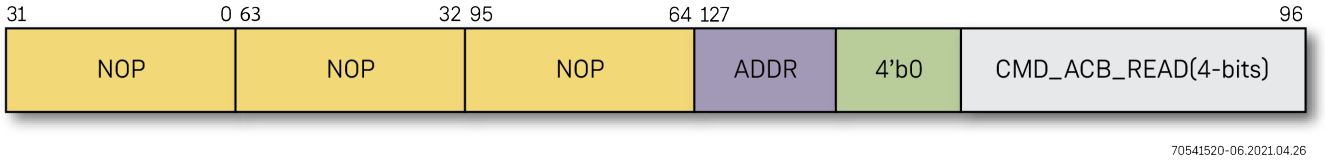

Figure 25: ***Read from ACB Register (Execution Order)***

FCU/ACB Register Read Waveforms

The waveform below details the sequence of operation in CPU×32 modes for back-to-back FCU/ACB register read operations. While there is no burst read function, back-to-back reads can be performed without needing to pull `i_cpu_csn` high, by simply issuing the same 128-bit "READ CMD, NOP, NOP, NOP" payload.

For narrower CPU widths, refer to Programming Data Ordering. (see page 13)

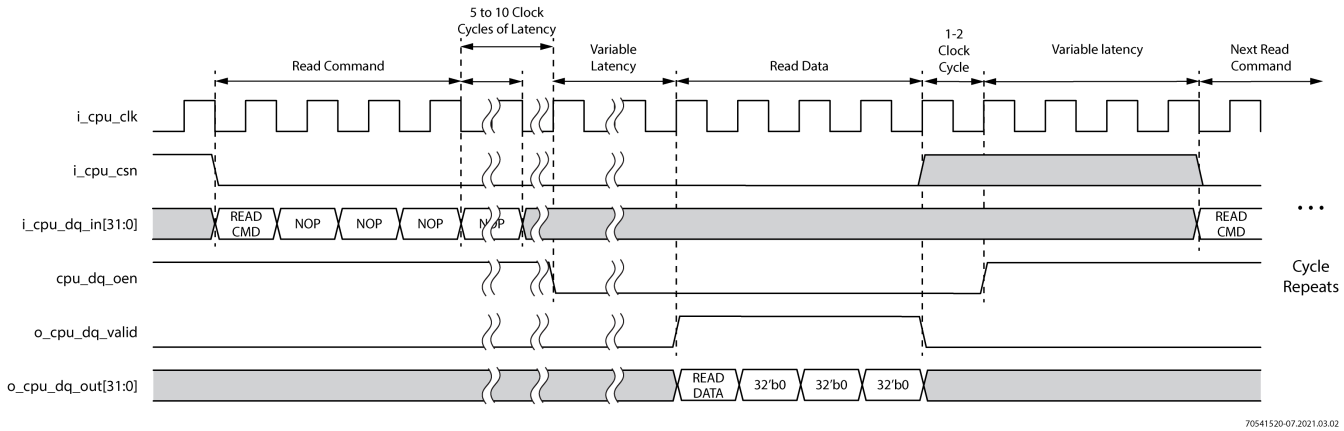

Figure 26: ***Back-to-Back Register Read Operations in CPU×32 Mode***

The FCU/ACB register read operations are very similar to the configuration memory read operations. The one notable difference is that the time between cpu_dq_oen going low and o_cpu_dq_valid going high has variable latency (between 1 clock cycle and a larger number). For FCU register read operations, this would still be 1 or 2 clock cycles. However, for ACB register reads, this would depend on the ACB address and implementation, since the number of pipeline stages in the architecture dictate the latency here. In all cases, this latency is never less than 1 clock cycle, but might be as high as tens of clock cycles depending on the ACB implementation.

AXI Write

The IP registers are configured by providing AXI write commands. The FCU enters AXI write command state, and transfers the data and address to the AXI interface block. AXI write is non-blocking which can accept the 128 bits of data at every 4 cycles. The first cycle consists of bits 127:96 , the second cycle consists of bits 95:64, the third cycle consists of bits 63:32 and the fourth cycle consist of bits 31:0 in CPUx32 mode.

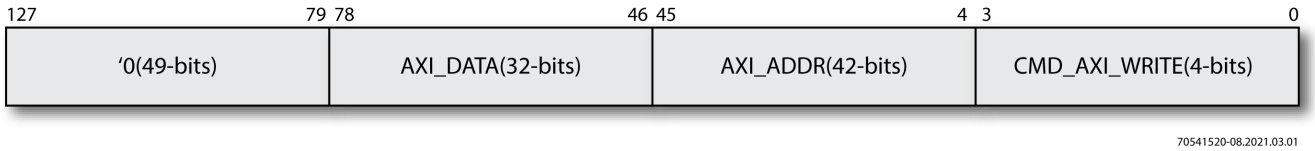

Figure 27: ***AXI Write (Transmission order)***

AXI Read

The IP register read is performed by issuing an AXI read command. The FCU enters AXI read command state and transfers the address to the AXI interface block. The AXI read is a blocking read in that a new read command cannot be accepted until the response from earlier commands is received. The first cycle consists of bits 127:96, the second cycle consists of bits 95:64, the third cycle consists of bits 63:32 and the 4th cycle consist of bits 31:0 in CPUx32 mode.

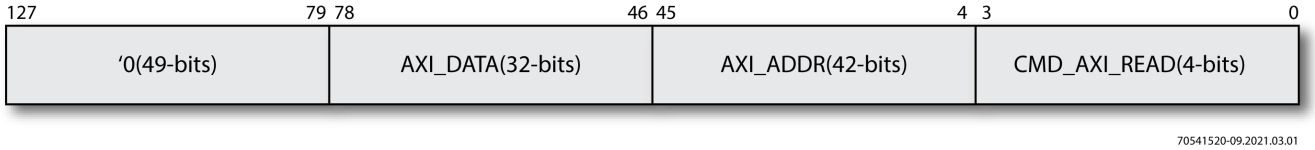

Figure 28: ***AXI Read (Transmission order)***

Configuration Memory Read

The Speedster7t FPGA supports reading back content for the CMEM, LRAM, BRAM bits and RLB registers by sending configuration memory read commands to the designated frame addresses. While the exact read-back command is specified in the Configuration Commands (see page 38) table above, the read operation for a specific frame, or set of frames, is somewhat complex. The procedure is described in detail below for both CPU×32 and CPU×128 modes.

Note

Between the bitstream configuration and read-back operations, there can never be valid data on the `i_cpu_dq_in` and `o_cpu_dq_out` buses at the same time. Therefore, it is possible to use bidirectional I/O for both the input and output bus functions, with the `cpu_dq_oen` signal specifying which direction to set the bidirectional I/O based on the designated FCU operation.

Read-back data from the FCU is not a single continuous stream for the entire fabric. Every FCU read-back command triggers the retrieval of data from one fabric frame/address. The data for a single frame arrives on back-to-back cycles. However, there are gaps between frames as read-frame commands are issued.

The Speedster7t FPGA supports read-back in both JTAG and CPU modes:

- Read-back in JTAG mode is performed entirely by instructing ACE to generate the necessary STAPL commands and uses the JTAG pins to scan out and ultimately compare the resulting data with the content of the configuration bitstream.
- For read-back In CPU mode, ACE generates a file with a list of frame addresses and programmed data at those addresses. With this information, it is then up to the user to generate the set of commands and implementation in the desired CPU width, based on the details in this section, to successfully implement the read-back.

Note

Read-back operations for the configuration memory (CMEM), BRAM and LRAM *cannot* be enabled while scrubbing is active because scrubbing requires the ability to read core fabric frame contents, and as needed, repair and write back into the CMEM. Since this circuitry is common and shared for CMEM /BRAM/LRAM read-back operations and CMEM scrubbing operations, scrubbing must first be disabled before a successful CMEM/BRAM/LRAM read-back command can be issued. There are no such restrictions for performing RLB, FCU or ACB register read-back when scrubbing is active.

Data Ordering

The CPU mode read-back data ordering is slightly different from what is used for bitstream programming. The bits within the mode width are ordered from MSB to LSB, but the ordering across clock cycles is LSB to MSB. This ordering makes the FCU implementation more uniform as it prevents the need to determine which bits to transmit based on what is being read back. Also, every read command sends back a 128-bit packet; therefore, the user should be aware of this situation and account for it appropriately.

For a 32-bit FCU register read, the ordering of the bits for the different CPU modes is as follows:

1. CPU×128 – `o_cpu_dq_out[127:32]` is set to all zeros. The 32 bits are transferred with the MSB on `o_cpu_dq_out[31]` and the LSB on `o_cpu_dq_out[0]`.
2. CPU×32 – `o_cpu_dq_out[31:0]` contains 32 bits with the MSB on `o_cpu_dq_out[31]` and the LSB on `o_cpu_dq_out[0]`. There are three extra clock cycles with 32 bits of all zeros on each additional clock cycle.
3. CPU×16 – `o_cpu_dq_out[15:0]` contains bits 15 through 0 on the first clock followed by bits 31 through 16 on the second clock cycle. There are six extra clock cycles with 16 bits of all zeros on each additional clock cycle.
4. CPU×8 – `o_cpu_dq_out[7:0]` contains bits 7 through 0 on the first clock, bits 15 through 8 on the second clock, bits 23 through 16 on the third clock and bits 31 through 24 on the fourth clock cycle. There are 12 extra clock cycles with 8 bits of all zeros on each additional clock cycle.

5. CPU×1 – `o_cpu_dq_out[0]` contains bit 0 of the 32 bits in the first clock cycle, bit 1 on the second clock cycle all the way up to bit 31 on the 32nd clock cycle. There are 96 extra clock cycles with a zero on each additional clock cycle.

Commands

The full read-back procedure requires providing clock-enable/clock-disable commands and issuing additional commands to flush out stale data. This process is necessary because there is a two-stage pipe of data registers used in bitstream programming and read-back. The first stage is used to scan data in, and the second stage is used to clock the scanned data into a final register set which can then be used as inputs to the configuration frame array. Therefore, it is critical that the read-back data associated with a command is received with a two read command latency when performing configuration memory read-back

There are four commands needed for the successful implementation of read-back:

1. CONFIG_CMD_CLK_ENABLE – Enables clocking for the second stage of the data registers needed to write or read data from the fabric.
2. CONFIG_CMD_REG_WRITE – This FCU register write is needed to specify the CMEM frame address for the read-back. Since the address is actually a 32-bit field, it cannot be provided as part of the memory read-back command itself.
3. CONFIG_CMD_MEM_READ – This payload simply contains the FCU command to execute the instruction.
4. CONFIG_CMD_CLK_DISABLE – Disables clocking for the second stage of the data registers needed to write or read data from the fabric.

The figures for the 128-bit payloads needed to execute the above commands are shown below.

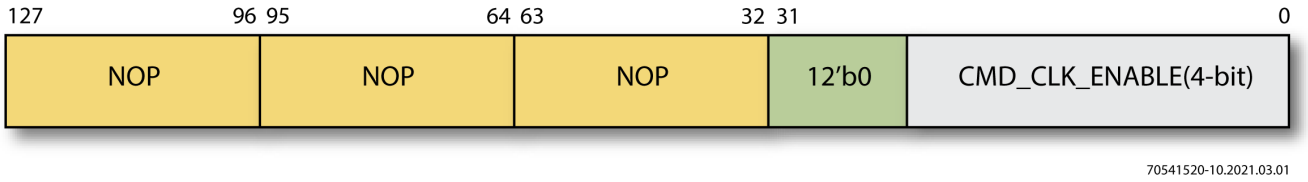

Figure 29: ***Clock Enable Command (Transmission Order)***

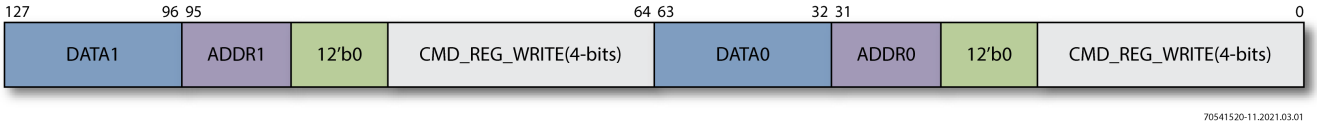

Figure 30: ***FCU Register Writes for CMEM Size and Address Specification (Transmission Order)***

Figure 31: ***Configuration Memory Read Command (Transmission Order)***

Figure 32: ***Clock Disable Command (Transmission Order)***

The sequence of commands, events and expectations when performing a configuration memory read-back of one frame, two frames and the entire Speedster7t FPGA fabric is illustrated in the example command snippet below. The frame address starts at 0. Refer to the Appendix for total frame size and count.

Speedster7t FPGA Read-Back Procedure and Command List

```
(Single Frame Read)
CONFIG_CMD_CLK_ENABLE
CONFIG_CMD_REG_WRITE - Set frame address to 0
CONFIG_CMD_MEM_READ - Ignore return data (Clock enabled)
CONFIG_CMD_CLK_DISABLE
CONFIG_CMD_MEM_READ - Ignore return data (Clock disabled)
CONFIG_CMD_MEM_READ - Capture data at frame address 0 (Clock disabled)
(Two Frame Reads)
CONFIG_CMD_CLK_ENABLE
CONFIG_CMD_REG_WRITE - Set frame address to 0
CONFIG_CMD_MEM_READ - Ignore return data (Clock enabled)
CONFIG_CMD_REG_WRITE - Set frame address to 1
CONFIG_CMD_MEM_READ - Ignore return data (Clock enabled)
CONFIG_CMD_CLK_DISABLE
CONFIG_CMD_MEM_READ - Capture data at frame address 0 (Clock disabled)
CONFIG_CMD_MEM_READ - Capture data at frame address 1 (Clock disabled)
(Readback of the entire fabric - N Frames)
CONFIG_CMD_CLK_ENABLE
CONFIG_CMD_REG_WRITE - Set frame address to 0
CONFIG_CMD_MEM_READ - Ignore return data (Clock enabled)
CONFIG_CMD_REG_WRITE - Set frame address to 1
CONFIG_CMD_MEM_READ - Ignore return data (Clock enabled)
CONFIG_CMD_REG_WRITE - Set frame address to 2
CONFIG_CMD_MEM_READ - Capture data at frame address 0 (Clock enabled)
CONFIG_CMD_REG_WRITE - Set frame address to 3
CONFIG_CMD_MEM_READ - Capture data at frame address 1 (Clock enabled)
...
CONFIG_CMD_REG_WRITE - Set frame address to N
CONFIG_CMD_MEM_READ - Capture data at frame address N-2 (Clock enabled)
CONFIG_CMD_CLK_DISABLE
CONFIG_CMD_MEM_READ - Capture data at frame address N-1 (Clock disabled)
CONFIG_CMD_MEM_READ - Capture data at frame address N (Clock disabled)
```

ACE provides support for performing an entire fabric configuration read-back in JTAG mode by generating the file with all of the necessary commands to run the JTAG operations. For CPU mode, ACE can provide a file with a list of all of the written frame addresses in the currently loaded bitstream configuration, as well as the data written at each address.

Waveforms and Descriptions

The waveforms below show how the above instruction sequences are translated into actual operations. Also, a procedure for a successful configuration memory read of the entire fabric in CPU×32 is detailed. Both of these waveforms appear in two parts. The first part provides a macro view of the signals over time while the second is a detailed view referring to the cycle-per-cycle behavior shown in the portion of the first waveform with the gray background.

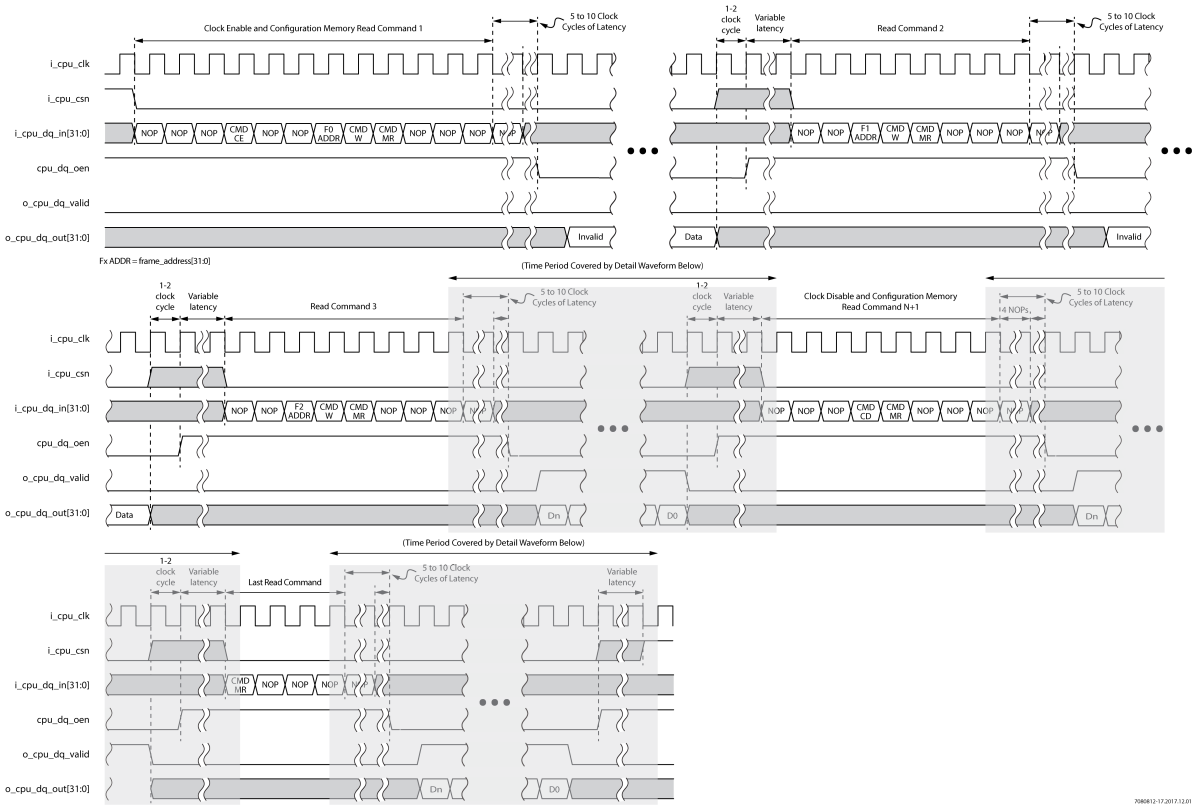

Figure 33: ***Configuration Memory Read of the Entire Fabric in CPU×32 Mode***

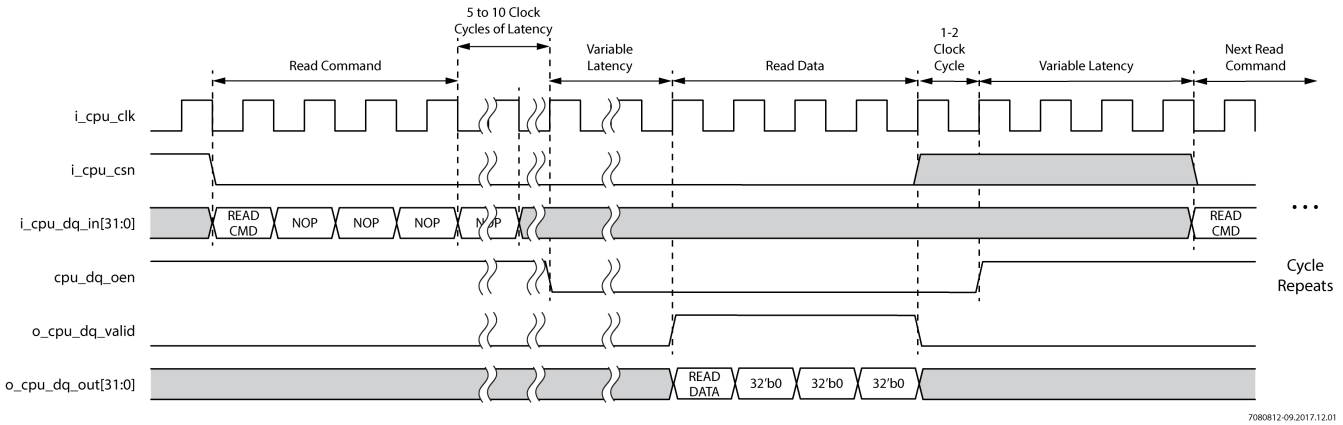

Figure 34: ***Configuration Memory Read of the Entire Fabric in CPU×32 Mode (Detail Waveform)***

Following is a description of the above behavior for CPU×32:

1. The signal `i_cpu_csn` is first pulled low on the negative edge of `i_cpu_clk`. This active-low chip select for the FCU CPU then enables commands and data to be provided on the data input bits.
2. At the first positive edge of `i_cpu_clk` after `i_cpu_csn` is brought low, the first set of FCU commands need to be provided on `i_cpu_dq_in`. As described in the code block sequence above, these are CONFIG_CMD_CLK_ENABLE, CONFIG_CMD_REG_WRITE (for frame 0) and CONFIG_CMD_MEM_READ. This sequence serves as the first dummy configuration memory read.
3. When the FCU receives and processes this command (which can be from 3 clock cycles in x128 mode, 5 in x32 mode, and up to 10 clock cycles after it has been issued), the FCU brings `cpu_dq_oen` low to indicate that the bidirectional I/O must be set to output mode.
4. For a period of *X* clock cycles as shown in the 'detail waveform', the `cpu_dq_oen` signal is driven low. Since this is a dummy configuration memory read, `o_cpu_dq_valid` remains low during this entire period of time.
    a. The value of X is dependent on the CPU data width mode and the number of blocks within the frame. The only other variables that need to be accounted for are the clock cycles between the rising/falling edge of `o_cpu_dq_valid` and the rising/falling edge of `cpu_dq_oen`. The exact formula is:

       [Block_count x (128/CPU_data_width)] + [(2 or 3) for CPU×32 mode and (2 to 5) for CPU×128 mode]

    b. For CPU×32 mode, with a block count of 73 on the Speedster7t FPGA, then X = [73 x 4] + 2 or 3 = 294 or 295.
5. The second set of commands are then issued in much the same way as the first set. As described in the code block sequence above, these are CONFIG_CMD_REG_WRITE (for frame 1) and CONFIG_CMD_MEM_READ. This sequence starts the second dummy configuration memory read.
6. Steps 3 through 5 described above are then repeated to complete the second dummy configuration memory read.
7. After two dummy reads (instruction set 0 and 1), for instruction sets 2 through N, the sets of commands used are the same. As before, CONFIG_CMD_REG_WRITE should be issued (incrementally for each frame 2 through N), followed by CONFIG_CMD_MEM_READ.
8. Unlike the first two dummy reads though, these reads actually return valid data on the `o_cpu_dq_out` bus. The `o_cpu_dq_valid` signal is driven high at the designated points shown in the CPU×32 Mode (Detail Waveform) (see page 47).
9. When the read-back from instruction set N has been completed, a CONFIG_CMD_CLK_DISABLE command is provided before issuing the CONFIG_CMD_MEM_READ command for instruction set N+1, to read back data from frame N-1.
10. A final CONFIG_CMD_MEM_READ command is then issued. This command is the instruction set N+2 to read back from frame N.

note

CMEM read can only be performed on an unencrypted waveform.

Important

It is important to re-iterate the fact that not all CMEM frames are used and that CMEM cells are not populated for every bit of the 128-bit block in every CMEM frame. During bitstream programming, while it might appear as though all bits in the bitstream are being used to program CMEM SRAM cells, in fact, that is not the case. Similarly, for configuration memory read-back, the read-back data shows 0s for bits which do not have actual CMEM SRAM cells associated with them. As described in Bitstream File Generation Through ACE, users should rely on the ACE-generated CMEM bitmap file to identify these sections.

Chapter - 6: Configuration Sequence and Power-Up

The requirements for the power-up and configuration sequencing for the Speedster7t FPGA are illustrated in the table below and followed by a description of each step.

Table 19: *Power-up and Configuration Sequence*

Step	Event
1	Device power-up
2	Read non-volatile memories and BRAM Redundancy
3	Clear configuration memory
4	Bitstream sync, JTAG ID, instance ID and preamble data
5	Load configuration bits
6	CRC
7	Startup sequence
8	User mode

Device Power-Up

The first step in bringing up the Speedster7t FPGA is to appropriately power it up. The Speedster7t Power User Guide details how the power supplies and configuration related pins/signals need to be asserted to ensure a successful power-up. To summarize these requirements:

1. Drive `FCU_CONFIG_RSTN` low.
2. Power-up all supplies to full rail while keeping `FCU_CONFIG_RSTN` low to ensure that the Speedster7t FPGA powers up in a reset state. The FCU clock need not be running at this time.
3. If the `FCU_CONFIG_MODESEL` pins are not statically set (tied off to ground/V_{DD} using resistor loading options), drive them to set the desired configuration mode using the external interface.
4. Drive `FCU_CONFIG_RSTN` high to release the reset. Start providing clocks on the FCU clock.
5. The user must ensure that all clocks used by Speedster7t FPGA are stable when reset is released.

> **Note**
> FCU_CLK is limited to 250 MHz in all configuration modes.

There are no signaling or sequencing requirements for powering down . The supplies can simply be turned off.

Read Non-Volatile Memories

When FCU_CONFIG_RSTN is released and the FCU is supplied with a running clock, device initialization can start. Device initialization begins with the reading of the non-volatile memory (fuse) contents and storing the value in the appropriate FCU registers. These fuses are factory set to zero. Manufacturing and ID related fuses are programmed during device ATE testing. Fuses that pertain to design security are available for customers to program (refer to Design Security (see page 70) for details).

Even if non-volatile memories are not used to enable security and/or redundancy, the FCU requires 17290 clock cycles to shift in BRAM redundancy bits. These clocks need to be factored in when calculating clock cycle requirements for different steps in the power-up and start-up procedures.

After receiving a trigger, the state machines progresses through 32 start-up (or shut-down) states. There is an option to have each state wait for one or more PLLs to lock before continuing to the next state. The final startup state waits for assertion of the FCU_CONFIG_DONE signal before asserting FCU_USER_MODE.

Clear Configuration Memory

After the non-volatile memory is read, the FCU begins to clear the configuration memory one frame at a time by writing 0s to each memory cell location. The user must continue to supply FCU clocks during this operation and at least until the Speedster7t FPGA enters user mode. If scrubbing is disabled, no further clocking of the FCU is needed. If scrubbing is enabled, FCU clocking must continue even in user mode. If the configuration memory clear state is entered after a full FPGA power-up, it is imperative that all configuration memory be cleared.

This step can be bypassed as a debug or optimization step by asserting the FCU_CONFIG_BYPASS_CLEAR pin if the application requires re-configuring the FPGA without a power-down. Bypassing the configuration memory clear requires a very precise implementation to ensure that the subsequent bitstream program load occurs successfully. Care must be taken because a standard configuration file generated by ACE only writes to frames that need to have cells programmed to 1s. 'All 0' frames need not be written to since they are assumed to contain 0s through the configuration memory clear process. However, when FCU_CONFIG_BYPASS_CLEAR is asserted and the configuration memory clear is not performed, the user needs to ensure that frames that had '1's written into in the first bitstream *and* not being programmed in the second bitstream (because they are supposed to be all 0s) are actually programmed to be all 0s and not simply bypassed. The figures below illustrate this concept.

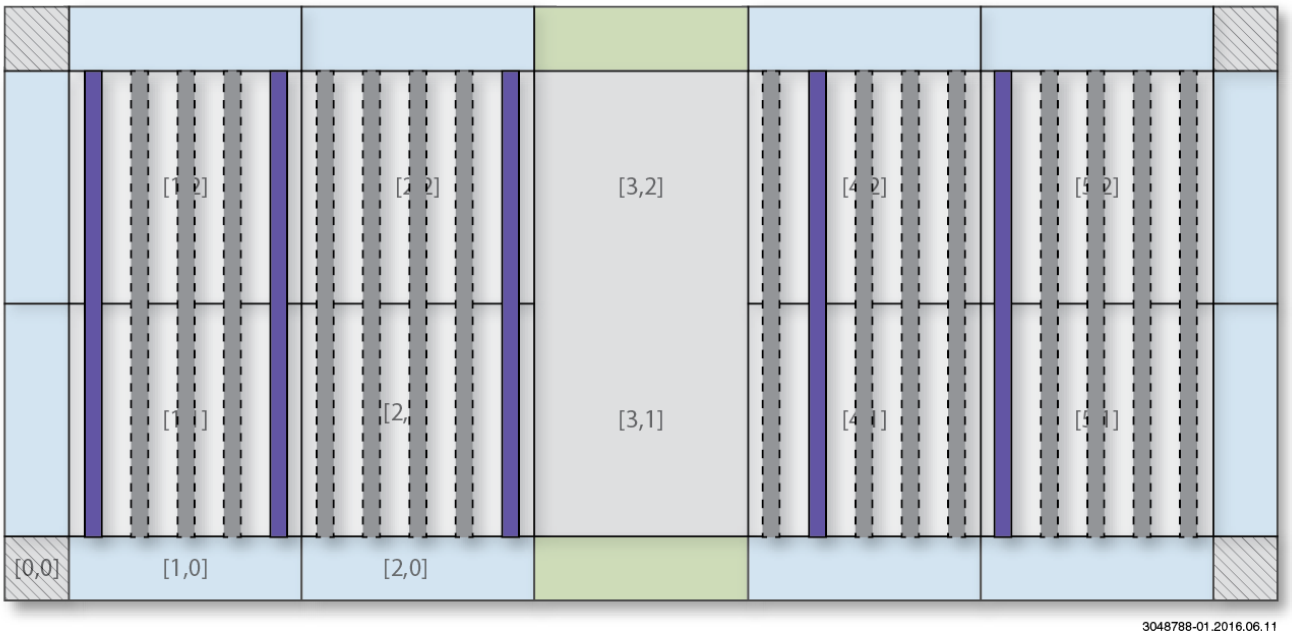

Figure 35: ***Frames Needing to be Programmed in First Bitstream (Shown in Purple)***

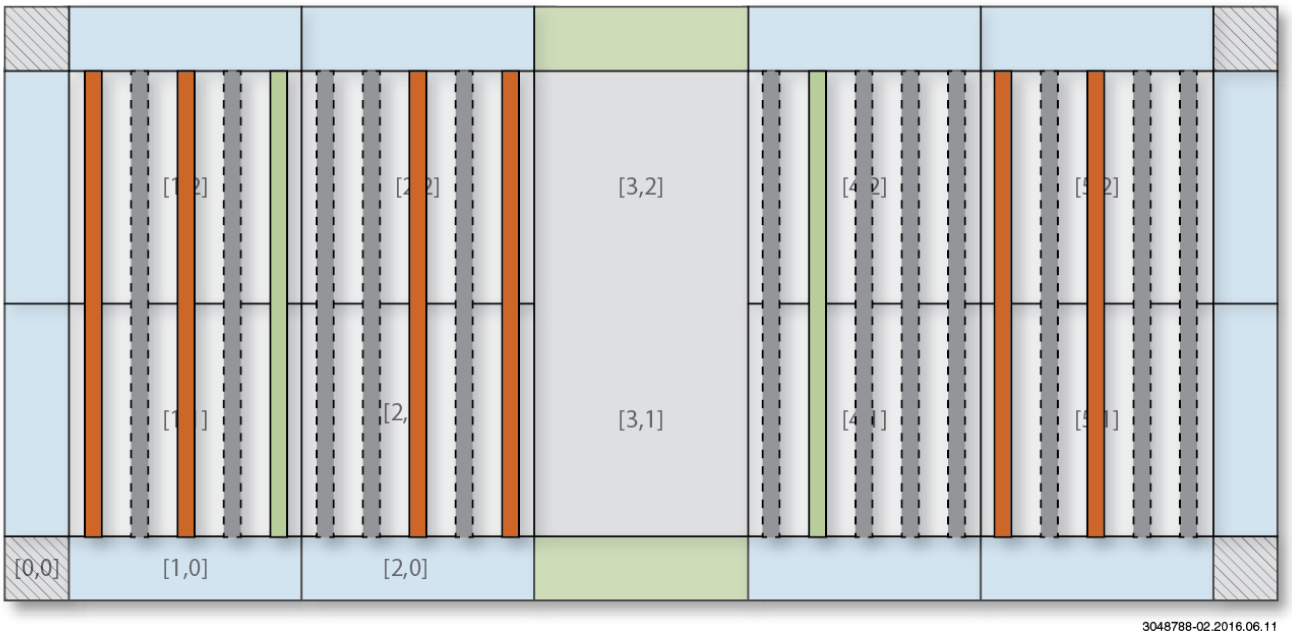

Figure 36: ***Frames Needing to be Programmed in Second Bitstream (Shown in Orange); Green Frames Must be Zeroed***

When the memory clear is complete, the pin FCU_CONFIG_STATUS is asserted high by the device to indicate that the FCU is ready to read the bitstream.

> **Note**
>
> During this step, only the configuration memory is cleared. The embedded BRAM and LRAM memory cells are *not* cleared and should be assumed to power up to unknown states after configuration and in user mode. Preloading memory contents is done separately. See section BRAM72K_SDP-Memory Initialization in Speedster7t FPGA IP Component Library User Guide for details.

When the device powers up, the FCU_CONFIG_STATUS (output pin might temporarily be in an unknown state). Therefore, the FCU_CONFIG_STATUS signal should not be monitored if FCU_CONFIG_RSTN is low. After FCU_CONFIG_RSTN is pulled high, FCU_CONFIG_STATUS can be monitored. When FCU_CONFIG_STATUS goes high, the FCU is ready to accept the bitstream.

Bitstream Sync, JTAG ID and Instance ID

Speedster7t FPGA bitstreams always start with a pre-programmed sync code and one or more device-specific ID codes set by the ACE software. The ID codes are checked to avoid programming Speedster instances with bitstreams meant for other devices.

There are two ID codes present in Speedster7t FPGAs:

1. **JTAG ID** – Hard-coded inside a Speedster7t FPGA instance to differentiate between different variants, products or silicon revisions.
2. **Speedster7t FPGA instance ID** – A unique 4-bit identifiers to distinguish between instances of the same product or variant on a die, package or board.Instance ID is not used in the Speedster7t FPGA, it is set to 4'h0.

The sync and JTAG ID code is followed by a 512 bit preamble to be sent to the security block for processing the encrypted bitstream. If the bitstream sync code ("AA55AA55") is never received or the ID codes do not match, the FCU simply ignores the rest of the bitstream and remains in an idle state.

Load Configuration Bits

The configuration bitstream is a series of data words which are loaded into configuration memory frames in the FPGA fabric. The bitstream also includes command words which control whether the I/O ring configuration registers or the core configuration memory is programmed.

Configuration file sizes and configuration times are directly proportional to the FPGA size and resource counts and the number of configuration memory frames that need to be programmed within the fabric. The configuration file size is also dependent on the programming mode used, but the raw hex file for the bitstream can vary from <1 MB for very small designs to >10 MB for the largest designs on Speedster7t FPGAs that fill the entire fabric and preload the BRAM memories (see also Bitstream File Generation Through ACE (see page 56)).

Preloading the BRAMs is performed in one of two ways:

1. In the RTL by configuring the appropriate BRAM parameters. The 1024 72-bit parameters initd_0 through initd_1023 can be used to designate the initial contents for a particular BRAM instance or set of instances in the design. Refer to section BRAM72K_SDP-Memory Initialization in Speedster7t IP Component Library User Guide for details of BRAM preload.

2. By specifying in the RTL that a memory initialization file needs to be read. This file would contain hexadecimal entries for the required memory contents, with the first entry corresponding to address zero and moving upward.

When the bitstream is generated, ACE reads these parameters and uses them to provide configuration data that allow for programming of the BRAM, SRAM frames along with the CMEM frames.

CRC

If CRC is enabled completely, an accumulative CRC is computed for each 128-bit data packet that passes through the configuration data mux. The final CRC must match a hard-coded value in order to allow a startup or shutdown sequence to begin. The CRC register is set to 32'hFFFFFFFF on reset and whenever the CRC register is written. The current CRC computation can be read back at any time via an FCU register. CRC check can also be completely bypassed.

Startup Sequence

The FCU has a startup sequencing block responsible for the initial power-up sequence out of reset. During power-on and bitstream programming, the startup state machine remains in its default IDLE state. After programming is finished and the chip is ready to be put into user mode, the state machine progresses through a number of startup states, de-asserting resets to the rest of the chip in a certain sequence. The final state of the startup process is user mode where it remains until it receives a request to initiate the shutdown process. The shutdown process is much like the startup process, but performed in reverse (asserting resets along the way) and ending in the IDLE state.

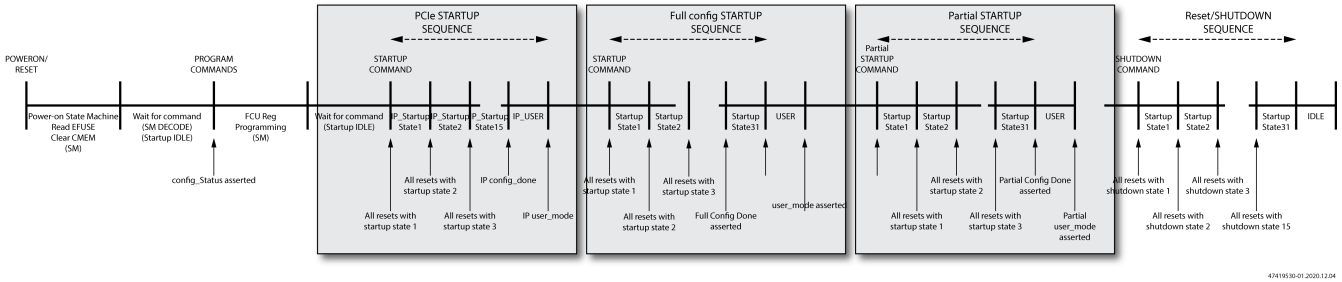

Figure 37: *FCU Startup Sequence*

The FCU startup sequencing block has three stages, the first two to support two-stage programming of the fabric and the third for partial reconfiguration. The FCU startup state machine generates 32 resets where 16 resets are connected to fabric and the other 16 resets are connected to the hard IPs. The fabric resets are staggered to avoid inrush currents.

User Mode

When the device enters user mode as indicated by the assertion of the `FCU_CONFIG_USER_MODE` signal, the design has been fully programmed and the user can start sending and receiving data to/from the FPGA and performing intended operations.

If scrubbing is enabled, the user *must* continue providing free-running clocks on FCU_CLK to have the scrubbing state machine (which is clocked with FCU_CLK) perform the necessary operations. If scrubbing is not enabled,

the user can gate FCU_CLK if and until ACB operations are required. There are no specific requirements for gating FCU_CLK beyond ensuring that it happens glitch free. The clock rest state can be high or low. It is even possible to start issuing ACB clocks before enabling ACB operations through FCU commands.

Chapter - 7: Speedster7t Bitstream Generation

ACE has a straightforward interface to generate the bitstream files required to implement all of the supported configuration modes. The bitstream file is generated during the the 'FPGA Programming – Generate Bitstream' step of the compilation flow (see Flow View in the *ACE Software User Guid*e (UG070) for more details).

The STAPL Jam file needed for JTAG mode configuration is always generated by default. The 'Bitstream Generation' section of the Project Options menu, shown in the following figure, also provides a menu option to generate bitstream files for the other configuration modes.

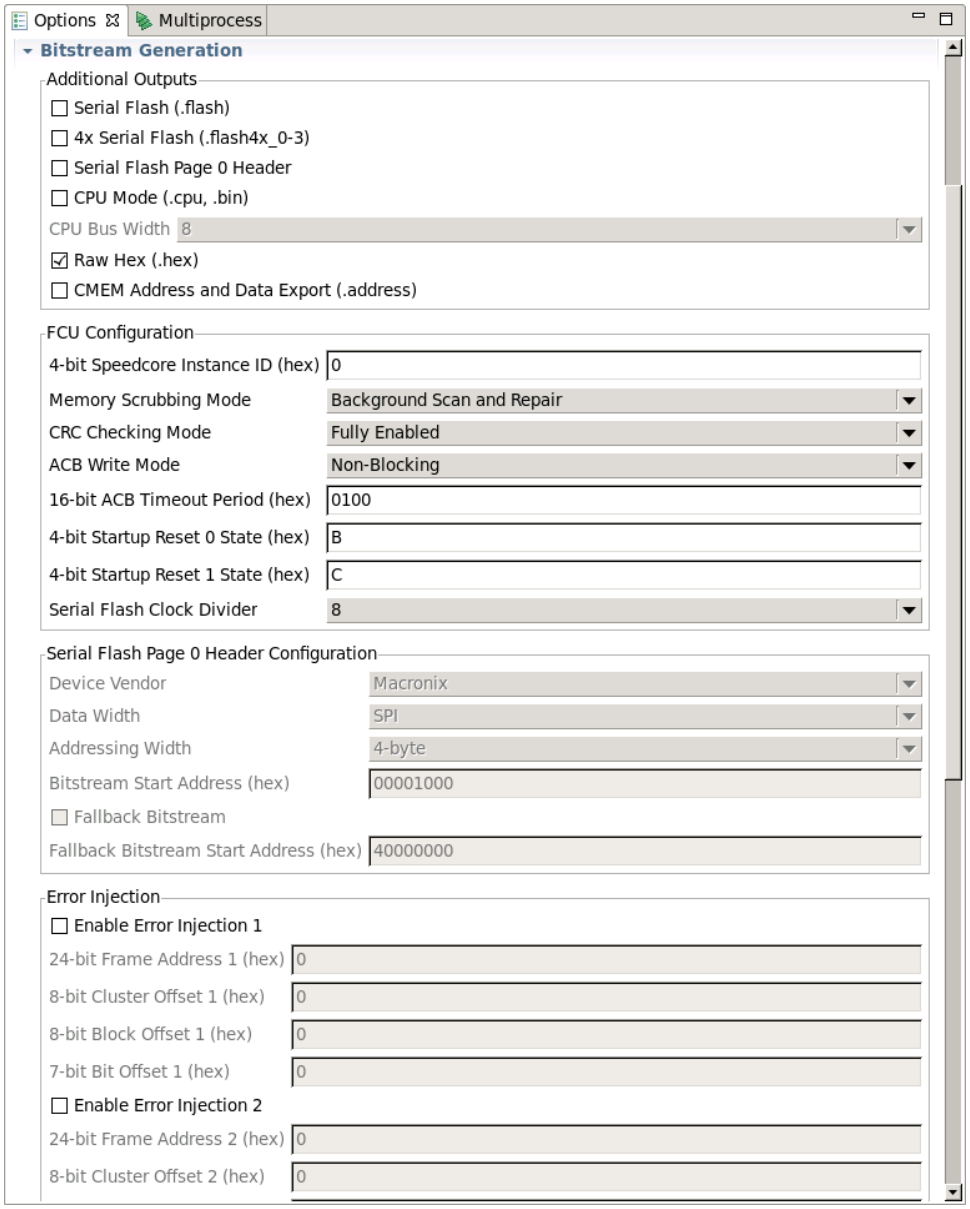

Figure 38: ***ACE Bitstream Generation Options Dialog***

Table 20: *Bitstream Generation Implementation Options - Additional Outputs*

Option	TCL Option	Description
Serial Flash (`.flash`)	`bitstream_output_flash`	This option enables the generation of an additional serial flash formatted output file. This file is named the same as the STAPL file, but with a `.flash` extension. The file contains a binary image that can be directly loaded into a single serial flash memory.
4x Serial Flash (`.flash4x_0-3`)	`bitstream_output_4xflash`	This option enables the generation of four additional 4x serial flash formatted output files. These files are named the same as the STAPL file, but with a `.flash4x_0` to `.flash4x_3` extension. Each file contains a binary image that can be directly loaded into each serial flash memory in a x4 configuration.
Serial Flash Page 0 Header	`bitstream_output_page0`	This option enables the generation of the flash Page0 header file which is a binary file.
CPU Mode (.cpu)	`bitstream_output_cpu`	This option enables the generation of an additional CPU Mode formatted output file. This file is named the same as the STAPL file, but with a `.cpu` extension. The file contains hexadecimal-formatted data organized with "CPU Bus Width" number of bits per file line. Data from this file is sent to the FCU CPU interface line by line (one line per clock cycle) from the top to the bottom of the file, where the left-most bit on each line is the MSB and the right-most bit is the LSB. In simulation, this file can be loaded using the readmemh function. For convenience, an additional binary representation of the CPU Mode output file is written, named the same as the STAPL file, but with a `_cpu.bin` extension. It contains the same data in the same bit order as the `.cpu` file.
CPU Bus Width	`bitstream_output_cpu_width`	This option controls the bit width of the CPU-mode formatted output file. When using the CPU interface in ×8 mode, set this value to 8. If using the CPU interface in ×128 mode, set this to 128. The value determines how many bitstream bits are printed per line in the `.cpu` output file. The bit sequence required by the FCU (and output in the generated bitstream file) might be different for each CPU Bus Width setting; therefore, it is important to set this option to match the actual CPU hardware interface width.
Raw Hex (.hex)	`bitstream_output_hex`	This option enables the generation of an additional Raw Hex formatted output file. This file is named the same as the STAPL file, but with a `.hex` extension. This file is used for debug purposes.
CMEM Address and Data Export (.address)	`bitstream_output_address`	This option can be used to enable an additional CMEM Address and Data Export output file. All addresses listed in this file are "used" in the bitstream. The data in this file can be compared against readback data. The file is named the same as the STAPL file, but with a `.address` extension.

Table 21: *Bitstream Generation Implementation Options - FCU Configuration*

Option	TCL Option	Description
Memory Scrubbing Mode	`bitstream_scrub_mode`	Selects the CMEM scrubbing mode. Allowed values include **Disabled** (`0`), **Background Scan** (`1`), and **Background Scan and Repair** (`2`).
CRC Checking Mode	`bitstream_crc_mode`	Selects the CRC checking mode. Allowed values include **Fully Enabled** (`0`), **Partially Enabled** (`1`), and **Bypassed** (`2`).
Serial Flash Clock Divider	`bitstream_sf_clock_div`	Selects the serial flash clock divider value.

Table 22: *Bitstream Generation Implementation Options - JTAG Scan Chain*

Option	TCL Option	Description
Single Device Chain	`bitstream_single_device`	Specifies whether the bitstream STAPL file is output for a single-device JTAG scan chain (the target device is the only device on the JTAG scan chain). Set this to **1** to indicate a single device. If this option is set to **0** (indicating multiple devices in the scan chain), then either the chain description file is used or the pre-IR, post-IR, and chain offset options are used to generate the bitstream STAPL file with knowledge of the scan chain.
Use JESD32 Chain Description File	`bitstream_use_chain_file`	When using a multi-device JTAG scan chain, specifies whether to use a JESD32 chain description file, or to use the explicit pre-IR, post-IR, and chain offset implementation options.
Chain Description File	`bitstream_chain_file`	Specifies the optional JESD32 chain description file used by the bitstream generator to automatically pad the JTAG IR chain for multi-device chains.
Chain Offset of Target	`bitstream_chain_offset`	Specifies the offset of the target device on the JTAG scan chain for multi-device chains. Setting this to **0** selects the first device on the chain, **1** selects the second device on the chain, and so on.
IR Bits Before Target	`bitstream_preir_padding`	Specifies the total number of Instruction Register bits on the JTAG scan chain prior to the target device Instruction Register. This option is used for multi-device scan chains in order to pad the IR chain properly with 1s to put other devices in bypass mode.
IR Bits After Target	`bitstream_postir_padding`	Specifies the total number of Instruction Register bits on the JTAG scan chain after the target device Instruction Register. This option is used for multi-device scan chains in order to pad the IR chain properly with 1s to put other devices in bypass mode.

Chapter - 8: Achronix Configuration Bus (ACB)

The ACB interface is provided to configure the interface cluster registers through the ACB pipe block. There are 4 ACB ports as indicated in the block diagram below namely P0, P1, P2 and P3. P0 and P1 are internal to Achronix, P2 and P3 are unused.

A diagram showing a high-level view of the ACB interface is provided below.

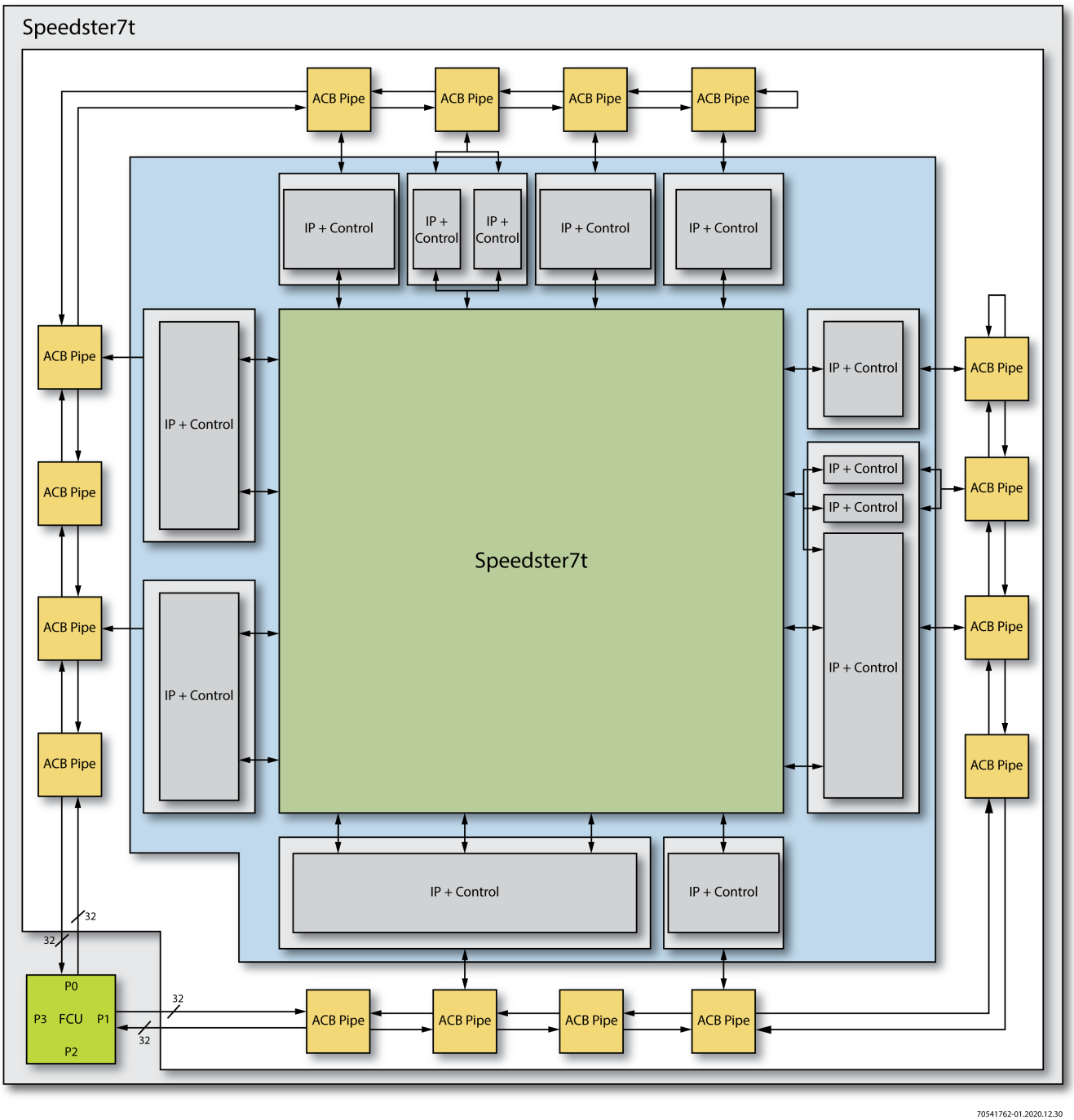

Figure 39: ***Achronix Configuration Bus Interface***

ACB Address Space

The ACB circuitry in the FCU is implemented using a 4-port scheme. For the Speedster7t FPGA, Ports 0 and 1 are used to interface with the internal boundary circuitry while ports 2 and 3 are not used. A 22-bit address is provided by the FCU to interface with the ACB structure plus 2 bits which are allocated to ACB chain select; namely P0, P1, P2 or P3. The 24-bit Internal Interface address is detailed in the following table.

Table 23: *ACB Internal Interface Address*

Bits	Description
23:22	Chain select, P0, P1, P2 or P3
21:16	Client ID
15:0	Client address

ACB Write and Read Protocols

ACB operations consist of writes and reads to the memory space in the IP+Control units inside the Speedster7t FPGA. These are split as follows:

Write to Memory

These operations come in two forms, non-blocking and blocking as described below:

- **Non-blocking** – The default mode of operation for writes. Allows writes to be pipelined without needing to wait for an acknowledgment or hready signal from the destination or client logic.
- **Blocking** – The non-blocking write mode assumes that clients are able to process write requests as they are received without any stalling. Since there is no mechanism for writes to exert back-pressure, the non-blocking write scheme will *not* work if clients cannot process the write requests as they arrive. In this case, the mode of operation for writes can be changed via an FCU register to blocking. In blocking mode, writes cannot be pipelined. Every write request that is issued requires an acknowledgement by returning an hready pulse before the request for the next write can be processed.
- **Timeout** – If the acknowledgement does not come within a predetermined time period (which could be due to many factors like illegal addressing or incorrect client IDs, etc.), the FCU times out. The period for timeout is configurable via an FCU register, and has a max value of 64k clocks (16-bits). An FCU status bit, which can be read back by the user, indicates whether or not the timeout actually happened. If and when the timeout occurs, the write request is flushed, and the FCU waits for or implements the next command.

> **Note**
>
> The timeout status bit is cleared when the next ACB operation is issued. There is no counter to indicate the number of timeouts that have occurred.

The mode is configured using a drop-down menu selection in the ACE GUI and only applies to ACB writes after bitstream programming has completed and the Speedster7t FPGA has transitioned to user mode. The ACB writes during bitstream loading are all performed in non-blocking mode. Based on the user selection, at the end of bitstream loading, and before entering user mode, there is an FCU command to update the mode for ACB writes. For additional clarity, the sequence of operations is as follows:

1. After power-up, the register controlling the ACB write mode is set to its default mode of operation, which is non-blocking mode.
2. All of the bitstream operations are performed in this mode.
3. At the end of the bitstream, just before the final CRC check, another FCU command is issued to set the desired mode (determined using the provided drop-down menu). The CRC check does not gate the change to ACB blocking mode. Moreover, the ACB blocking mode setting is retained even after the CRC check. The CRC check does gate the FCU from entering user mode (asserting `FCU_CONFIG_USER_MODE`) in the next step.
4. The Speedster7t FPGA is then taken through startup to enter user mode.

The two sub-sections below provide additional details on the non-blocking and blocking write protocols.

Non-Blocking Write

The figure below shows two back-to-back non-blocking writes on a port (specified by the `o_acb_hsel_P0/P1`). DATA1 and DATA2 are written to addresses ADDR1 and ADDR2 respectively with `o_acb_hwrite_P0/P1` asserted. After this, `o_acb_hsel_P0/P1` and `o_acb_hwrite_P0/P1` is de-asserted.

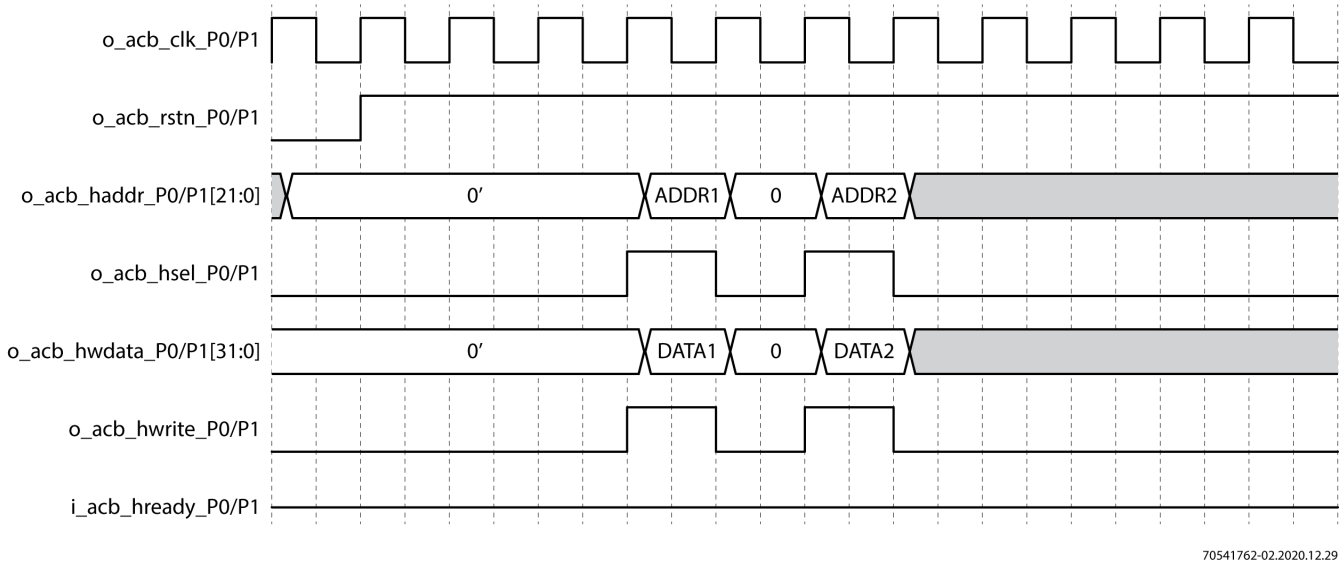

Figure 40: ***Non-blocking ACB Write***

Blocking Write

The waveform below shows a blocking write. This mode is not selected by default and must be enabled via an FCU register. With a blocking write, a write request is issued by asserting `o_acb_hwrite_P0/P1`. The required data, DATA1, is on `o_acb_hwdata_P0/P1[31:0]`, the address is specified by ADDR1 on `o_acb_haddr_P0/P1[21:0]`, and the request is going to port with `o_acb_hsel_P0/P1` set. The FCU then waits for a pulse on `i_acb_hready_P0/P1` some non-deterministic time later. After this pulse, the FCU can issue the next write request as shown below with value in DATA2 needing to be written to the address in ADDR2.

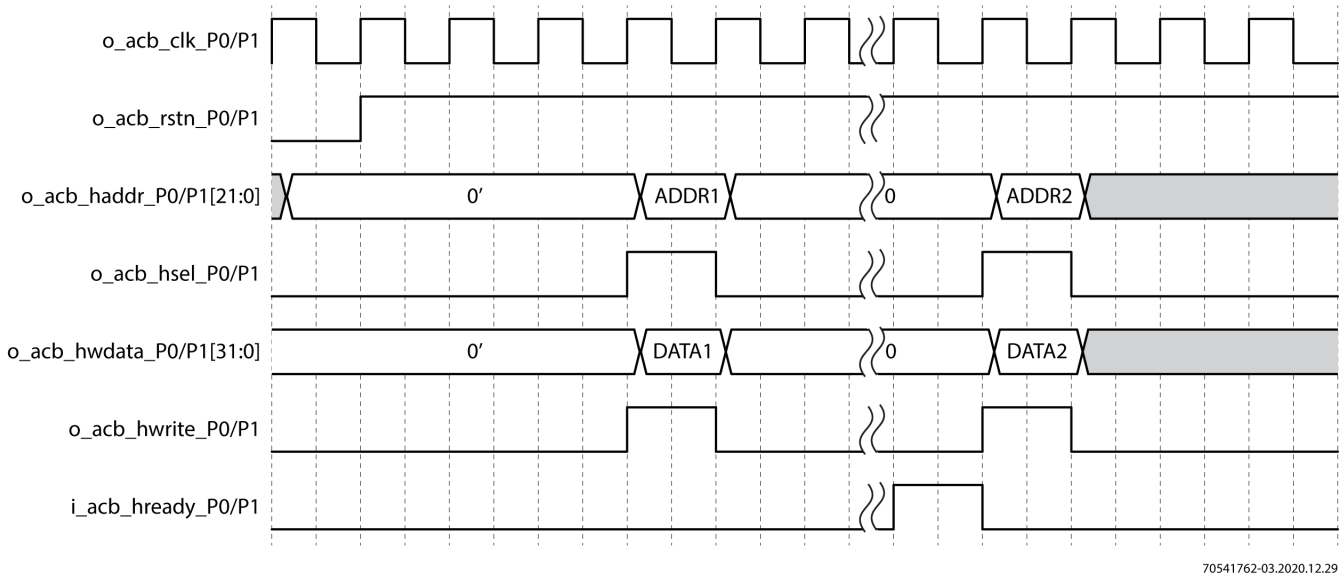

Figure 41: ***Blocking ACB Write***

Read from Memory

All reads are blocking. When a read request is issued, no additional requests can be issued until the read data and corresponding hready signal is returned, or the FCU times out due to the requested data not being returned in time. The period for timeout is configurable via an FCU register and has a maximum value of 64k clocks (16-bits). An FCU status bit, which can be read back by the user, indicates whether the timeout actually happened. If and when the timeout occurs, the read request is flushed and the FCU waits for or implements the next command.

While there are four independent ACB networks, they cannot operate independently of one another — all of the circuitry controlling and observing the four ports in the FCU is common. As a result, a port x operation followed by a port y operation are treated the same way regardless of whether x and y are the same or different ports.

The waveform below shows an ACB read operation which, as described above, can only be blocking. The FCU needs to wait for an acknowledgment in the form of an `o_acb_hready_P0/P1` pulse (or a timeout) to proceed with the next request. In this waveform, after reset de-assertion, a read request is made on a given port with `o_acb_hsel_P0/P1` set, setting `o_acb_haddr_P0/P1` to the desired location of ADDR1, and by keeping the `o_acb_hwrite_P0/P1` signal low. At some non-deterministic time later, read data is provided on the `i_acb_hrdata_P0/P1[31:0]` port with the pulse on `i_acb_hready_P0/P1` indicating when that data is valid.

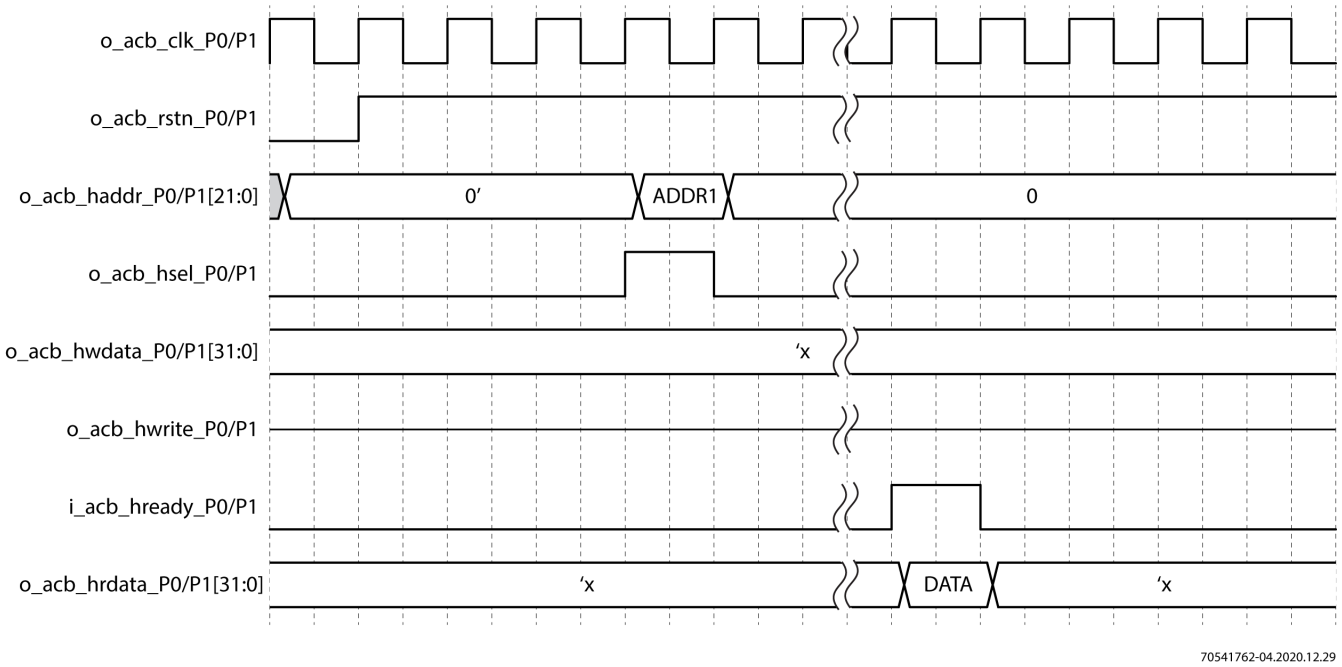

Figure 42: ***ACB Read***

Chapter - 9: Partial Reconfiguration

Partial reconfiguration enables the user to reprogram a part of the fabric with a smaller bitstream while leaving the remaining configuration intact. Each region that can be reconfigured independently is called a fabric cluster or just cluster. The Speedster7t FPGA has 80 clusters which can be reconfigured independently. Partial reconfiguration can only be initiated after the device has entered user-mode.

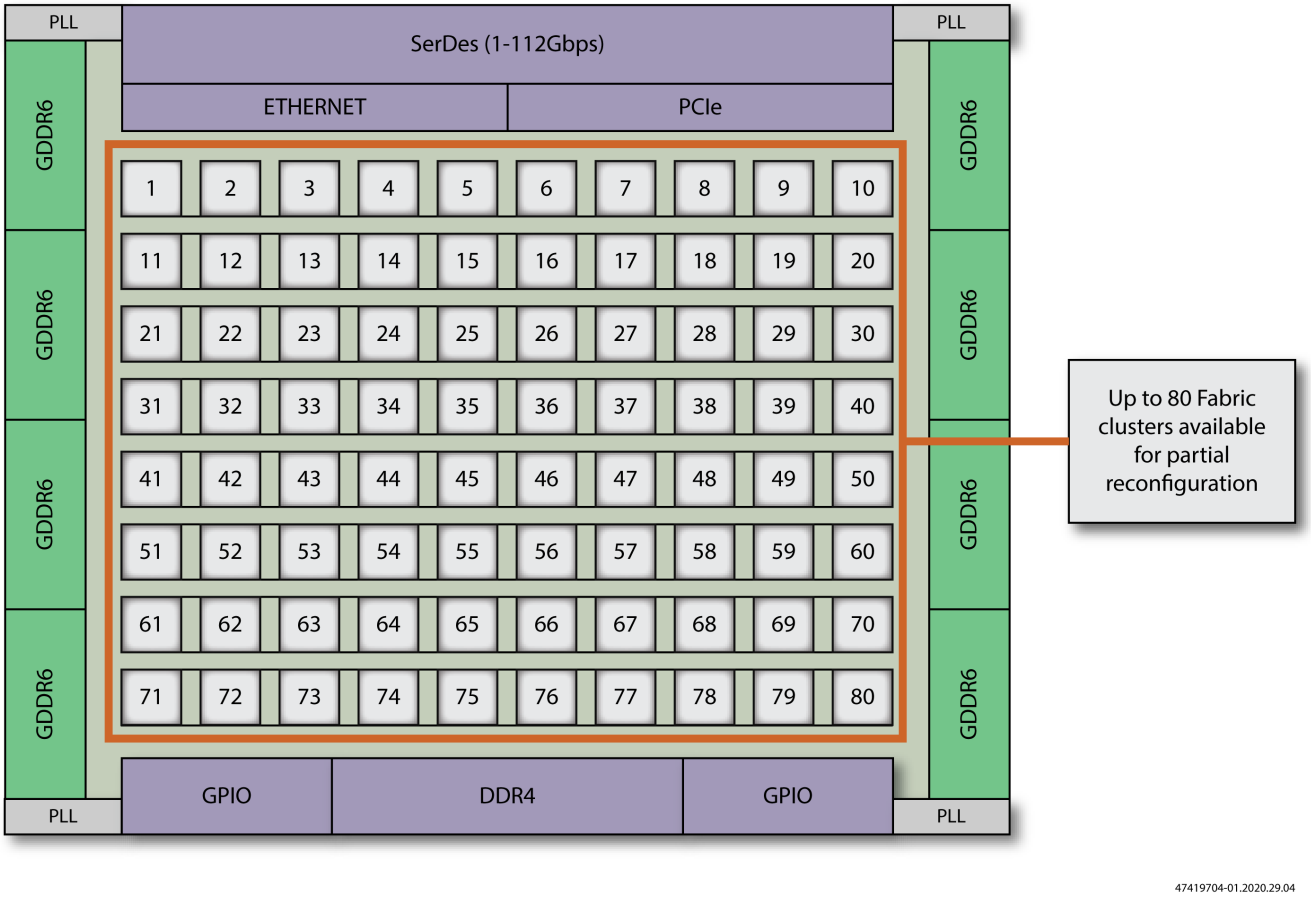

Figure 43: ***Speedster7t FPGA Partial Reconfiguration Fabric Cluster Layout***

There are many advantages to partial-reconfiguration:

- Enable dynamic functions for certain blocks in the design
- Smaller FPGA logic functions can be programmed on the FPGA when needed
- Faster programming times

Design Considerations

Partial-reconfiguration introduces additional complexity in the design. Defining correct functional hierarchy is very important for designs that use partially reconfigurable modules. It is important to ensure that there are no functional issues when the target module is being partially reconfigured. No outputs driven from that module can

be actively used during partial-bitstream programming since the remaining FPGA fabric is alive and performing regular tasks during partial reconfiguration.

Reconfigured regions cannot have any clock configurations that affect elements outside of the re-configured region. For this reason, a re-configured cluster cannot change where clocks are routed down the branch but it can choose how those existing clocks are connected inside the cluster.

Timing paths into and out of the module might change after partial reconfiguration. It is important to ensure that there are no timing violations after partial-reconfiguration for a design that met timing earlier. A good practice is to use the most challenging module for initial timing closure and, ideally, register all inputs and outputs.

Also, port definitions for the the new module and the module being swapped out must be the same. The reset scheme for the target module should be correctly defined and understood.

It is also important to define the correct placement constraints so that the target module is completely contained within the cluster marked for partial-reconfiguration. The resources for the module cannot exceed the available resources for a cluster and optimizations across the cluster should be disabled.

Partial Reconfiguration Steps

Partial re-configuration can be entered when in user mode as described below. To enable partial re-configuration, complete these steps:

1. Write a value of 32'h1000_0000 to CONFIG_REG_CRC and a value of 32'h0000_0000 to the CONFIG_REG_CRC2 register. This brings the partial state machine to shutdown state and asserts the partial reset.
2. Send SYNC, JTAG ID and the preamble header, then program the selected clusters using the partial bitsream.
3. Write a value of 32'h0200_0000 to CONFIG_REG_CRC and a value of 32'h0000_0000 to CONFIG_REG_CRC2 register. This releases the reset to the partial clusters and generates partial_config_done.

Chapter - 10: Remote Update

Introduction

The remote Update feature in Speedster7t FPGAs implements device reconfiguration using dedicated remote system upgrade logic in the FCU eliminating the need for expensive custom external logic. The ability to upgrade an image remotely in an FPGA deployed in the field allows delivery of feature enhancements and bug fixes without recalling a product, reduces time-to-market and extends product life.

The Remote Update logic within the FCU commands the configuration module to start a reconfiguration cycle. Error detection is enabled during and after the configuration process. If any errors are detected, the logic facilitates system recovery by reverting back to a safe, default factory configuration image and then provides error status information.

Implementation

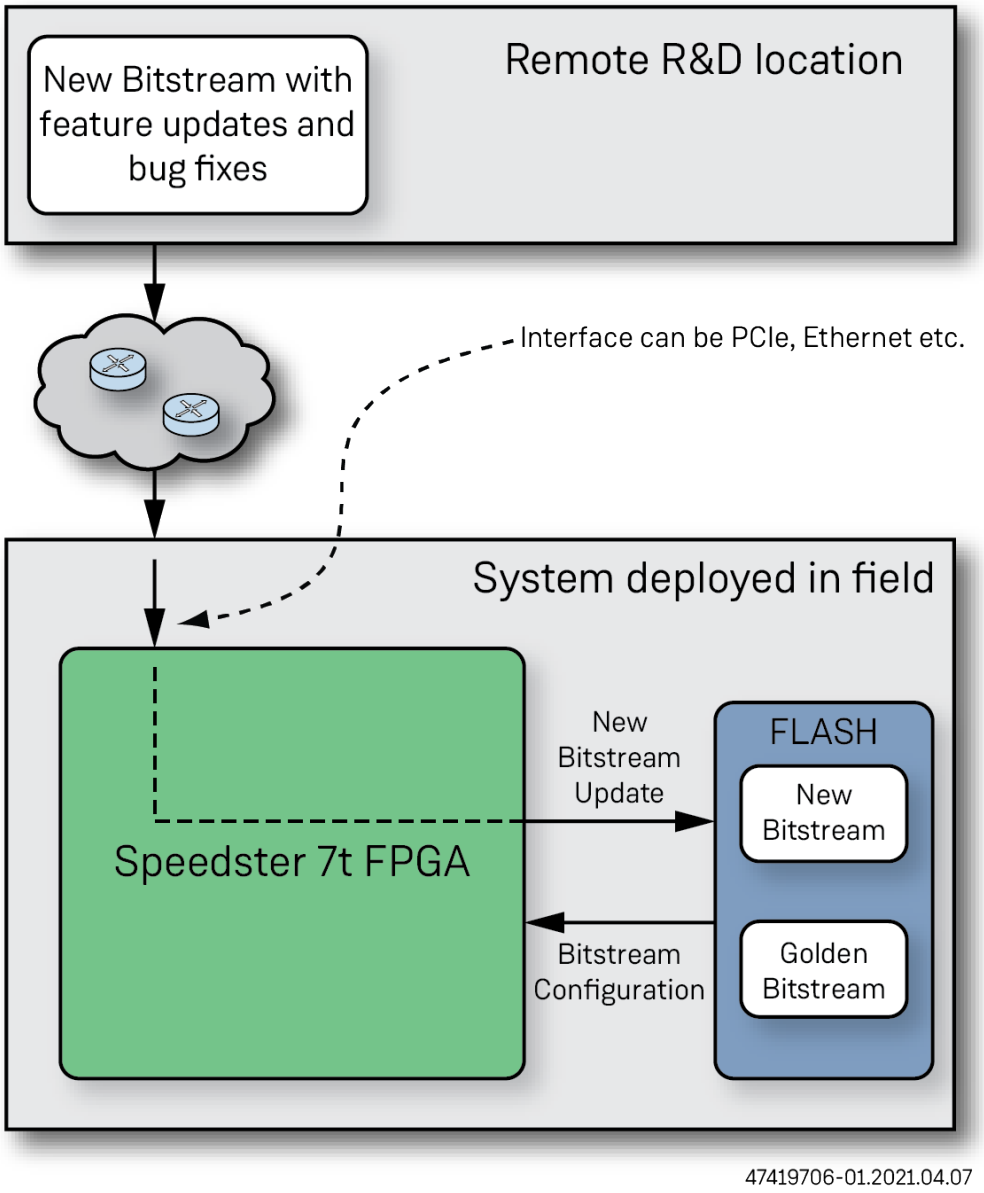

Figure 44: ***Speedster7t FPGA Remote Update Feature***

Remote update follows this procedure:

1. The flash device holds two bitstreams:
    a. Known good working image, "Golden bitstream".
    b. New bitstream with enhanced features and/or bug fixes, "New Bitstream".
2. Initially the device boots from the golden bitstream and enters user mode.
3. System software initializes the "Current Bitstream Address" register, which is the start address of the New Bitstream programmed by the user.
4. System software initializes the "Golden Bitstream Address" register, which is the start address of the Golden Bitstream programmed by the user.
5. Based on configuration modes, software writes the command and start address into the flash configuration header.
6. The system initiates a reset, the FPGA re-configures from the Current Bitstream Address and reads the first 512 bits of the New Bitstream data from the flash device and enters into wait state.
7. If encryption is not enabled, the complete bitstream is read and the FPGA is re-configured.
8. If encryption is enabled and the eFuse key is ready:
    a. The header segment0 data is read and sent to the secure boot core.
    b. The flash read state machine enters into a wait state of 2.6 ms.
    c. The flash interface reads the complete bitstream and configures the FPGA.

Fallback on Error

In the event of a bitstream load failure, The fallback procedure is invoked:

1. After bitstream load, failure can occur in two scenarios:
    a. No IDCODE match after timeout expires.
    b. CRC error after timeout expires.
2. If these checks fail, a retry is attempted N times (the number of retries is described in the flash configuration header).
3. If the failures persist and the system is unable to boot from the New Bitstream and fallback is present in the FPGA configuration header, a fast read is issued from the header fallback address.
4. The user should then update the New Bitstream or point the default boot address to the Golden Bitstream.

Chapter - 11: Design Security for Speedster 7t FPGA

Achronix recognizes the importance of protecting the sensitive IP a customer downloads onto their FPGA. To provide a high level of protection, Speedster7t FPGAs have a number of features to support bitstream encryption as well as authentication. These features ensure that no one can access the design configuration on the FPGA and also ensures that the design is the intended design. Speedster7t FPGAs provide this high level of security through the following features:

- Support for ECDSA authenticated and AES-GCM encrypted bitstream
- Dynamic power analysis (DPA) protection to prevent side-channel attacks
- Physically unclonable function (PUF) for tamper-proof protection
- Securely stores both public and encrypted private keys

With this security solution deployed, a customer's design is secure. Even with possession of the device, no one can extract the underlying design, the design cannot be reverse engineered, nor can the design be altered in any way.

Bitstream Authentication

Authentication of a bitstream ensures that the design on the device is the intended design. Achronix provides a two-step authentication process that first authenticates an encrypted bitstream before decrypting it, and then performs authentication a second time on the decrypted bitstream before configuring the device:

1. A bitstream is encrypted using AES-GCM, which provides authenticated encryption.
2. The user provides an asymmetric private key to sign the encrypted bitstream using ECDSA.
3. When an encrypted and signed bitstream is loaded into the FPGA, the device uses the public key stored in an electronic fuse (eFuse) on the device to authenticate the bitstream using the public key.
4. When authenticated, the bitstream decryption is enabled, and the bitstream is authenticated a second time while decrypting with AES-GCM.
5. After the second authentication, the bitstream is used to configure the FPGA.

Bitstream Encryption

Bitstreams consist of sensitive intellectual property of the designer. Achronix provides tools to generate bitstreams that are encrypted and signed using very strong encryption with hardware designed to be resilient to side-channel attacks, such as dynamic power analysis (DPA). Additionally, the key derivation function (KDF) inside the secure boot portion of the FPGA, along with the physically unclonable function (PUF) ensure protection of the secret keys to decode and authenticate the bitstreams. Together these systems provide a solution that is safe from attacks such that even with possession of the device, an adversary cannot extract the underlying design, cannot change the system to perform another task other than the intended task, and cannot reverse engineer the core intellectual property.

The following figure shows an overview of the security system and how elements work together to protect the bitstream. Blocks shown in yellow represent encryption/decryption elements. Blocks shown in blue are authentication elements and green blocks handle authenticated and encrypted bitstreams.

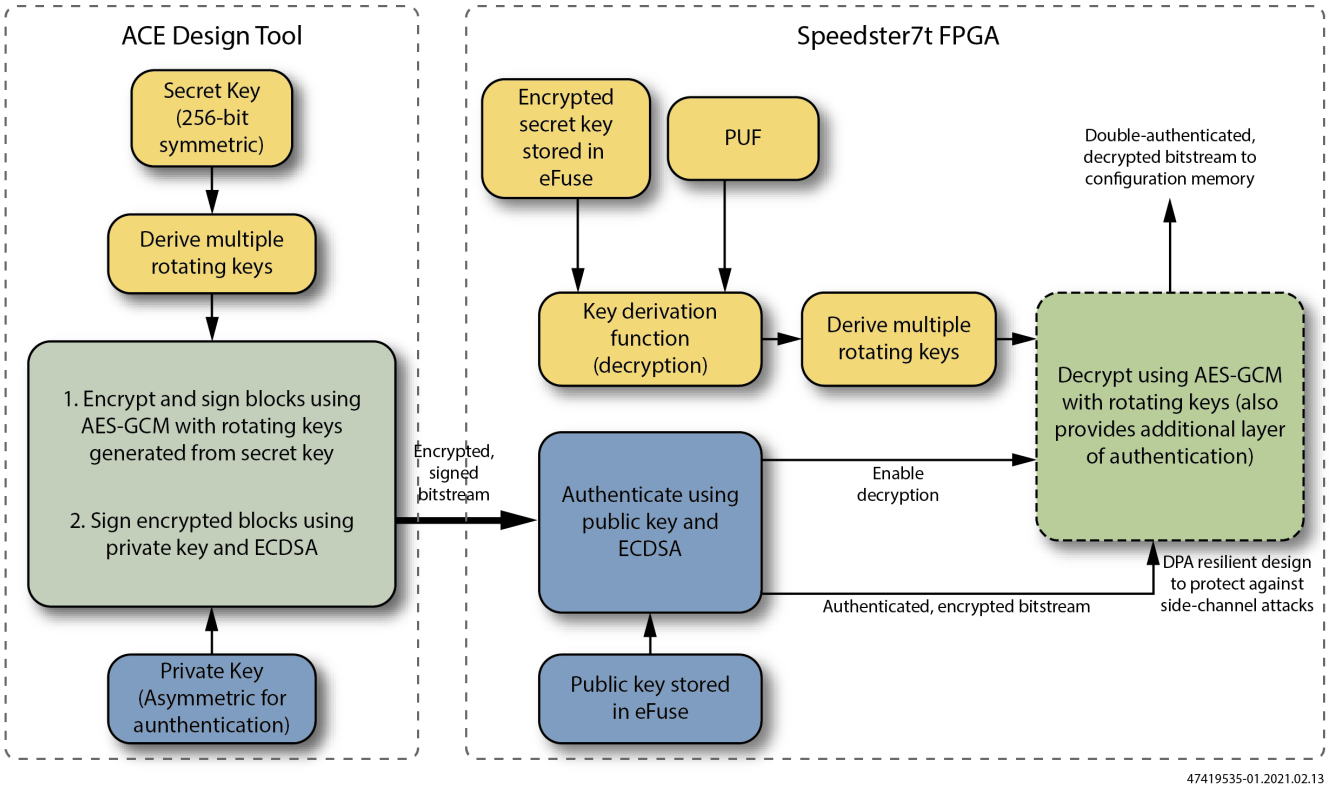

Figure 45: ***Bitstream Encryption/Authentication Block Diagram***

Generating Encrypted Bitstreams

To generate an encrypted bitstream, the user provides a 256-bit secret key to ACE. In order to provide better protection against side-channel attacks, ACE does not simply use this secret key to encrypt the entire bitstream. Instead, the secret key is used as an initial key. ACE then generates new derived keys based on the initial secret key to encrypt smaller segments of the bitstream, each with a different derived key and a new nonce. Here the nonce, also known as an initialization vector (IV), is a random number only used once per segment such that the same pattern is not generated while replaying or encrypting the same bitstream. Bitstream encryption is performed using the highly secure 256-bit AES-GCM encryption standard. Galois/counter mode (GCM) is an advanced form of symmetric-key block encryption which enhances the 256-bit advanced encryption standard (AES) by using a nonce (one-time use random value) and a counter mode so that each segment of data is uniquely encrypted. ACE also uses a Galois message authentication code (GMAC) to simultaneously sign and authenticate the data, including the unencrypted preamble section of the bitstream to guarantee the bitstream has not been altered. To further protect the bitstream, ACE also signs each segment of the encrypted bitstream using ECDSA. See the section on Bitstream Authentication (see page 70) above for more details on the ECDSA authentication.

Hardware Security

There are several security features available in the hardware to support decryption of encrypted bitstreams, safe storage of secret keys, and strict rule enforcement which locks the device if security rules are violated. The main features for decryption and safe storage of keys use the physically unclonable function (PUF) which provides a unique secret value per individual chip, and the key derivation function (KDF) which uses the PUF as the key to encrypt/decrypt the real secret keys from the encrypted keys that are stored in an electronic fuse (eFuse).

Physically Unclonable Function

The PUF generates a unique secret identifier for each individual chip. It is created from random physical variations that occur during the semiconductor manufacturing process, such that the same circuit on a device creates completely different and unique values on each chip, even chips on the same wafer. The value of the PUF is random per individual chip, but remains constant over the lifetime of that chip. The PUF value is not known to Achronix or the manufacturer, and the value cannot be observed without destroying or altering the value of the PUF. This PUF value can be used to encrypt the user's secret key and store an encrypted version of the secret key in an eFuse. Then when an encrypted bitstream is loaded into the FPGA, the PUF value is used to temporarily decrypt the stored encrypted secret key. This secret key is then used to generate the multiple rotating keys to decrypt the bitstream blocks that configure the FPGA.

Key Derivation Function

The KDF uses 256-bit AES encryption in conjunction with the PUF to create an encrypted version of the user's secret key that can be stored in an eFuse. While it is theoretically possible to observe the contents of the eFuse if an adversary is in possession of the device and has access to advanced reverse engineering equipment, the stored key is an encrypted version of the secret key that uses the PUF value as the master key for encryption. Again, the PUF value cannot be known and is unique to each individual device, thus making the stored key safe. Additionally, when the KDF needs to decrypt an encrypted bitstream, it loads the encrypted key from the eFuse along with the PUF value and temporarily decrypts the secret key. The secret key is then used as the initial key for the module that generates the multiple derived keys for AES-GCM decryption of the bitstream prior to loading it into the configuration memory in the FPGA.

The following two figures show how the PUF and KDF are used to generate a secure encrypted key to store in an eFuse, and how they are used to recreate the secret key to decrypt the bitstream.

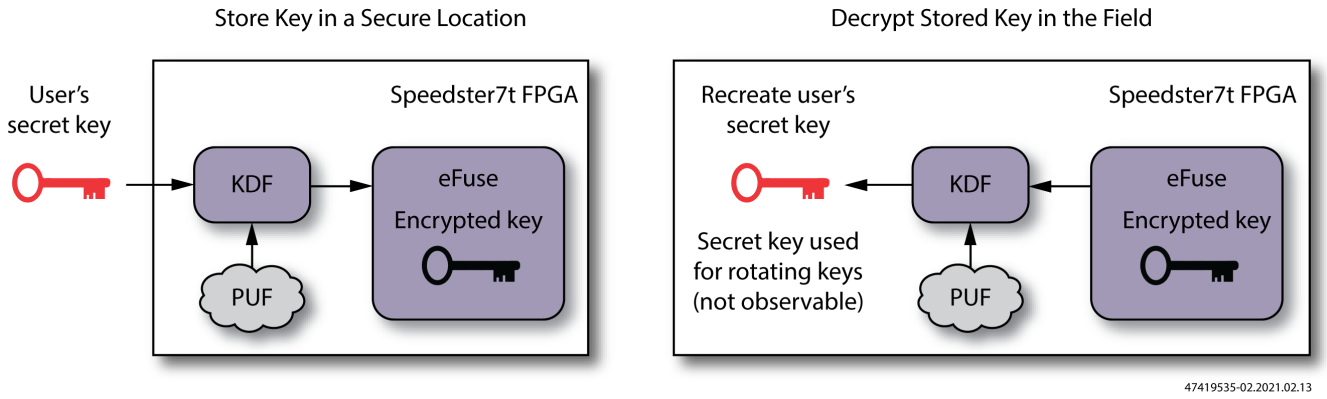

Figure 46: *Safe Secret Key Storage*

Rules for Encryption

When using encrypted bitstreams, the FPGA enforces a set of rules. If the security rules are violated, the FPGA locks up and cannot be used in any way without powering down the device. First, there is an ordering rule determining how bitstreams are to be loaded. Speedster7t FPGA bitstreams have three phases and must follow these ordering rules:

1. Zero, one, or multiple pre-configuration bitstreams.
2. One, and only one, full configuration bitstream.
3. Zero, one, or multiple partial reconfiguration bitstreams.

Additionally, there are rules to determine which keys can be used for the encryption. The eFuses can store up to four secret keys — bitstreams can be encrypted using up to four different initial keys. These rules must be followed to prevent locking the device:

1. If the `encrypted_bitstreams_only` eFuse bit has been set for the FPGA, the device only accepts encrypted bitstreams.
2. If any pre-configuration bitstream is encrypted, all pre-configuration bitstreams must be encrypted using the same key.
3. If either the pre-configuration bitstream or the full bitstream are encrypted, they both must be encrypted and both must use the same key.
4. Any partial reconfiguration bitstreams may use a different key if and only if the previous bitstream sets the `same_key` bit to 0 in the preamble, and the partial reconfiguration bitstream also sets that same bit to 0 in its preamble.

Note

It is acceptable to load an unencrypted bitstream after a previous encrypted bitstream. It is *not* acceptable to load an encrypted bitstream after a previous unencrypted bitstream.

Security Fuses

There are several eFuses that are related to the security features in Speedster7t FPGAs. Some of these are set during manufacturing and cannot be changed by the customer, and others are available for customer use. See the eFuse (see page 75) chapter for details.

Fuses Set at Manufacturing

There are two fuses that can be set at manufacturing time to limit the features of the FPGA (The part number of the device indicates if these limitations exist in a part):

- **Bitstream decrypt disable** – If set, the FPGA cannot accept encrypted bitstreams.
- **DPA disable for bitstream decrypt** – If set, the FPGA still supports encrypted bitstreams, but there is limited hardware protection for differential power analysis (DPA) side-channel attacks that can potentially expose secret keys.

Fuses Set By Customer

There are several eFuses that can be set by the customer if using encrypted bitstreams:

- **Bitstream authentication key** – This fuse contains a 768-bit hash of the public key used for first-level authentication of encrypted bitstreams. This fuse is not readable.
- **Bitstream decryption key** – These fuses contain the four 256-bit secret keys that can be used for decryption and authentication of encrypted bitstreams. These fuses can contain the actual secret keys or the encrypted version of the secret keys (using PUF and KDF). These fuses are not readable.
- **Bitstream user register** – This fuse contains the 32-bit value set by the user to identify the key version used. The secret key itself cannot be read back, but the user register value can be read. The user keeps a mapping of key versions to keys.
- **Bitstream user lock** – This one-bit fuse, if set, disables further updates to the authentication key, decryption key, and user register.

- **Encrypted bitstreams only** – This one-bit fuse, if set, forces the FPGA to only accept encrypted bitstreams that use one of the keys in the fuses.

Default Keys

Achronix provides a default public key for authentication and a default secret key for encryption/decryption of the bitstream. These keys are available for testing, so that a user has confidence the security system works. The default keys should not be used to protect sensitive designs — they are only made available for testing purposes. Additionally, when a user sets the eFuse to accept encrypted bitstreams only, the FPGA no longer accepts the default keys.

Loading Encrypted Bitstreams

Loading an encrypted bitstream is similar to loading an unencrypted bitstream. However, the most important difference is that when the unencrypted 512-bit preamble of the bitstream is loaded, the FPGA disables all data read-out, thus securing the device containing a user's sensitive IP and protecting it from being known, reverse engineered, or altered in any way. Encrypted bitstreams are loaded following these steps:

1. When the hardware detects the loading of an encrypted bitstream, all readout and debug features are disabled, preventing the reading of any internal state related to the FPGA fabric or the FCU.
2. Security rules for loading encrypted bitstreams are checked. If the rule checker fails, the FPGA enters a locked state and can only be re-enabled with a power cycle.

If a board management controller is used to load in the bitstreams, there are additional requirements to be aware of:

1. After the 512-bit preamble of the bitstream, the board management controller must pause and wait for 300 clock cycles before sending the next portion of the encrypted bitstream.
2. After the first 12,688 bytes of the encrypted bitstream, the board management controller must pause and wait at least 520,000 FCU clocks, or about 2 ms (assuming a 32-bit data path and 250 MHz FCU clock).
3. For encrypted bitstreams, a board management controller is limited to sending 32-bits per FCU clock. For unencrypted bitstreams, the controller can send data at a rate up to 128-bits per FCU clock.

Note

When using encrypted bitstreams, it is *not* possible to use any debug features of the FPGA. Debug features are *only* available when using unencrypted bitstreams.

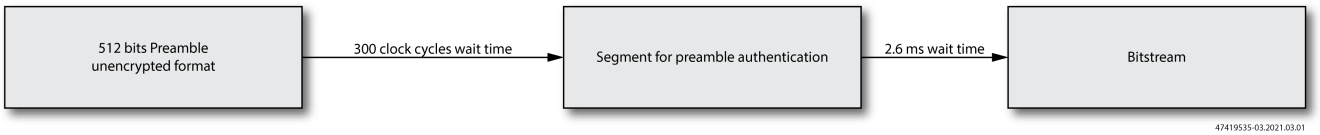

Figure 47: ***Encrypted bitstream loading sequence***

Chapter - 12: eFuse

The eFuse blocks hold:

- Four 256-bit bitstream decryption keys
- One 768-bit public key for secure boot authentication
- BRAM redundancy bits
- Other control information

Additionally, the blocks hold the default redkey and public key. eFuse registers are mapped within the FCU registers space and the configuration block has a built-in eFuse interface that generates the required signals for eFuse reads and writes.

An eFuse read can be performed after POR, where the configuration block triggers the eFuse read operation (refer to Read Non-Volatile Memories (see page 50)). The eFuse interface has clock dividers which generate the required clock using a default clock divider value to meet the minimum time requirements. The maximum clock considered here is 250 MHz.

The eFuse can be written through the FCU registers by loading a bitstream with the JTAG/PCIe/CPU interface. The write is performed using an unencrypted bitstream. During eFuse write, the clock divider register can be reconfigured by the user based on the clock source in order to generate the eFuse clock to meet the min/max requirements.

Table 24: *eFuse Registers*

Register Name	Address	Description
CONFIG_REG_ADDR_EFUSE_VERSION_ID	16’h1240	KEY Version.
CONFIG_REG_ADDR_EFUSE_UNIQUE_DEVID1	16’h1244	96 bit unique device ID.
CONFIG_REG_ADDR_EFUSE_UNIQUE_DEVID2	16’h1248	
CONFIG_REG_ADDR_EFUSE_UNIQUE_DEVID3	16’h124c	
CONFIG_REG_EFUSE_CNTRL	16'h107C	eFuse control register.
CONFIG_REG_ADDR_EFUSE_CLK_DIV_ADDR	16’h12c0	eFuse clock divider.
CONFIG_REG_EFUSE_KEY_ADDR	16'h1080	eFuse key write address.
CONFIG_REG_EFUSE_KEY_DATA0	16'h1084	256 bit key data to eFuse.
CONFIG_REG_EFUSE_KEY_DATA1	16'h1088	
CONFIG_REG_EFUSE_KEY_DATA2	16'h108C	
CONFIG_REG_EFUSE_KEY_DATA3	16'h1090	
CONFIG_REG_EFUSE_KEY_DATA4	16'h1094	

Register Name	Address	Description
CONFIG_REG_EFUSE_KEY_DATA5	16'h1098	
CONFIG_REG_EFUSE_KEY_DATA6	16'h109C	
CONFIG_REG_EFUSE_KEY_DATA7	16'h10A0	
CONFIG_REG_EFUSE_BRAM_ADDR	16'h10A4	eFuse bram/control write address
CONFIG_REG_EFUSE_BRAM_DATA	16'h10A8	eFuse BRAM red data/control data

Table 25: *eFuse Control Register*

Register Name	Bit position	Type	Reset value	Description
CONFIG_REG_BIT_EFUSE_CNTRL_PUF_EN	0	RW	‘h0	PUF enable, which enables the key encryption and decryption
CONFIG_REG_BIT_EFUSE_CNTRL_KEYWR_EN	1	RW	‘h0	Key write start

Table 26: *eFuse Clock Divider Register*

Register Name	Bit position	Type	Reset value	Description
CONFIG_REG_BIT_EFUSE_CLK_DIV	31:0	RW	‘d100	eFuse clock divider

Table 27: *eFuse Key Write Address*

Register Name	Bit position	Type	Reset value	Description
CONFIG_REG_BIT_EFUSE_KEY_ADDR	10:0	RW	‘h0	• eFuse physical write address for Keys • Each key needs 16 eFuse locations to store the information • Start address for each key is 0, 16, 32, 48

Revision History

Version	Date	Description
1.0	20 Apr 2021	Initial release.
1.0.1	26 Apr 2021	Change images Read from FCU Register (see page 41) and Read from ACB Register (see page 41) to execution order.