Xilinx MultiBoot module

October 29, 2013

Theodor-Adrian Stana (CERN/BE-CO-HT)

Revision history

Date	Version	Change
28-10-2013	0.1	First draft

Contents

List of Figures

List of Tables

List of Abbreviations

FPGA	Field-Programmable Gate Array
IPROG	Internal PROGRAM_B
OHWR	Open Hardware Repository
PROM	Programmable Read-Only Memory
SPI	Serial Peripheral Interface

1 Introduction

Xilinx MultiBoot technology [1] allows reprogramming an FPGA by downloading a bitstream to a PROM chip external to the FPGA and then issuing an IPROG (Internal PROGRAM_B) command to the configuration logic of the FPGA. This command triggers deletion of the FPGA configuration and rewriting it with the new configuration written to the PROM.

This document describes *xil_multiboot*, an Open Hardware (OHWR) [2] FPGA design that can be used to remotely reprogram a Xilinx Spartan-6 FPGA using MultiBoot technology.

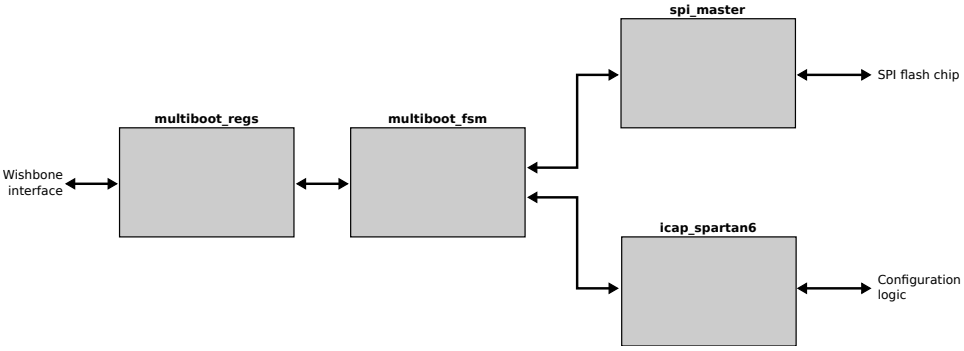

Figure 1: Block diagram of *xil_multiboot* module

The main features of the *xil_multiboot* module are:

- software controls operation of the module
  - Wishbone interface implements registers for control and status readout
  - writing FPGA bitstream data to the flash chip
  - issuing reprogramming command to the FPGA (via Xilinx ICAP)
  - reading boot status register from the FPGA configuration logic (via Xilinx ICAP)
- a finite-state machine (FSM) controls writing to the flash chip and sending the reprogramming command to the FPGA
- modular, easily modifiable design
  - PROM chip is controlled by software, so virtually any 8-bit SPI PROM chip is supported by writing software to send the various commands to the chip
  - Wishbone interface can easily be replaced by some other interconnect (e.g., AXI)

A block diagram of the *xil_multiboot* module is shown in Figure 1. Users can write bitstream data to a flash by writing each byte of the bitstream to a module register (in the *multiboot_regs*) module. Writing to the flash is done via the *spi_master* module under the control of the finite-state machine (FSM) module (*multiboot_fsm*). After the bitstream has been written to the flash module, a remote reprogramming command is sent to the configuration logic by setting a bit in one of the control registers. The Xilinx *ICAP_SPARTAN6* primitive is the interface between the *xil_multiboot* module and the FPGA configuration logic.

For the rest of the document, the external PROM chip that the FPGA uses to program itself will be referred to as flash, since a flash chip was used as the external PROM during the design of the module. However, this does not mean *xil_multiboot* works only with flash chips. Any 8-bit SPI PROM should be usable with the *xil_multiboot* design.

2 Instantiation

Table 1 lists the ports of the *xil_multiboot* module. In order to instantiate the MultiBoot module, one needs to connect the Wishbone slave ports to a Wishbone master, such as the *xwb_crossbar* Wishbone crossbar module on OHWR [3]. The SPI ports should be connected directly to the FPGA output ports connected to the flash chip.

Table 1: Ports of the Xilinx MultiBoot module

Port	Size	Description
clk_i	1	Clock input
rst_n_i	1	Active-low reset input
wbs_i		Wishbone slave interface inputs
wbs_o		Wishbone slave interface outputs
spi_cs_n_o	1	Active-low chip select output
spi_sclk_o	1	SPI clock output
spi_mosi_o	1	SPI data output line (Master Out, Serial In)
spi_miso_i	1	SPI data input line (Master In, Serial Out)

Note that in order to use the module, the OHWR *general-cores* library [3] needs to be imported into the design. This library is where the structures for the Wishbone slave interface ports (*wbs_i* and *wbs_o*) are defined.

3 Using the Xilinx MultiBoot module

For an example project of where the Xilinx MultiBoot module is used, see the CONV-TTL-BLO project [4]. The firmware of this project, starting from version 2.0, instantiates the MultiBoot module. The project also contains example Python scripts for writing a bitstream to the flash. Refer to the project webpage [4] for more information.

Table 2 shows the MultiBoot workflow [5]. See [6] for pointers on how to generate a MultiBoot bitstream. The address map of the MultiBoot module can be found in Appendix A. This appendix details the various registers the user should write as part of the MultiBoot workflow.

Table 2: MultiBoot workflow

Step	Action
1	Prepare a Xilinx FPGA bitstream
2	Send the bitstream to the flash by writing to the FAR register
3	Write the MultiBoot bitstream start address and flash chip read command op-code into the MBBAR register
4	Write the Golden bitstream start address and flash chip read command op-code into the GBBAR register
5	Unlock the IPROG bit in the FPGA by setting CR.IPROG_UNL
6	Issue a reprogramming command to the FPGA by setting CR.IPROG

4 Xilinx MultiBoot technology

Spartan-6 configuration logic is organized in a set of frames, which can be written to or read from using 16-bit word transfers. A Spartan-6 FPGA bitstream consists of commands that access the FPGA configuration logic to read and write these frames, as well as the data for the configuration logic frames.

When using the MultiBoot technology, multiple bitstreams exist for one FPGA. These bitstreams are all stored on the attached flash chip and the user can send a special instruction called IPROG (Internal PROGRAM_B) to reprogram the FPGA chip using one of the bitstreams on the flash.

Most MultiBoot designs will contain at least three bitstreams, as shown in Figure 4. When the FPGA board is powered on, if it is configured in master mode configuration [1], the FPGA will start loading a bitstream from address zero of the attached flash chip. This is where the Header

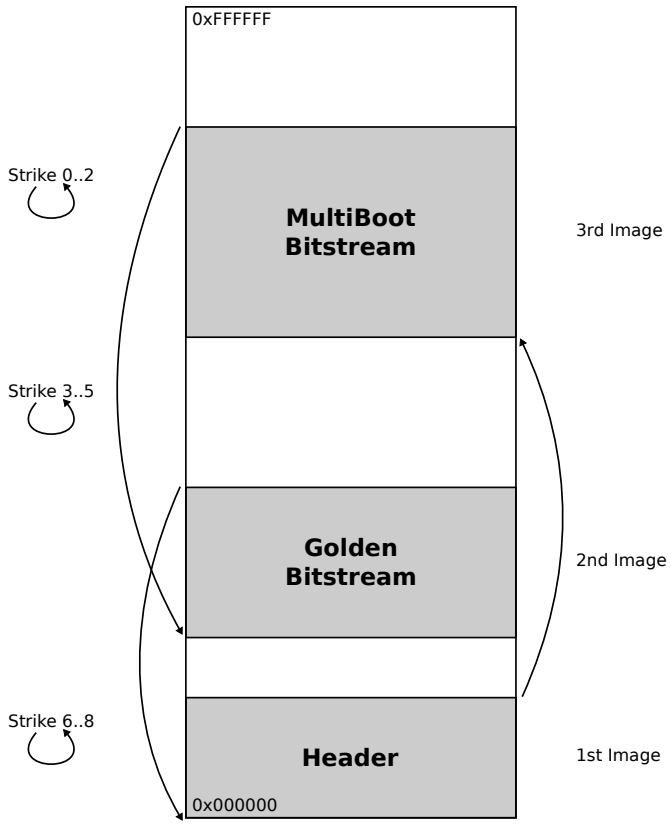

Figure 2: Bitstreams in a MultiBoot design

bitstream resides. This small bitstream contains a synchronization word for the configuration logic, sets the start address of the MultiBoot and Golden bitstreams and sends an IPROG command to the configuration logic.

The IPROG command causes the configuration logic to start loading the MultiBoot bitstream from the flash chip, starting at the given MultiBoot address. If the MultiBoot bitstream fails to load three times (see below for bitstream load failure reasons), the configuration logic falls back to the Golden bitstream. This is a bitstream which is known to be safe, should the MultiBoot bitstream be corrupted on load.

Should the Golden bitstream also be corrupted, the configuration logic tries to load it three times and then returns to the Header bitstream. When here, the configuration logic attempts to load the MultiBoot and Golden bitstreams three more times, before configuration failure.

A strike counter is used to select between which of the three bitstreams is loaded. This strike counter is stored in the configuration BOOTSTS register and can only be reset by a power-on reset or by pulsing the FPGA's PROGRAM_B pin. The strike counter is shared among the three bitsreams; as Figure 4 shows and as just described, each bitstream is selected based on

the value of this counter:

- if it is 0..2, the MultiBoot bitstream gets loaded
- if it is 3..5, the Golden bitstream gets loaded
- if it is 6..8, the Header bitstream gets loaded, and MultiBoot and Golden bitstreams are attempted three more times
- if it is 9, configuration is halted

4.1 Reasons for bitstream load failure

There are two ways in which a bitstream load can fail:

- synchronization word Watchdog timeout
- bitstream CRC error

Prior to performing any work, the FPGA configuration logic looks for a synchronization word in the bitstream. This synchronization word is shown in Table 4.1. The configuration logic contains a Watchdog timer that counts down from a value specified by the Xilinx *bitgen* tool when the bitstream is generated (see Table 4.1). The Watchdog is disabled when the synchronization word is found. If the Watchdog timer times out, the strike count is incremented and configuration is reattempted as described above.

Note that the Watchdog timer is only enabled in master configuration modes, when configuration restarts. It is disabled when the synchronization word is found, or when the FPGA uses one of the other configuration modes [1].

Table 3: Bitstream synchronization word

31 .. 24	**23..16**	**15..8**	**7..0**
0xAA	0x99	0x55	0x66

The second failure mode of the configuration is based on the bitstream CRC. Each bitstream contains at the end a CRC word which the configuration logic checks before putting the FPGA in running mode. If the CRC at the end of the bitstream does not correspond to what the configuration logic computes, a CRC error occurs and the strike count is incremented as described above.

For configuration to be reattempted and to be able to generate configuration errors, the configuration logic needs some information present in the bitstream. This information can be provided by setting some *bitgen* flags when generating the bitstream. The necessary flags with the recommended settings are listed in Table 4.1. For more information on *bitgen* and these flags, refer to [7].

Table 4: *bitgen* flags

Flag	Setting	Description
-g reset_on_err	Yes	Enables the strike count mechanism for retrying to load the bitstream
-g CRC	Enable	Enables the generation of a CRC at the end of the bitstream
-g TIMER_CFG	1fff	Sets the watchdog timeout value to 8191 CCLK cycles

4.2 Generating the bitstreams

Refer to [6] for information on how to generate the various bitstreams for a MultiBoot design.

4.3 Important note regarding MultiBoot bitstreams

Users should be aware that they should include the *xil_multiboot* module when generating a new MultiBoot bitsream. Otherwise, once a bitstream without the *xil_multiboot* module inside it is loaded into the FPGA, the remote reprogramming capability of the FPGA is lost, and the user will need to use JTAG or other means to program the FPGA with a MultiBoot-enabled design.

Thus, always remember to include the *xil_multiboot* module in any bitstream generated after the Golden bitstream.

5 Implementation

A block diagram of the design has already been presented in Figure 1. This section describes the actions of the sub-modules in the *xil_multiboot* module. The description is rather high-level, some details are omitted for ease of understanding. The main purpose of this section is for the reader to understand how the MultiBoot module works together with the flash chip and the configuration logic to achieve FPGA reprogramming, rather than specifying every detail about the module and its sub-modules.

For more involved details, the user is free to consult the code in the project repository [8].

The *multiboot_fsm* module is at the heart of the design. It implements a finite-state machine (FSM) with 34 states, which controls operation of the module. A simplified diagram of the FSM is shown in Figure 3. The diagram shows the various phases the FSM is in, and Table 5 lists these phases. Bear in mind that each phase may contain multiple FSM states. However, many of these states are just steps of accessing configuration logic

through the Xilinx ICAP module, so for simplicity they are not listed in this manual. Refer to the code for more complete description.

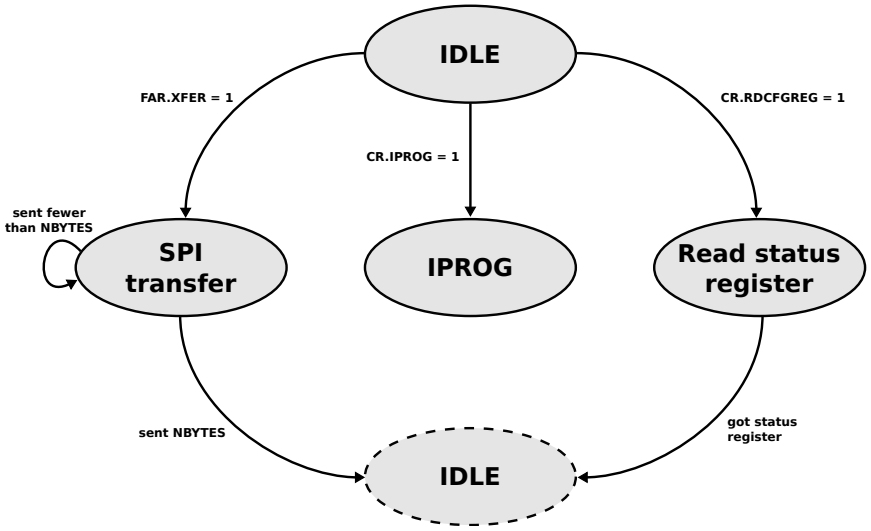

Figure 3: Phases of the *multiboot_fsm*

Table 5: Phases of the *multiboot_fsm*

Phase	Description
IDLE	Wait for one of the following control bits to be set: CR.RDCFGREG CR.IPROG FAR.XFER
SPI transfer	Shift out NBYTES of the three DATA fields in the FAR register, and simultaneously shift in data received from the flash When NBYTES have been sent, FAR.READY is written high and the FSM returns to IDLE
IPROG	IPROG sequence (Table 7-1, p.130 [1])
Read status register	Configuration register readout sequence (Table 6-1, p.113 [1])

5.1 Sending data to the flash chip

Table 6 summarizes the flash read and write sequence. A more verbose description is offered below.

Data to be sent to the flash chip is written in the FAR (see Appendix A.5). Up to three data bytes can be sent via the FAR during one transfer phase. These data bytes are written in the DATA fields in little-endian order. The

Table 6: Flash data sequence

Step	Action
1	User sets FAR.NBYTES to the number of data bytes to write
2	User writes NBYTES FAR.DATA fields with the data to send to the flash
3	User sets FAR.CS to '1' for a transfer with the flash chip enabled, or to '0' for a dummy transfer (e.g., wait interval between flash commands, with the chip select high)
4	User sets FAR.XFER to '1' to start the SPI transfer (FAR.XFER is automatically cleared by hardware)
5	The *multiboot_fsm* starts shifting out NBYTES DATA fields to the *spi_master*, starting with DATA[0]
6	The *spi_master* handles shifting out each bit in a byte
7	When done, the *spi_master* signals the *multiboot_fsm*, which shifts out the next byte (if NBYTES > 0)
8	When NBYTES bytes have been shifted out, the *multiboot_fsm* sets the FAR.READY bit
9	After FAR.READY is set, NBYTES DATA fields contain data retrieved from the flash

NBYTES field selects how many of the DATA fields contain bytes to send. Setting XFER to '1' with CS set to '1' starts the transfer. Setting XFER to '1' with CS set to '0' starts a dummy transfer, with the flash chip not selected.

The transfer is performed via the *spi_master* module, which handles the shifting of each DATA byte to the flash. However, the *spi_master* module can only send one byte at a time, so the *multiboot_fsm* module handles shifting bytes to the *spi_master*. After a byte has been transferred between the FPGA and the flash, the *multiboot_fsm* places the byte returned from the flash into the DATA byte that has just been sent. For example, after sending DATA[0], the byte received from the flash is placed into DATA[0]; after sending DATA[1], the byte received from the flash is placed into DATA[1].

When NBYTES data transfers have been completed, the *multiboot_fsm* sets the signal for the FAR.READY bit to '1', to signal a completed transfer. The user can now read the DATA fields for data retrieved from the flash.

The feature of using more than one DATA field in the FAR is useful when a lower-speed interface than SPI is used to access the FAR register (e.g., the VBCP interface in the CONV-TTL-BLO project [4]). If the interface used to access the FAR is fast, only one byte in the FAR register may be used, and NBYTES left to 0.

Appendix B gives an example of how data can be written to flash.
The following settings are used for the SPI communication (Figure 4).

- CPOL = 0
- CPHA = 0

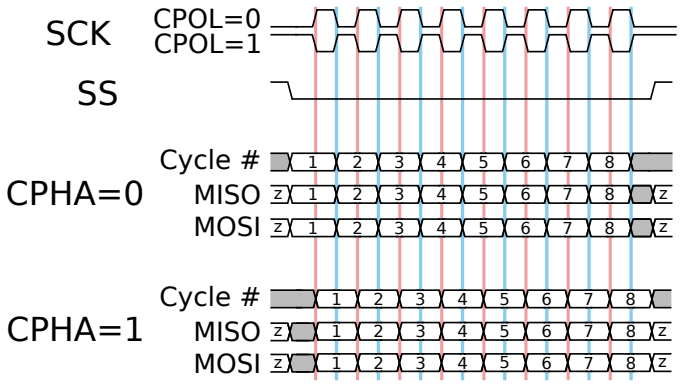

Figure 4: SPI settings (source: Wikipedia)

This yields that data bits are shifted out from the FPGA on the falling edge of SCLK and shifted in from the flash chip on the rising edge of SCLK.

5.2 Reading FPGA configuration registers

The Spartan-6 FPGA contains registers which can be read to get the stauts of the configuration logic. In order to read these registers, a special sequence must be followed. The sequece (listed in Table 6-1 of [1]) is implemented in the *multiboot_fsm*. Table 7 lists the sequence users should follow to read out an FPGA configuration register via the *xil_multiboot* module.

When the user sets the RDCFGREG bit in the *xil_multiboot* control register (CR), the *multiboot_fsm* initiates the configuration register readout sequence. It first sends a synchronization word, and then takes the value of CFGREGADR from the CR and use it to build a Type 1 configuration frame to read the configuration register (see [1], p. 93 for more details). The configuration logic will respond to this frame with the value of the configuration register. This value is placed in the CFGREGIMG field of the image register, and the *multiboot_fsm* continues to perform the final steps of the configuration register readout sequence, prior to returning to IDLE.

Note that some configuration registers in the Spartan-6 FPGA are more than 16 bits wide. The *xil_multiboot* module does not support reading the full length of the registers; it can only return the least significant 16 bits of the registers.

Table 7: Configuration register readout via *xil_multiboot*

Step	Action
1	User writes the FPGA configuration register address (Table 5-30, p.94 [1]) in the CFGREGADR field of the CR
2	User sets the CR.RDCFGREG bit to '1' to initiate a configuration register read via the ICAP module (CR.RDCFGREG is automatically cleared by hardware)
3	The *multiboot_fsm* performs the sequence in Table 6-1, p.113 [1] and returns one 16-bit value of the configuration register to the CFGREGIMG field of IMGR and sets the VALID bit of the same register to '1'
4	The user reads the configuration register value from IMGR.CFGREGIMG if the VALID bit is '1'

5.3 Sending the IPROG command

Table 8 lists the actions needed to issue the IPROG command to the FPGA using the *xil_multiboot* module. When the IPROG bit is set in the CR, the *multiboot_fsm* handles sending the IPROG sequence (Table 7-1, p.130 [1]) to the ICAP.

Table 8: Sequence for sending the IPROG command

Step	Action
1	User sends the bitstream to the flash chip (Section 5.1)
2	User sets the IPROG_UNL bit in the CR
3	User sets the IPROG bit in the CR
4	The *multiboot_fsm* performs the sequence in Table 7-1, p.130 [1] and sends the IPROG command
5	The FPGA starts deleting the configuration logic and loading the new bitstream from the flash
6	After configuration finishes, the user reads the custom firmware version number register to see that the reprogramming was successful

Note that after the IPROG command is sent, the FPGA starts the reconfiguration sequence and communication to it will be lost until the new bitstream is loaded is sent.

The user should implement some form of firmware version numbering to detect whether an IPROG succeeds. This version number can be stored into a read-only register in the FPGA and read after the IPROG command. An

example of this is given in the CONV-TTL-BLO project [4].

6 Modifying the design

The *xil_multiboot* module is purposely modular in case users want to interface to different FPGA interconnect standards, or different flash chips. In order to make modifications to the design, knowledge of VHDL is required.

If the user would like to adapt the design for a new FPGA interconnect standard, the steps to be followed are listed in Table 9.

Table 9: Changing the Wishbone interconnect

Step	Action
1	Change *wbs_i* and *wbs_o* in *xil_multiboot* ports to the preferred interconnect ports
2	Implement or change the current *multiboot_regs* module, keeping the interface to the FSM side (e.g., the *multiboot_cr_iprog_o* port, etc.)
3	Instantiate the new *multiboot_regs* module into the *xil_multiboot* module

The SPI interface to the flash chip is hardwired to the settings listed in Section 5.1. Table 10 lists the steps to be performed in case these SPI settings need to be changed.

Table 10: Modifying SPI settings

Step	Action
1	First, see if the CPOL setting alone will not fix the problem. If so, simply change the *cpol_i* port value where the *spi_master* is instantiated
2	Change the design of the *spi_master* module; it is an easy-to-follow FSM design

Another potential addition to the design would be the capability of reading the full value of all FPGA configuration registers. As outlined in Section 5.2, some configuration registers are more than 16 bits in length, and the *xil_multiboot* module cannot return their full value. This can be modified by, implementing an extra COUNT field in the CR; the *multiboot_fsm* can then use this field to build a Type 1 configuration package to return the full length of the configuration register. More information on this can be found in the Configuration Packets section of [1].

7 Synthesis results

The synthesis results for the *xil_multiboot* design using *xst* on the Spartan-6 XC6SLX45T are shown in Table 11.

Table 11: Synthesis results

Resource	**Used**	**Available**	**%**
Slices	123	6822	1.8
Slice registers	270	54576	0.5
LUTs	332	27288	1.2

Appendices

A Memory map

The memory map of the Xilinx MultiBoot module is shown below. The following sections detail the fields of each register.

Offset	Name	Description
0x00	CR	Control Register
0x04	SR	Status register
0x08	GBBAR	Golden Bitstream Base Address Register
0x0c	MBBAR	Multiboot Bitstream Base Address Register
0x10	FAR	Flash access register

A.1 CR – Control Register

Bits	Field	Access	Default	Description
31..18	*Reserved*	–	X	
17	IPROG	R/W	0	IPROG bit
16	IPROG_UNL	R/W	0	IPROG unlock bit
15..7	*Reserved*	–	X	
6	RDCFGREG	R/W	0	Read config register
5..0	CFGREGADR	R/W	0	Config register address

Field	Description
Reserved	Write as '0'; read undefined
IPROG	When 1, it triggers the FSM to send the IPROG command to the ICAP controller This bit needs to be unlocked by setting the IPROG_UNL bit in a previous cycle
IPROG_UNL	Unlock bit for the IPROG command. This bit needs to be set to 1 prior to writing the IPROG bit
RDCFGREG	Initiate a read from the FPGA configuration register at address CFGREGADR This bit is automatically cleared by hardware
CFGREGADR	The address of the FPGA configuration register to read (see Configuration Registers section in [1])

A.2 IMGR – Image Register

Bits	Field	Access	Default	Description
31..17	*Reserved*	–	X	
16	VALID	R/O	0	Image register is valid
15..0	CFGREGIMG	R/O	0	Config. register image

Field	Description
Reserved	Write as '0'; read undefined
VALID	A read has been performed from the FPGA configuration register at address CR.CFGREGADR, and its value is present in CFGREGIMG
CFGREGIMG	Contains the value of the FPGA configuration register; validated by the VALID bit (see Configuration Registers section in [1])

A.3 GBBAR – Golden Bitstream Base Address Register

Bits	Field	Access	Default	Description
31..24	OPCODE	R/W	0	Flash chip read op-code
23..0	GBA	R/W	0	Golden Bitstream Address

Field	Description
OPCODE	Op-code for the flash chip read (or fast-read) command. Get this value from the flash chip datasheet
GBA	Start address of the Golden bitstream on the flash chip

A.4 MBBAR – MultiBoot Bitstream Base Address Register

Bits	Field	Access	Default	Description
31..24	OPCODE	R/W	0	Flash chip read op-code
23..0	MBA	R/W	0	MultiBoot Bitstream Address

Field	Description
OPCODE	Op-code for the flash chip read (or fast-read) command. Get this value from the flash chip datasheet
MBA	Start address of the MultiBoot bitstream on the flash chip

A.5 FAR – Flash Access Register

Bits	Field	Access	Default	Description
31..29	*Reserved*	–	0	Flash chip read op-code
28	READY	R	1	SPI access status
27	CS	R/W	0	SPI chip select
26	XFER	R/W	0	Start SPI transfer
25..24	NBYTES	R/W	0	Number of bytes to send
23..16	DATA[2]	R/W	0	Data at offset 2
15..8	DATA[1]	R/W	0	Data at offset 1
7..0	DATA[0]	R/W	0	Data at offset 0

Field	Description
Reserved	Write as '0'; read undefined
READY	SPI transfer ready; NBYTES have been sent to the flash chip, and NBYTES read from the chip present in DATA fields
CS	SPI chip select. Note that this pin has opposite polarity than the normal SPI chip select pin: '1' – flash chip is selected (CS pin = 0) '0' – flash chip is not selected (CS pin = 1)
XFER	'1' – starts SPI transfer This bit is automatically cleared by hardware
NBYTES	Number of DATA fields to send in one transfer 0 – send 1 byte (DATA[0]) 1 – send 2 bytes (DATA[0], DATA[1]) 2 – send 3 bytes (DATA[0], DATA[1], DATA[2]) 3 – *Reserved*
DATA[2]	Write this register with the value of data byte 2 After an SPI transfer, this register contains the value of data byte 2 read from the flash
DATA[1]	Write this register with the value of data byte 1 After an SPI transfer, this register contains the value of data byte 1 read from the flash
DATA[0]	Write this register with the value of data byte 0 After an SPI transfer, this register contains the value of data byte 0 read from the flash

B Sending multiple data fields in one SPI transfer

The FAR register contains three bytes for data fields to be sent to the SPI chip. As mentioned in Section 5.1, this can be used to send data faster when the interface used to access the FPGA is slow.

This section shows an example of how to write the values listed in Table 12 to an 8-bit flash chip starting with address 0. Since there are 10 values to be sent, the transfer can be grouped into three 3-byte transfers and one one-byte transfer. The code snippet below shows how these transfers can be performed by writing to the FAR, and Figure 5 shows the effects of each of the writes.

Table 12: Values to send to flash chip

0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a

```
FAR = 0x0e030201;
while !(FAR & (1<<28))
    ;
FAR = 0x0e060504;
while !(FAR & (1<<28))
    ;
FAR = 0x0e090807;
while !(FAR & (1<<28))
    ;
FAR = 0x0c00000a;
while !(FAR & (1<<28))
    ;
```

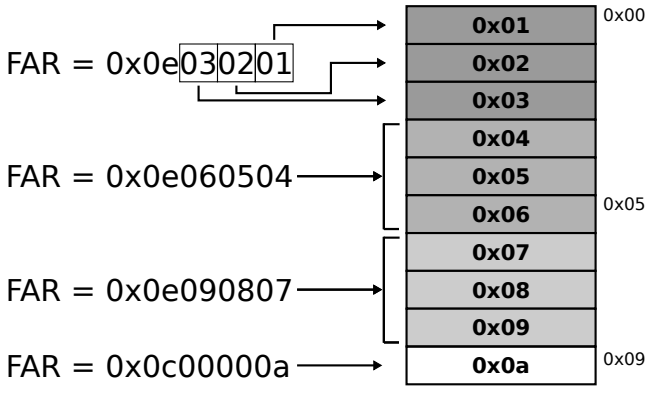

Figure 5: Effects of FAR writes

References

[1] Xilinx, "UG380 - Spartan-6 Configuration Guide." http://www.xilinx.com/support/documentation/user_guides/ug380.pdf, Jan. 2013. v2.5.

[2] "Open Hardware Repository." http://www.ohwr.org/.

[3] "Generic Cores Library OHWR Project Page." http://www.ohwr.org/projects/general-cores/wiki.

[4] "CONV-TTL-BLO Project Page on OHWR." http://www.ohwr.org/projects/conv-ttl-blo/wiki, Oct. 2013.

[5] Xilinx, "XTP059: SP605 MultiBoot Design." http://www.xilinx.com/support/documentation/boards_and_kits/xtp059.pdf.

[6] T.-A. Stana, "Generating Bitstreams for MultiBoot designs (Xilinx MultiBoot module Project Page on OHWR)." http://www.ohwr.org/projects/conv-ttl-blo-gw/wiki/Xil_multiboot#Generating-bitstreams-for-Xilinx-FPGA-reprogramming, Oct. 2013.

[7] Xilinx, "Xilinx Command Line Tools User Guide (UG628)." http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_2/devref.pdf, July 2012.

[8] "CONV-TTL-BLO Gateware Repository on OHWR." http://www.ohwr.org/projects/conv-ttl-blo-gw/repository, Oct. 2013.