XAPP058 (v4.0) October 1, 2007

Xilinx In-System Programming Using an Embedded Microcontroller

Summary

The Xilinx high-performance CPLD, FPGA, and configuration PROM families provide in-system programmability, reliable pin locking, and JTAG Boundary-Scan test capability. This powerful combination of features allows designers to make significant changes and yet keep the original device pinouts, thus, eliminating the need to re-tool PC boards. By using an embedded controller to program these CPLDs and FPGAs from an on-board RAM or EPROM, designers can easily upgrade, modify, and test designs, even in the field.

Xilinx Families

Virtex™ Series, Spartan™ Series, CoolRunner™ Series, 9500 Series, Platform Flash PROM Family, and 18V00 Family.

Introduction

The Xilinx CPLD and FPGA families combine superior performance with an advanced architecture to create new design opportunities that were previously impossible. The combination of in-system programmability, reliable pin locking, and JTAG test capability gives the following important benefits:

- Reduces device handling costs and time to market
- Saves the expense of laying out new PC boards
- Allows remote maintenance, modification, and testing
- Increases the life span and functionality of products
- Enables unique, customer-specific features

The ISP controller shown in Figure 1 can help designers achieve these unprecedented benefits by providing a simple means for automatically programming Xilinx CPLDs and FPGAs from design information stored in EPROM. This design is easily modified for remote downloading applications and the included C code can be compiled for any microcontroller.

To create device programming files, Xilinx provides iMPACT tool, included with the standard Xilinx ISE™ software. The iMPACT software automatically reads standard JEDEC/BIT/MCS/EXO device programming files and converts them to a compact binary format XSVF format that can be stored in the on-board EPROM or RAM. The XSVF format contains both data and programming instructions for the CPLDs, FPGAs, and configuration PROMs. JEDEC files are the programming files converted for CPLDs, BIT files for FPGAs, and MCS/EXO files for configuration PROMs. The 8051 microcontroller interprets the XSVF information and generates the programming instructions, data, and control signals for the Xilinx devices.

By using a simple IEEE 1149.1 (JTAG) interface, Xilinx devices are easily programmed and tested without using expensive hardware. Multiple devices can be daisy-chained, permitting a single four-wire Test Access Port (TAP) to control any number of Xilinx devices or other JTAG-compatible devices.

The files and utilities associated with this application note are available in a package for downloading from:

ftp://ftp.xilinx.com/pub/swhelp/cpld/eisp_pc.zip

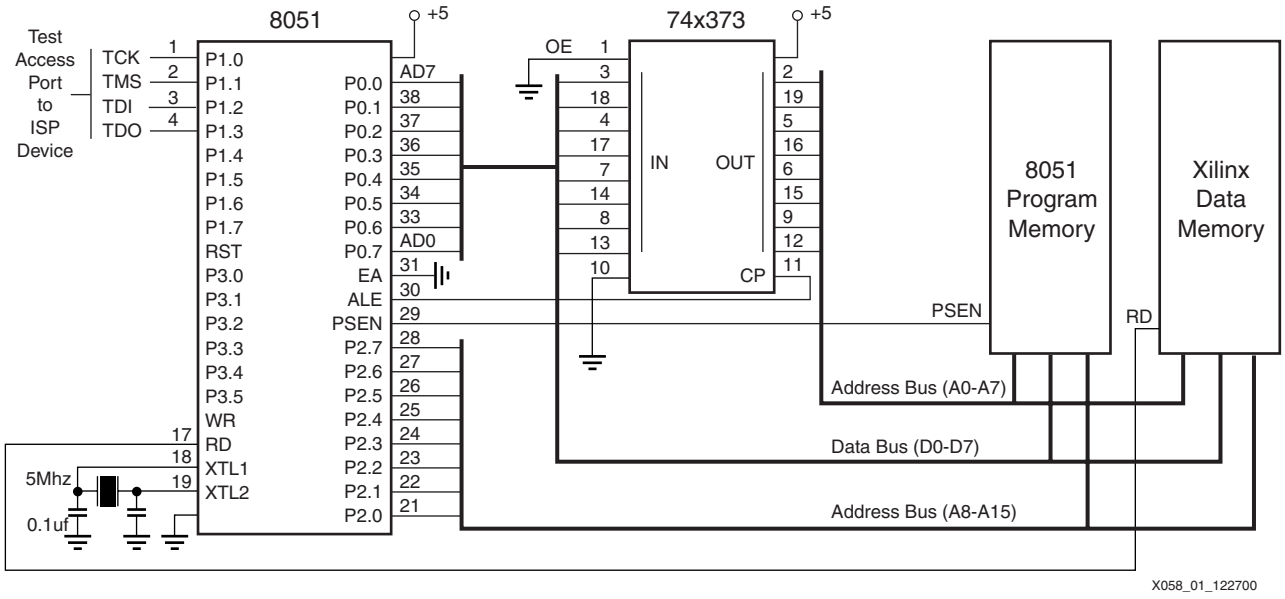


Figure 1: **ISP Controller Schematic**

Programming Xilinx CPLDs, FPGAs, and Configuration PROMs

Serial Vector Format (SVF) is a syntax specification for describing high-level IEEE 1149.1 (JTAG) bus operations. SVF was developed by Texas Instruments and has been adopted as a standard for data interchange by JTAG test equipment and software manufacturers such as Teradyne, Tektronix, and others. Xilinx CPLDs, FPGAs, and configuration PROMs accept programming and JTAG Boundary-Scan test instructions in SVF format, via the TAP. The timing for these TAP signals is shown in Figure 16, page 18. Since the SVF format is ASCII and has larger memory requirements it is inefficient for embedded applications. Therefore, to minimize the memory requirements, SVF is converted into a more compact (binary) format called XSVF.

The iMPACT software tool, included with Xilinx ISE, automatically converts standard JEDEC/BIT/MCS/EXO programming files into XSVF format. In this design, an 8051 C-code algorithm interprets the XSVF file and provides the required JTAG TAP stimulus to the target, performing the programming and (optional) test operations originally specified in the XSVF file.

Note: For a description of the SVF and XSVF commands and file formats, see [Ref 1].

The flow for creating the programming files that are used with this design are shown in Figure 2, Figure 3, page 3, and Figure 4, page 3.

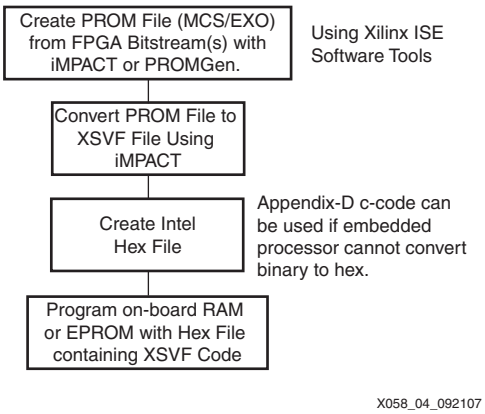


Figure 2: **Configuration PROM Programming File Creation and Storage Flow**

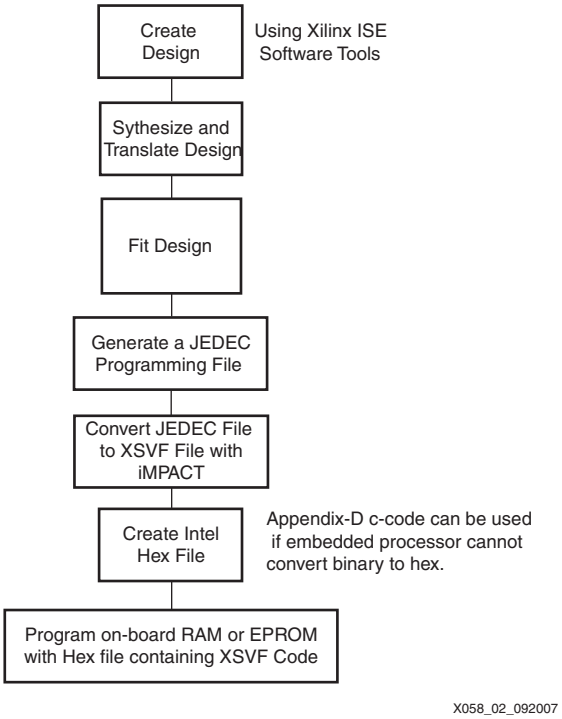


Figure 3: **CPLD Programming File Creation and Storage Flow**

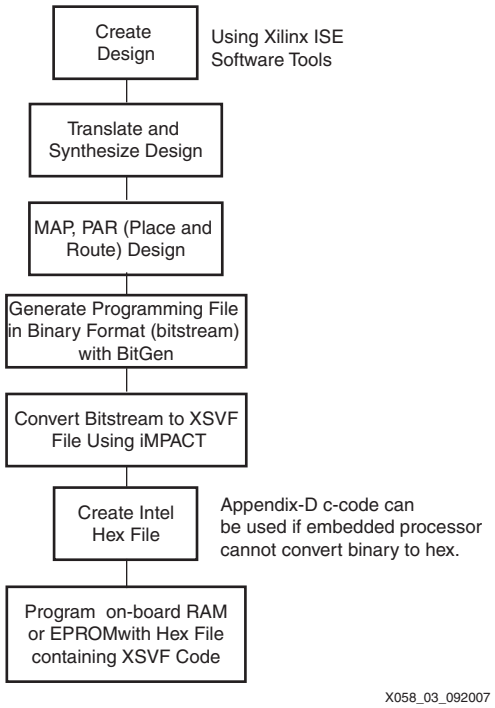


Figure 4: **FPGA Programming File Creation and Storage Flow**

JTAG Instruction Summary

Xilinx devices accept both programming and test instructions using the JTAG TAP. The JTAG commands and descriptions used for programming and functional testing are listed in Table 1:

Table 1: **JTAG Instruction for Programming and Testing**

Instruction	Description
Instructions Supported by all Devices	
EXTEST	Isolates the device I/O pins from the internal device circuitry to enable connectivity tests between devices. It uses the device pins to apply test values and to capture the results.
INTEST	Isolates the device from the system, applies test vectors to the device input pins, and captures the results from the device output pins.
SAMPLE/PRELOAD	Allows values to be loaded into the Boundary-Scan register to drive the device output pins. Also captures the values on the input pins.
BYPASS	Bypasses a device in a Boundary-Scan chain by functionally connecting TDI to TDO.
Instructions Common to CPLD, FPGAs, and Configuration PROMs	
EXTEST	Isolates the device I/O pins from the internal device circuitry to enable connectivity tests between devices. It uses the device pins to apply test values and to capture the results.
IDCODE	Returns a 32-bit hardwired identification code that defines the part type, manufacturer, and version number.
HIGHZ	Causes all device pins to float to a high-impedance state.
Instructions Supported by XC4000 Series, Spartan, and Spartan-XL Families Only	
CONFIGURE	Allows access to the configuration bus for configuration.
READBACK	Allows access to the configuration bus for readback.
Instructions Supported by Virtex Series and Spartan-II/3/3E/3A/3AN Families Only	
CFG_IN/CFG_OUT	Allows access to the configuration bus for configuration and readback.
JSTART	Clock the startup sequence when startup clock = JTAGCLK.
Commands Supported by CPLDs and Configuration PROMs	
ISPEN	Enables the ISP function in the XC9500/XL/XV device, floats all device function pins, and initializes the programming logic.
FERASE	Erases a specified program memory block.
FPGM	Programs specific bit values at specified addresses. An FPGMI instruction is used for the XC95216 and larger devices which have automatic address generation capabilities.
FVFY	Reads the fuse values at specified addresses. An FVFYI instruction is used for the XC95216 and larger devices which have automatic address generation capabilities.
ISPEX	Exits ISP Mode. The device is then initialized to its programmed function with all pins operable.

Table 2 list instructions that are also available but are not used for programing or functional testing:

Table 2: **Additional JTAG Instructions**

Instruction	Description
Instructions Specific to CPLDs and Configuration PROMs	
USERCODE	Returns a 32-bit user-programmable code that can be used to store version control information or other user-defined variables.
Instructions Specific to XC4000 Series, Spartan and Spartan-XL Families	
USER1/USER2	These instructions allow capture, shift and update of user-defined registers.
Instructions Specific to Virtex Series and Spartan-II/3/3E/3A/3AN Families	
USR1/USR2	These instructions allow capture, shift and update of user-defined registers.
Instructions Specific to Configuration PROMs	
FADDR	Sets the PROM array address register.
DATA0	Accesses the array word-line register.
PROGRAM	Programs the word-line into the array.
SERASE	Globally refines the programmed values in the array.

Creating an XSVF File Using iMPACT Software

This procedure describes how to create an XSVF file from a FPGA, CPLD or PROM programming file. This flow assumes that Xilinx ISE software is being used. This software package includes the Xilinx CPLD and FPGA implementation tools and the iMPACT programming and file generation software.

iMPACT is supplied with both a graphical and batch user interface. The graphical tool can be launched from the Project Manager. The batch tool is available by opening a shell and invoking **`impact -batch`** on the command line.

Using iMPACT Batch Tool to Create XSVF Files

After generating the programming files as specified in Figure 2, page 2, Figure 3, page 3, and Figure 4, page 3, the user can use the iMPACT batch tool to generate XSVF files:

1. Invoke the iMPACT batch tool from the command line in a new shell:

   ```
   impact -batch
   ```

   The following messages appear:

   ```
   Release <Release Number> - iMPACT <Version Number>
   Copyright (c) 1995-2007 Xilinx, Inc. All rights reserved.
   ```

2. Set up the device types and assign design names by typing following command at the iMPACT batch prompt:

   ```
   setMode -bs
   setCable -port xsvf -file "c:/filename.xsvf"
   addDevice -p # -file "c:/designname.xxx"
   ```

   Where the # determines the position in the JTAG chain. A single device chain uses **`-p 1`**. The extension of the file added is `.jed` for a CPLD, `.bit` for a FPGA, or a PROM file format such as `.mcs` for the PROMs, or `.bsd` for a device being bypassed. The `.bsd` file is used for a non-Xilinx device in the JTAG chain when creating the XSVF file in iMPACT.

   The supported iMPACT batch commands can be found in the on-line iMPACT software manual. The most common operations and an example are listed in Table 3, page 6. For a detailed list of all possible options use the reference manual.

Table 3: **Selected iMPACT Batch Commands**

Command	Description and Usage
erase [-o|-override] -p|-position <position> [-ver <version>]	
	Generates an XSVF file to describe the Boundary-Scan sequence to erase the specified part. The **-o** flag is used to generate an erase sequence that overrides write protection on devices. The **-p** is used to specify the position of the device in the JTAG chain being targeted. The **-ver** is used to specify the design revision on the Platform Flash PROM (XCFxxP) device to erase.
verify -p|-position <position> [-ver <version>]	
	Generates an XSVF file to describe the Boundary-Scan sequence to read back the device contents and compare it against the contents of the specified JEDEC file. The **-p** flag indicates the position in the JTAG chain of the device to be verified. The **-ver** flag refers to the Platform Flash PROM (XCFxxP) version to be verified.
program [-e|-erase] [-v|-verify] -p|-position <position> [-u|-usercode <codestring>][-ver <version>] [-parallel] [-loadfpga]	
	Generates an XSVF file to describe the Boundary-Scan sequence to program the device using that programming data specified JEDEC/BIT/MCS/EXO file. The **-p** flag refers to the JTAG chain device position. The **-u** refers to the usercode being specified. The **-ver** refers to the version for the Platform Flash PROMs [XCFxxP] being targeted for programming. The **-parallel** flag indicates the PROM parallel configuration bit is set to load the FPGA in SelectMAP mode. The **-loadfpga** has the Xilinx PROM configure the FPGA automatically after being programmed. Recommended command for programming a CPLD or configuration PROM: **program -e -v -p 1** Recommended command for programming an FPGA: **program -p 1**
readidcode -p|-position <position>	
	Generates an XSVF file that verifies the expected 32-bit IDCODE from the device at <position> in the Boundary-Scan chain.
quit	
	Exits iMPACT batch mode.

Using the iMPACT GUI to Create XSVF Files

After generating the programming files as specified in Figure 2, page 2, Figure 3, page 3, and Figure 4, page 3, the user can use the iMPACT GUI to generate XSVF files

1. Double-click on the **Configure Device [iMPACT]** option under the "Generate Programming File" process selection in the ISE Project Navigator or open a system shell and type **`impact`**. In the iMPACT wizard select **Prepare a Boundary-Scan File,** and then select **XSVF** from the pull-down menu as shown in Figure 5 and then click **Finish**.

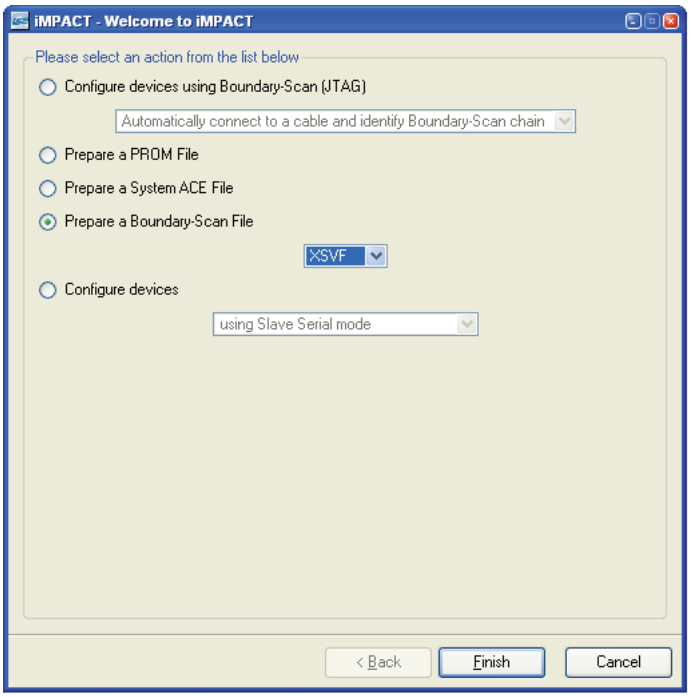
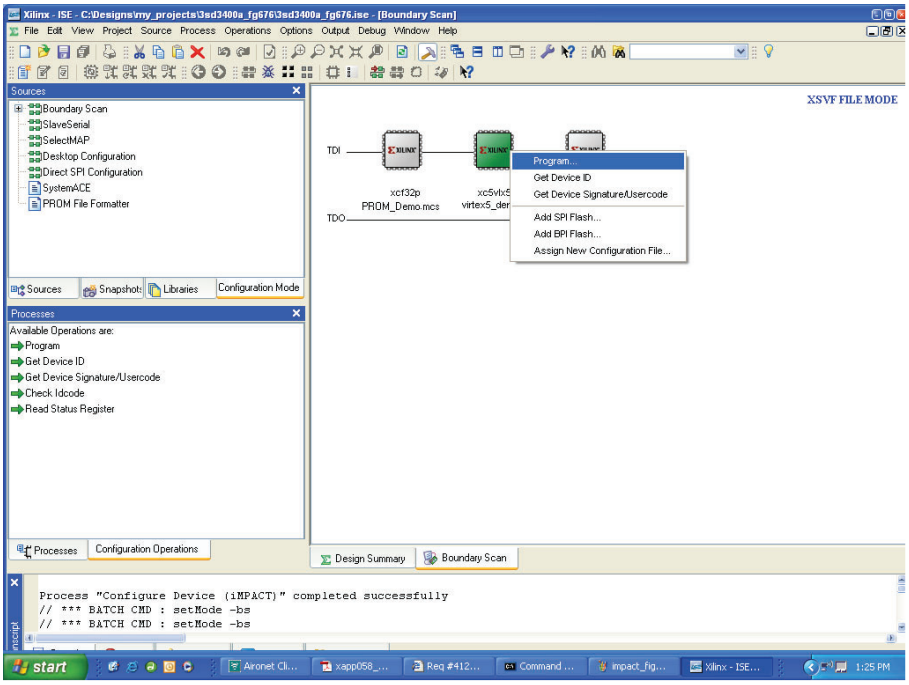


Figure 5: **iMPACT Wizard**

2. Set up the mode to write to the XSVF file. At the prompt, select the XSVF file name for data to be stored to. Click **OK** when the "File Generation Mode" prompt appears.
3. Set up the JTAG chain. First, specify the files for each of device in the JTAG chain. If there is more than one device in the chain, right-click next to the device to be added and select **Add Xilinx Device** or **Add Non-Xilinx Device** for each additional device in the JTAG chain.

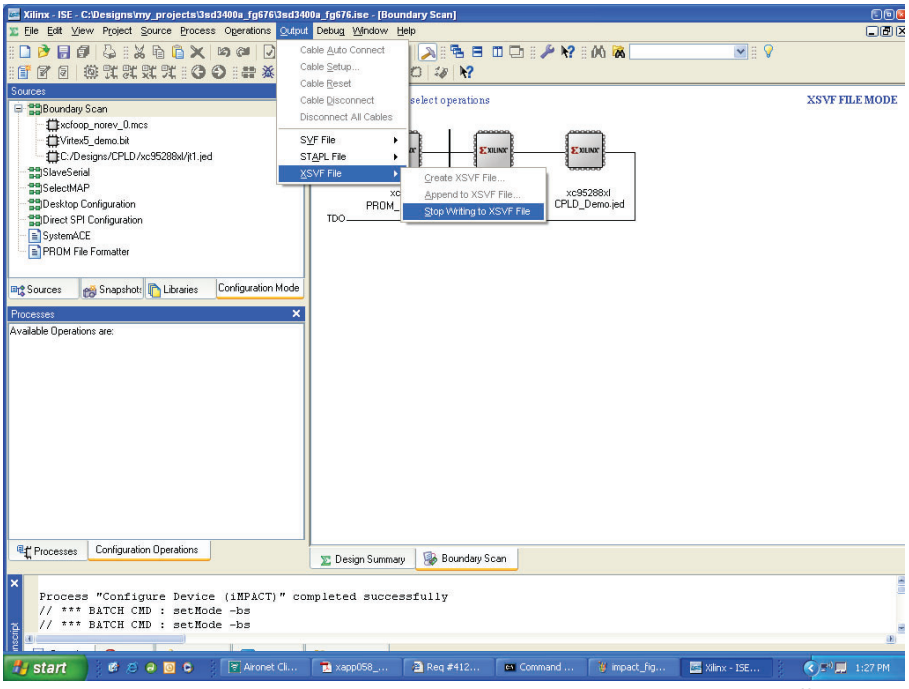
   Next, select the target device (it should be highlighted) and right-click to see the available operations for the device. Select the operation(s) to be written to the XSVF file in the order needed (Figure 6, page 8).



X058_22_092307

Figure 6: **Add Device**

4. **Output** → **XSVF File** → **Stop writing to the XSVF File** to exit and stop writing to the XSVF file (Figure 7).



X058_21_092307

Figure 7: **Add Device**

Notes:

1. The recommended operation for programming selected devices is **Operations** → **Program** with **Erase before Programming** and **Verify** selected. The **Verify** option is not recommended for FPGA devices.
2. Select **HIGHZ** instead of **BYPASS** option from the **Edit** → **Preferences** → **iMPACT** → **Configuration Preferences** menu if it is desired to have all the devices in the chain set to High-Z during the requested operation.

EPROM Programming

To program an EPROM, the binary XSVF file must be converted to an Intel Hex or similar PROM format file. Most embedded processor development system software automatically converts included binary files to the appropriate format. For systems without this capability, refer to "Appendix D: Binary to Intel Hex Translator," page 28 for reference C code to perform this conversion.

Software Limitations

iMPACT can generate XSVF files only for devices for which JEDEC/BIT/MCS/EXO files can be created. Designers should verify that the development software they are using can create JEDEC/BIT/MCS/EXO files for the specific devices they intend to use.

Instructions on generating SVF for CoolRunner CPLDs are available on the Xilinx Support website (http://support.xilinx.com).

Hardware Design

As shown in Figure 1, page 2, the reference design requires only an 8051 microcontroller, an address latch, and enough EPROM or RAM to contain both the 8051 code and the CPLD/FPGA/PROM programming data.

Hardware Design Description

The example 8051 design allows 64K of program and 64K of data space; however, some devices require more data space. The 8051 multiplexes port 0 for both data and addresses. The ALE signal causes the 74x373 to latch the low-order address, and the high-order address is output on port 2. Port 0 then floats, allowing the selected EPROM to drive the data inputs. Then the $\overline{\text{PSEN}}$ signal goes Low to activate an 8051 program read operation, or the $\overline{\text{RD}}$ signal goes Low to activate a CPLD programming data read operation.

Estimated EPROM Memory Requirements

Table 4 lists the estimated EPROM capacity needed to contain the programming data.

Table 4: **XSVF File Sizes**

Device Type	File Size (bytes)
XC9536	45572
XC9572	103928
XC95108	175250
XC95144	144222
XC95216	259620
XC95288	403698
XC9536XL	38186
XC9572XL	51590
XC95144XL	78398
XC95288XL	132014
XCR3064XL	21149
XCR3128XL	40067
XCR3256XL	90042
XC18V512	338119

Table 4: **XSVF File Sizes *(Cont'd)***

Device Type	File Size (bytes)
XC18V01	675399
XC18V02	1341767
XC18V04	2682183
XCS20XL	24010
XCS40XL	44186
XC2S100	103969
XC2S150	138352
XCV300	232876
XCV1000	814055
XCV100E	114943
XCV300E	249318
XCV600E	526368
XCV1000E	875119
XCV2000E	1349542
XC2V6000	397958
XC2VP7	565369

The XSVF file sizes are dependent only on the device type, not on the design implementation. If further compression of the XSVF file is needed, a standard compression technique, such as Lempel-Ziv can be used.

Modifications for Other Applications

The design presented in this application note is for a stand-alone ISP controller. However, it is also possible to apply these techniques to microcontrollers that might already exist within a design. To implement this design in an already existing microcontroller, all that is needed is four I/O pins to drive the TAP, and enough storage space to contain both the controller program and the CPLD/FPGA/PROM download data. In addition, care must be taken to preserve the JTAG port timing.

The TAP timing in this design is dependent on the 8051 clock. For other 8051 clock frequencies or for different microcontrollers, the timing must be calculated accordingly, in order to implement the timing specified in "Exception Handling," page 20.

The speed at which the TAP ports can be toggled affects the overall programming time for FPGAs and PROMs that require millions of TCK cycles to shift just the data. For CPLDs, the cumulative program pulse time has a greater affect on programming time than the data shift time.

Using a different microcontroller requires changing the I/O subroutine calls while preserving the correct TAP timing relationships. These subroutine calls are located in the `ports.c` file. All other C code is independent of the microcontroller and does not need to be modified.

RAM can be used instead of the EPROM in this design, allowing CPLD/FPGA/PROM devices to be programmed and tested remotely via modem, using remote control software written by the user.

Debugging Suggestions

The following suggestions can be helpful in testing this design:

- Decrease the TCK frequency to test that the wait times for program and erase are sufficiently long.
- Make certain that the function pins go into a 3-state condition in ISP mode.
- Test that the function pins initialize when ISP mode is terminated with the ISPEX command.
- Verify that the devices which are not being programmed are in bypass mode. Bypass mode causes TDO to be the same as TDI, delayed by one TCK clock pulse.
- Use the precompiled playxsvf.exe from the download package to execute the XSVF on a PC through the Xilinx Parallel Cable III or Parallel Cable IV.
- Generate a simple XSVF that only checks the IDCODE of the target device to test basic functionality of the hardware and software.
- Generate and execute separate XSVF files for the erase, blank check, program, and verify operations to narrow the problem area.
- Program the device from iMPACT and a download cable to verify basic hardware functionality.

Firmware Design

The flow chart for the C code is shown in Figure 8, page 12 through Figure 15, page 18. This code continuously reads the instructions and arguments from the XSVF file contained in the program data EPROM and branches in one of three ways based on the three possible XSVF instructions (XRUNTEST, XSIR, XSDR) as described in "Appendix A: XSVF File Merge Utility," page 22.

When the C code reads an XRUNTEST instruction, it reads in the next four bytes of data that specify the number of microseconds for which the device stays in the Run-Test/Idle state before the next XSIR or XSDR instruction is executed. The runTestTimes variable is used to store this value.

When the C code reads an XSIR instruction, it provides stimulus to the TMS and TCK ports until it arrives in the Shift-IR state. It then reads a byte that specifies the length of the data and the actual data itself, outputting the specified data on the TDI port. Finally, when all the data is outputted to the TDI port, the TMS value is changed and successive TCK pulses are output until the Run-Test/Idle state is reached again.

When the C code reads an XSDR instruction, it reads the data specifying the values that are output during the Shift-DR state. The code then toggles TMS and TCK appropriately to transition directly to the Shift-DR state. It then holds the TMS value at 0 in order to stay in the Shift-DR state and the data from the XSVF file is output to the TDI port while storing the data received from the TDO port. After all the data is outputted to the TDI port, TMS is set to 1 in order to move to the Exit-1-DR state. Then, the TDO input value is compared to the TDO expected value. If the two values fail to match, the exception handling procedure is executed as shown in Figure 18, page 20. If the TDO input values match the expected values, the code returns to the Run-Test/Idle state and waits for the amount of time specified by the runTestTimes variable (originally set in the XRUNTEST instruction).

Memory Map

The 8051 memory map is divided into two 64K byte blocks: one for the 8051 program and one for data. The 8051 program memory resides in the 8051 program block and is enabled by the $\overline{\text{PSEN}}$ signal. The Xilinx PLD program memory resides in the 8051 data block and is enabled by the $\overline{\text{RD}}$ signal. When additional data space is required, use one of the methodologies specified in the specific microprocessor's applications note.

Port Map

The 8051 I/O ports are used to generate the memory address and the TAP signals, as shown in Figure 1, page 2. Port 1 of the 8051 is used to control the TAP signals; Table 5 shows the port configuration.

Table 5: **8051 Port 1 Mapping**

TAP Pin	Port1 Bit	Configured As
TCK	0	Input
TMS	1	Input
TDI	2	Output
TDO	3	Input

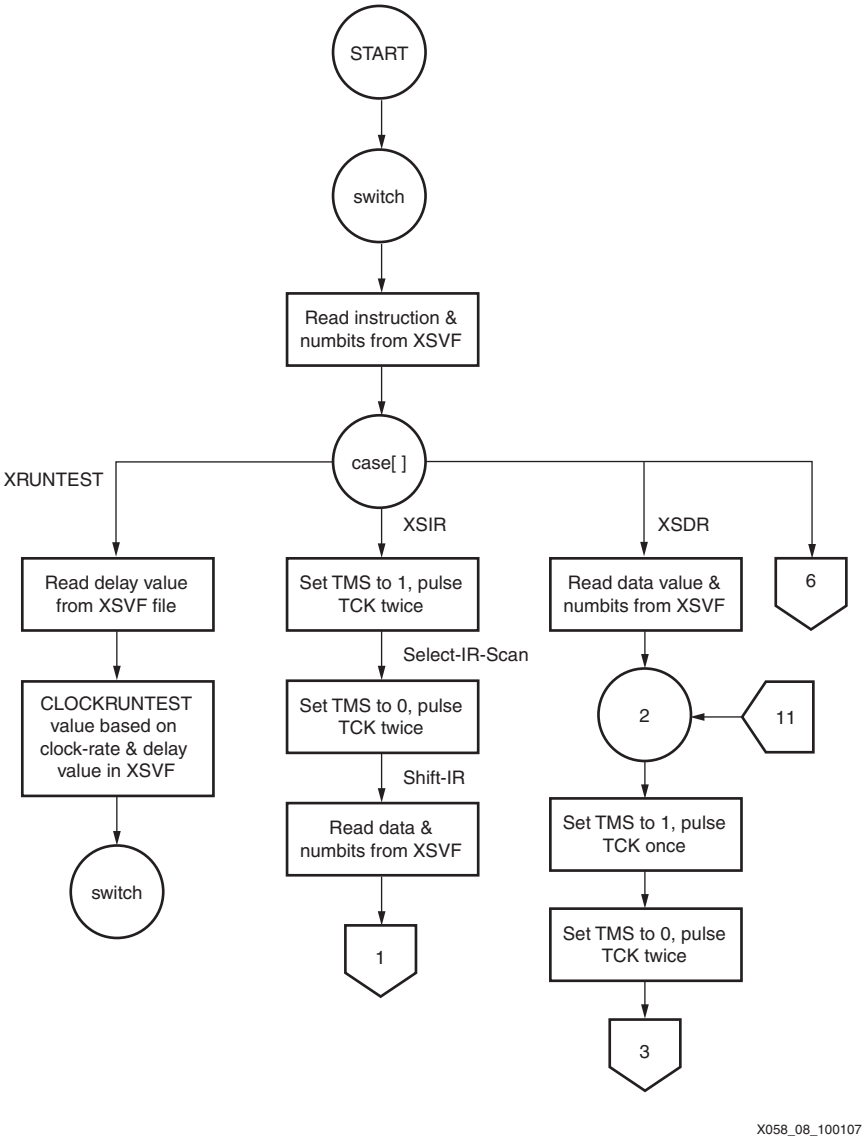
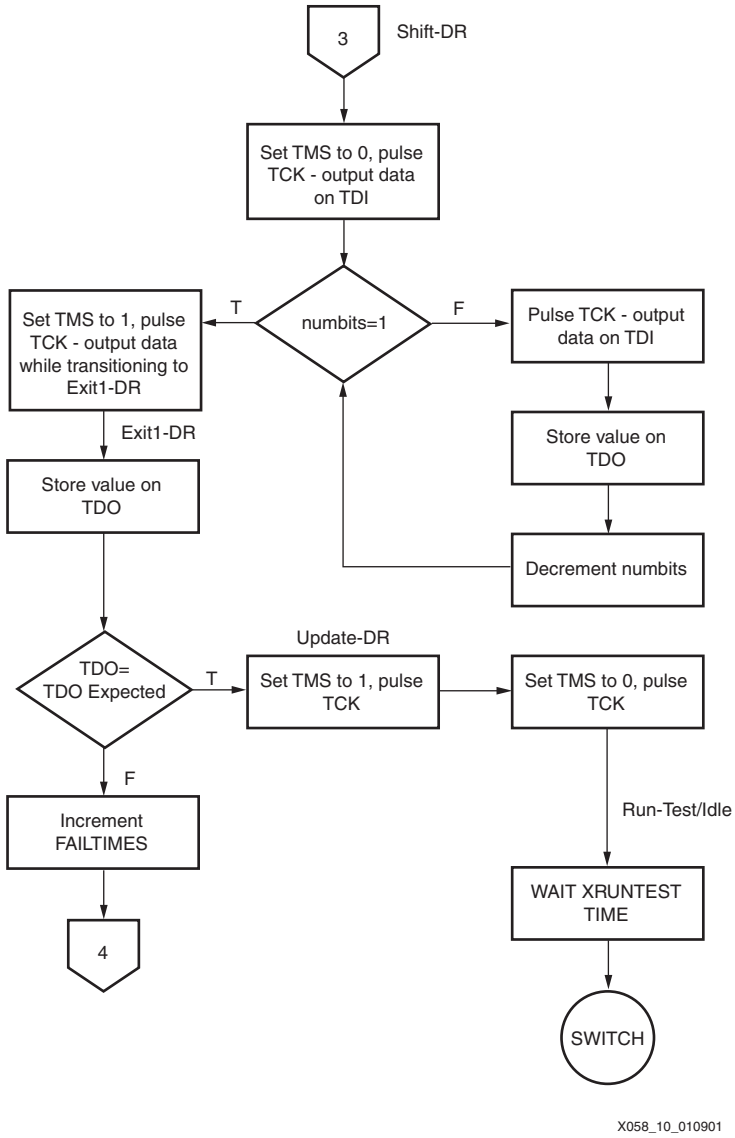


Figure 8: **Flow Chart for the ISP Controller Code**



Notes:

1. For FPGAs, step 4 is scrapped completely if the TDO expected does not match the actual TDO; the program quits with an error message.

Figure 9: **Flow Chart for the ISP Controller Code (Continued)**

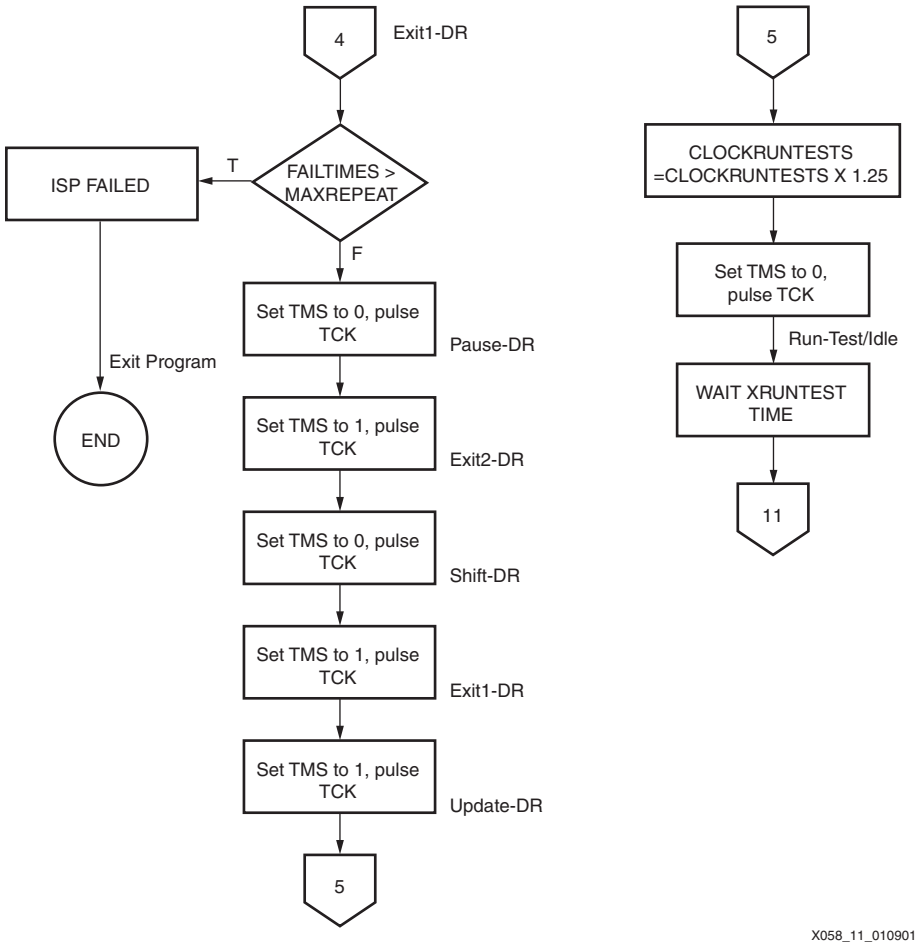


Figure 10: **Flow Chart for the ISP Controller Code (Continued)**

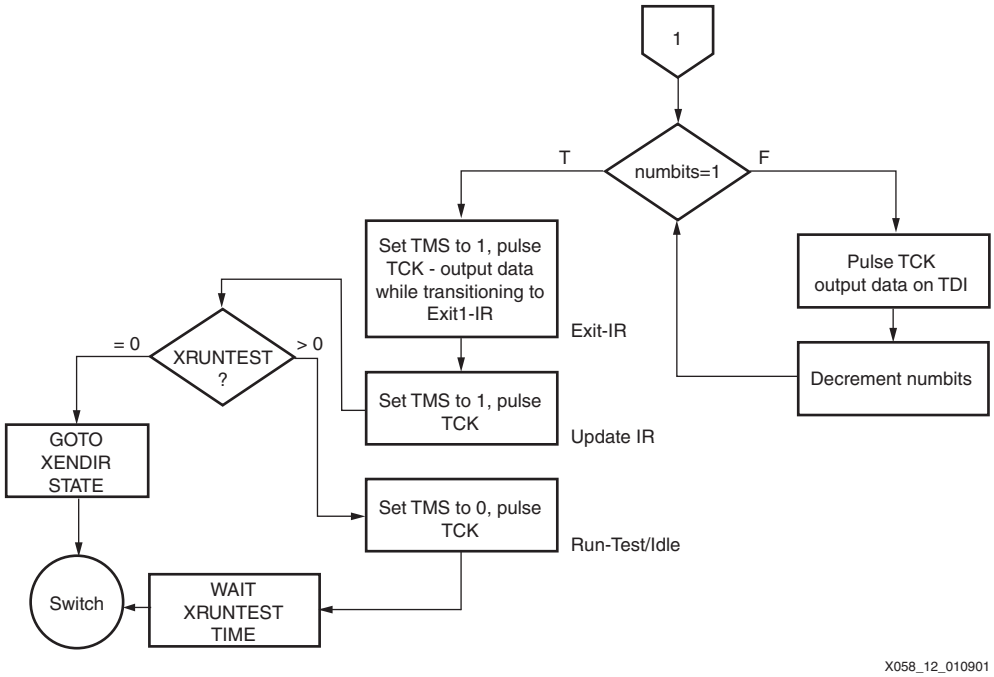


Figure 11: **Flow Chart for the ISP Controller Code (Continued)**

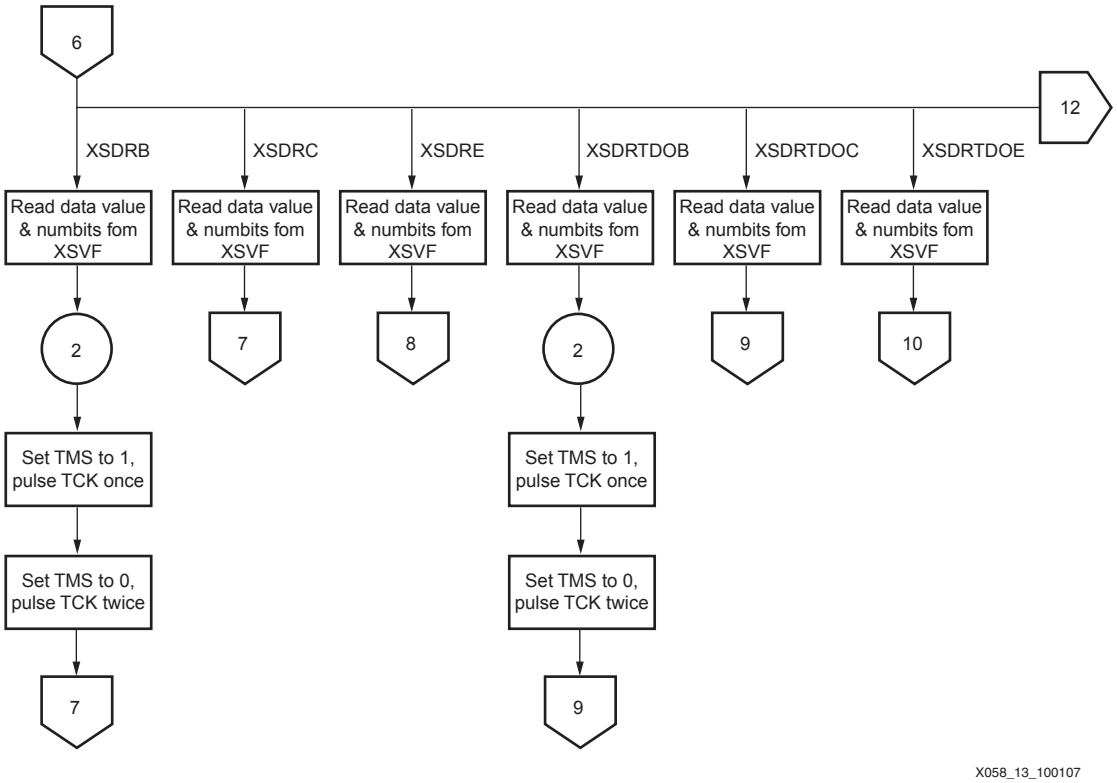


Figure 12: Flow Chart for the ISP Controller Code (Continued)

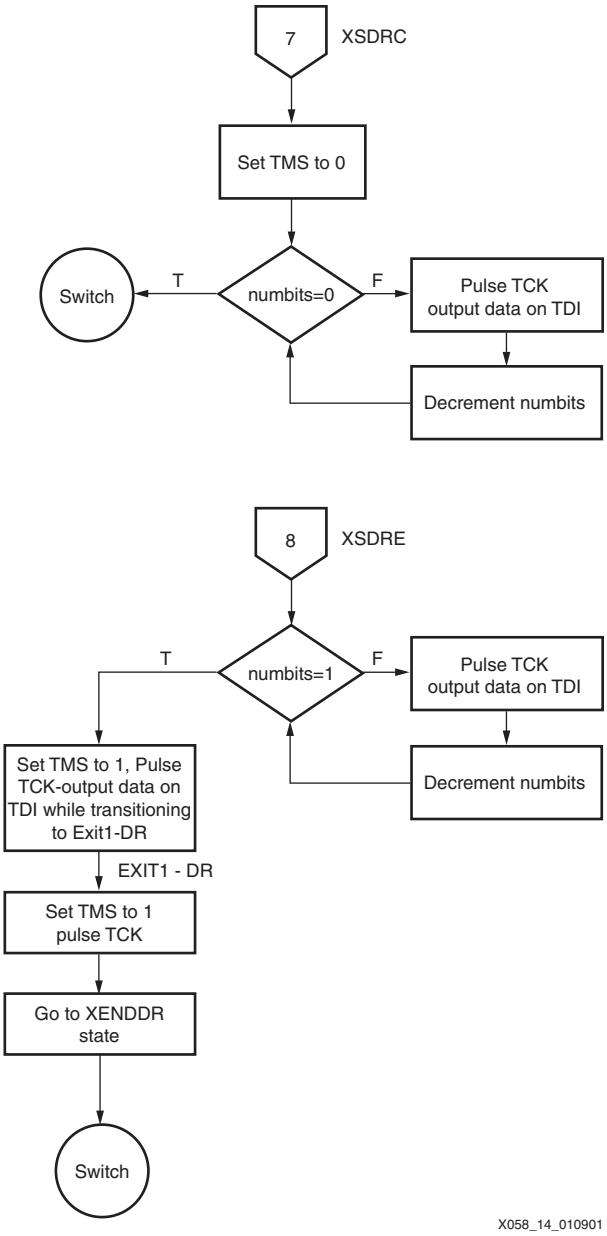


Figure 13: **Flow Chart for the ISP Controller Code (Continued)**

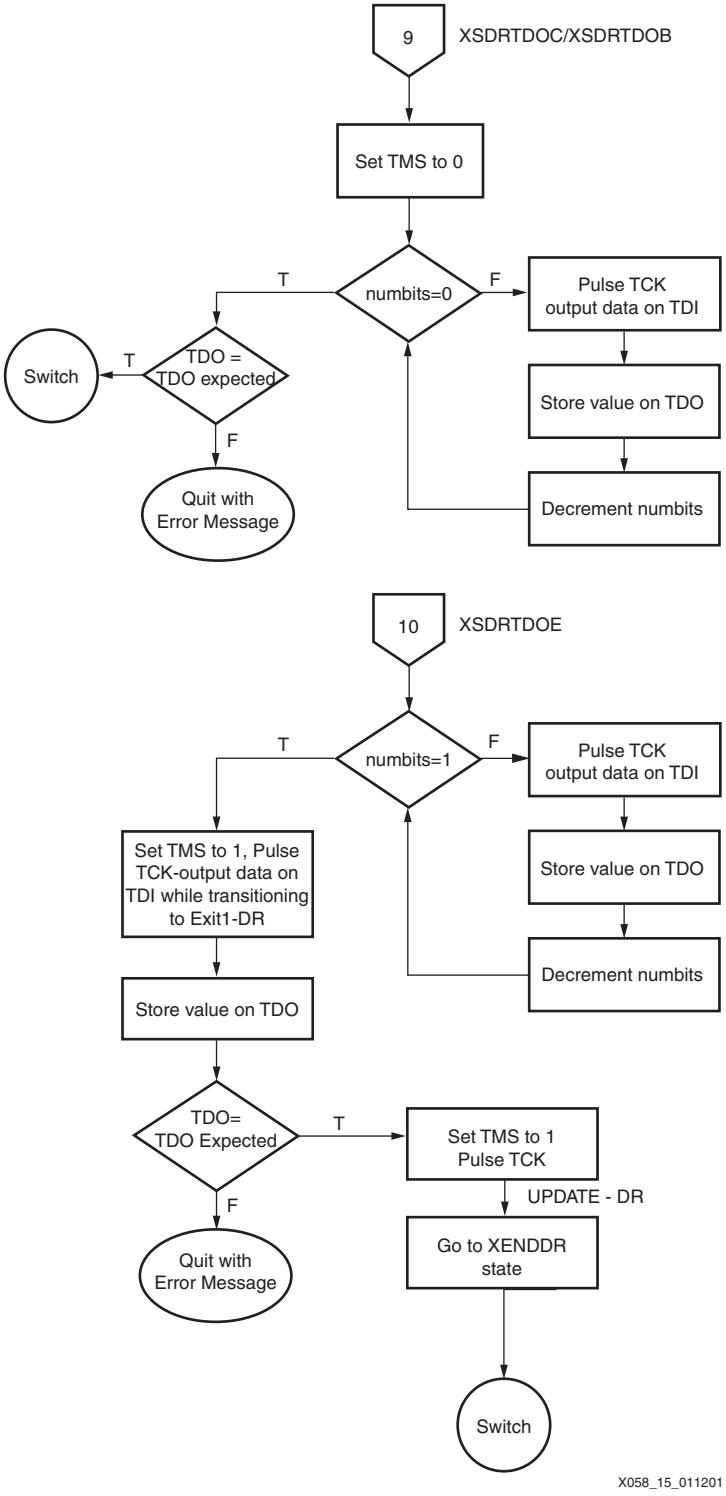


Figure 14: **Flow Chart for the ISP Controller Code (Continued)**

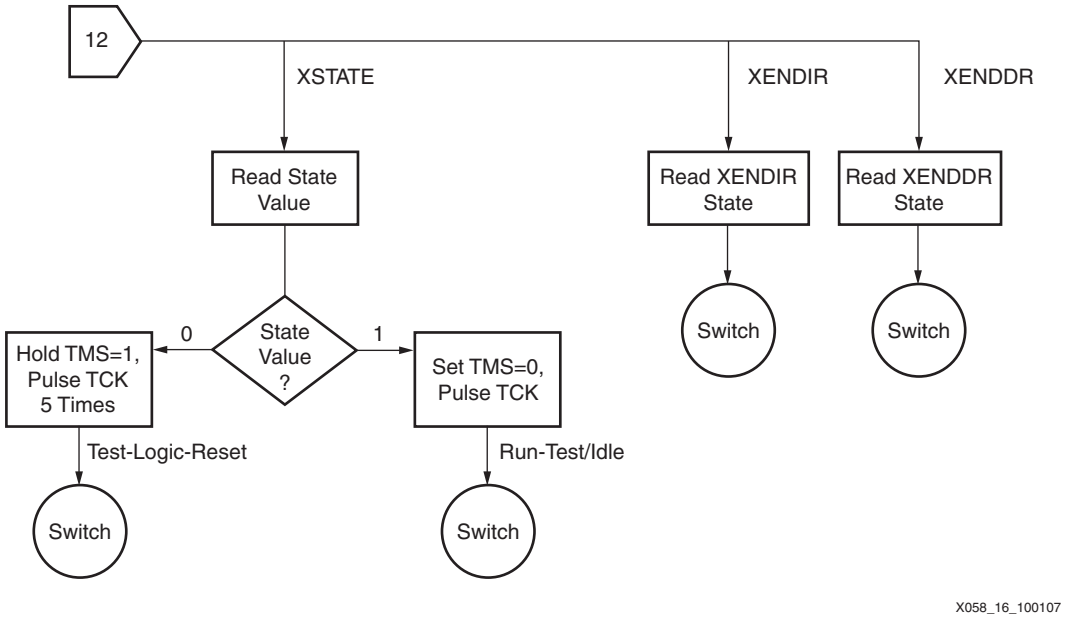


Figure 15: **Flow Chart for the ISP Controller Code (Concluded)**

TAP Timing

Figure 16 shows the timing relationships of the TAP signals. The C code running on the 8051 insures that the TDI and TMS values are driven at least two instruction cycles before asserting TCK. At that same time, TDO can be strobed.

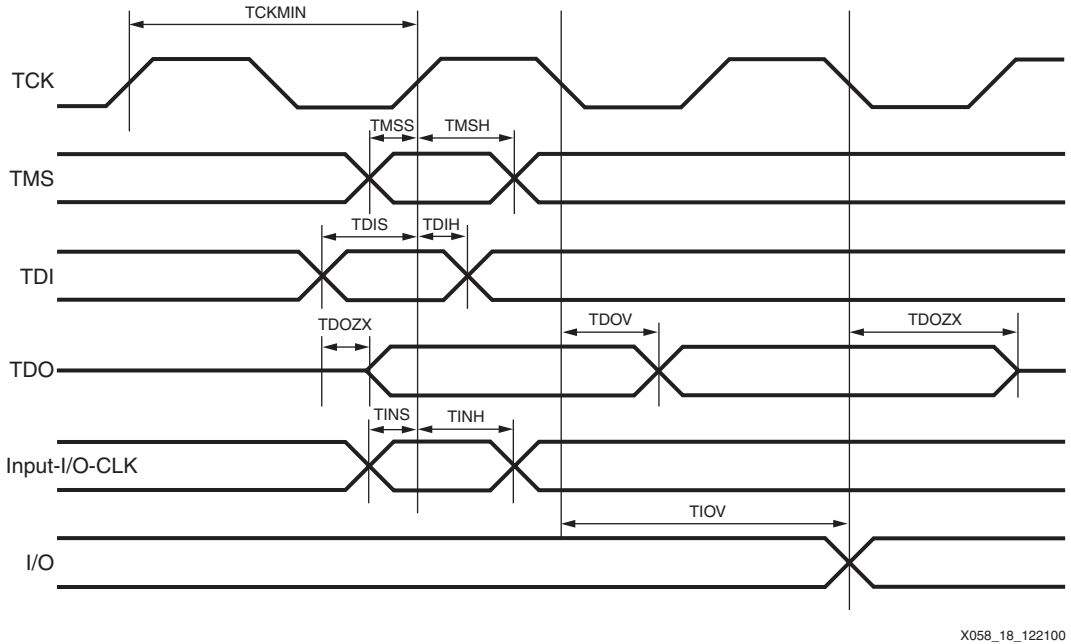


Figure 16: **Test Access Port Timing**

The key timing relationships include:

- TMS and TDI are sampled on the rising edge of TCK.
- A new TDO value appears after the falling edge of TCK.

The C code ensures proper TAP timing by:

- Updating TMS and TDI on the falling edge of TCK
- Sampling TDO after a sufficient delay following the falling edge of TCK.

Parts of the XSVF file specify wait times during which the device programs or erases the specified location or sector. Implementation of the wait timer can be accomplished either by software loops that depend on the processor's cycle time or by using the 8051's built-in timer function. In this design, timing is established through software loops in the ports.c file.TAP AC Parameters

Figure 17 shows the XC9500/XL/XV device programming flow.

Table 6 lists the XC9500 timing parameters for the TAP waveforms shown in Figure 16. For other device families, see the device family data sheet for TAP timing characteristics.

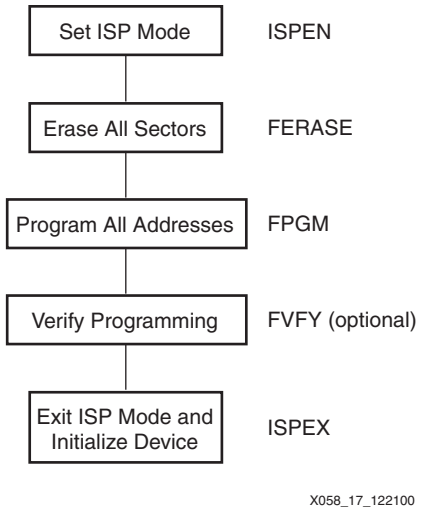


Figure 17: **XC9500/XL/XV Device Programming Flow**

Table 6: **XC9500 Test Access Port Timing Parameters (ns)**

Symbol	Parameter	Min	Max
TCKMIN	TCK Minimum Clock Period	100	
TMSS	TMS Setup Time	10	
TMSH	TMS Hold Time	10	
TDIS	TDI Setup Time	15	
TDIH	TDI Hold Time	25	
TDOZX	TDO Float-to-Valid Delay		35
TDOXZ	TDI Valid-to-Float Delay		35
TDOV	TDO Valid Delay		35
TINS	I/O Setup Time	15	
TINH	I/O Hold Time	30	
TIOV	EXTEST Output Valid Delay		55

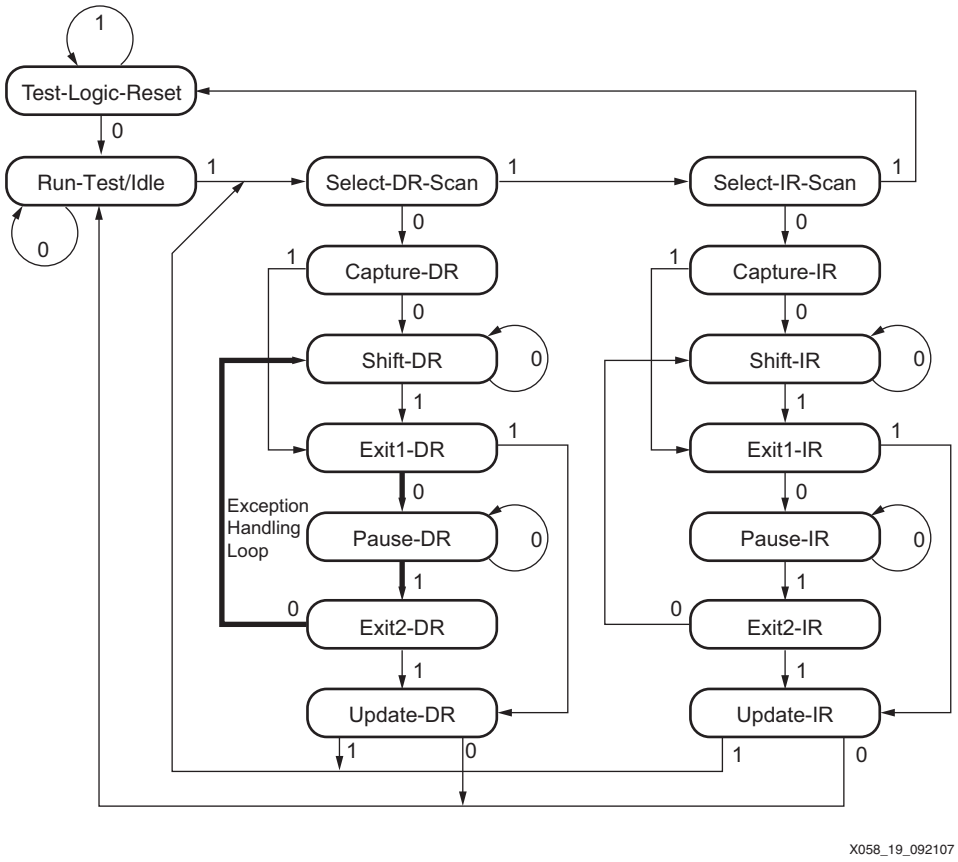
XC9500/XL/XV Programming Algorithm

This section describes the programming algorithm executed by the 8051 C code that reads the XSVF file; this code is contained in the `micro.c` file in "Appendix B: C-Code Listing," page 23. This information is valuable to users who want to modify the C code for porting to other microcontrollers.

The XSVF file contains all XC9500/XL/XV programming instructions and data. This allows the TAP driver code to be very simple. The 8051 interprets the XSVF instructions that describe the CPLD design and then outputs the TAP signals for programming (and testing) the XC9500/XL/XV device. The command sequence for device programming is shown in.

Exception Handling

Figure 18 shows the state diagram for the internal device programming state machine, as defined by the IEEE 1149.1 standard.



Notes:

1. The values shown adjacent to each transition represent the signal present at TMS during the rising edge of TCK.

Figure 18: **TAP State Machine Flow**

The C code drives the 1149.1 TAP controller through the state sequences to load data and instructions, and capture results. One of the key functions performed by the C code is the TAP controller state transition sequence that is executed when an XC9500/XL/XV program or erase operation needs to be repeated, which can occur on a small percentage of addresses. If a sector or address needs to be reprogrammed or re-erased, the device status bits return a value that is different from that which is predicted in the XSVF file. In order to retry the previous (failed) data, the following 1149.1 TAP state transition sequence is followed, if the TDO mismatch is identified at the EXIT1-DR state:

```
EXIT1-DR, PAUSE-DR, EXIT2-DR, SHIFT-DR, EXIT1-DR, UPDATE-DR, RUN-TEST/IDLE
```

The application then increments the previously specified XRUNTEST time by an additional 25 percent and waits for this amount of time in Run-Test/Idle. The effect of this state sequence is to re-apply the previous value rather than apply the new TDI value just shifted in.

This “exception handling loop” is attempted no more than N times. If the TDO value does not match after N attempts, the part is defective and a failure is logged. When the retry operation is successful, the algorithm shifts in the next XSDR data.

The recommended repeat value with iMPACT **`svf2xsvf -r`** (repeat) translation for XC9500/XL/XV devices is 16. The value for XC9500/XL/XV devices is 16.

XC4000 and Spartan/Spartan-XL Family Programming Algorithm

XC4000 Series devices can be configured through the Boundary-Scan pins. The basic procedure is as follows:

- Power up the FPGA with $\overline{\text{INIT}}$ held Low (or the $\overline{\text{PROGRAM}}$ pin Low for more than 300 ns followed by a High while holding $\overline{\text{INIT}}$ Low). Holding $\overline{\text{INIT}}$ Low allows enough time to issue the CONFIG command to the FPGA. The pin can be used as I/O after configuration if a resistor is used to hold $\overline{\text{INIT}}$ Low.
- Issue the CONFIG command to the TMS input to want to turn things red instead.
- Wait for $\overline{\text{INIT}}$ to go High.
- Sequence the Boundary-Scan Test Access Port to the SHIFT-DR state.
- Toggle TCK to clock data into TDI pin.

The user must account for all TCK clock cycles after INIT goes High, as all of these cycles affect the Length Count compare.

For more detailed information, refer to [Ref 2]. This application note also applies to XC4000E and XC4000X devices.

Virtex Series and Spartan-II/3/3E/3A Programming Algorithm

Virtex devices can be configured through the Boundary-Scan pins. Configuration through the TAP uses the special CFG_IN instruction. This instruction allows data input on TDI to be converted into data packets for the internal configuration bus.

The following steps are required to configure the FPGA through the Boundary-Scan port.

- Load the CFG_IN instruction into the Boundary-Scan instruction register (IR).
- Enter the Shift-DR (SDR) state.
- Shift a standard configuration bitstream into TDI.
- Return to Run-Test-Idle (RTI).
- Load the JSTART instruction into IR.
- Enter the SDR state (For Virtex-II devices, stay in the Run-Test/Idle state).
- Clock TCK for the length of the startup sequence.
- Return to RTI.
- Check the DONE pin status.

See [Ref 3] for details on Virtex configuration.

Notes:

1. The **`-fpga`** option is set in iMPACT when doing the XSVF translation for Virtex, XC4000, and Spartan devices.
2. The programming operation for each Virtex device ends by checking the DONE pin status. If multiple Virtex devices are to be configured and if the DONE pins of those devices are tied together, then the DONE pin does not go High until all the Virtex devices are configured. In this case, the check of the DONE pin status for the intermediate Virtex devices fail. To workaround this problem, the check on the DONE pin status for all but the last Virtex device must be removed from the SVF before translation to XSVF.

CoolRunner Programming Algorithm

The CoolRunner devices can be programmed through the Boundary-Scan pins. The basic procedure is as follows:

- Enter the device into ISP mode
- Erase the entire device
- Program all addresses
- Verify all addresses
- Exit the ISP mode and return to normal functional mode.

XC18V00 PROM Programming Algorithm

The XC18V00 devices can be programmed through the Boundary-Scan pins. The basic procedure is as follows:

- Enter the device into ISP mode
- Erase the entire device
- Program all addresses
- Apply global operation to refine programmed values.
- Verify all addresses
- Exit the ISP mode and return to normal functional mode.

Conclusion

Xilinx CPLDs and FPGAs are easily programmed by an embedded processor. Because they are 1149.1 compliant, system and device test functions can also be controlled by the embedded processor, in addition to programming. This capability opens new possibilities for upgrading designs in the field, creating user-specific features, and remote downloading of CPLD/FPGA programs.

Appendix A: XSVF File Merge Utility

mergexsvf File Merge Utility

This executable takes multiple XSVF files and merges them into a single XSVF file. When the files are merged, the XCOMPLETE commands are removed from the intermediate file images and a header is inserted between files that resets the parameters for the following commands: XSTATE, XENDIR, XENDDR, and XRUNTEST.

Usage:

```
mergexsvf [-d] [-v2] -o <output.xsvf> -i <input1.xsvf> -i <input2.xsvf>
[-i <inputN.xsvf>…]
```

Options:

-d – Delete pre-existing output file.

-i <inputN.xsvf> – Input files to be merged in the order listed.

-o <output.xsvf> – Merged output file.

-v2 – Generates an output file with intermediate headers that do not include the XSTATE, XENDIR, and XENDDR commands.

Note: The input XSVF files should be generated using the **-v2** option during the iMPACT svf2xsvf file conversion.

Example:

```
mergexsvf -d -o merged.xsvf -i xc9536xl.xsvf -i xc18v04.xsvf
```

Appendix B: C-Code Listing

The following files contain the C source code used to read an XSVF file and output the appropriate Test Access Port control bits:

C-Code Files

- `lenval.c` — This file contains routines for using the lenVal data structure.
- `micro.c` — This file contains the main function call for reading in a file from an EPROM and driving the JTAG signals.
- `ports.c` — This file contains the routines to output values on the JTAG ports, to read the TDO bit, and to read a byte of data from the EPROM.

Header Files

- `lenval.h` — This file contains a definition of the lenVal data structure and extern procedure declarations for manipulating objects of type lenVal. The lenVal structure is a byte oriented type used to store an arbitrary length binary value.
- `ports.h` — This file contains extern declarations for providing stimulus to the JTAG ports.

To compile this C code for a microcontroller, only four functions within the `ports.c` file need to be modified:

- setPort — Sets a specific port on the microcontroller to a specified value.
- readTDOBit — Reads the TDO port.
- readByte — Reads a byte of data from the XSVF file.
- waitTime — Pauses for a specified amount of time.

  Note: The waitTime function is called when the device is in Run-Test/Idle state to pause the system for the specified amount of time. For all device families other than the Virtex-II, TCK pulses are not required (but can occur) while the waitTime function is pausing the system. For the Virtex-II devices, the parameter to the waitTime function must be interpreted as a minimum number of TCK pulses to be generated. Typically, the Virtex-II devices require less than 25 TCK pulses.

  Caution! THE waitTime IMPLEMENTATION MUST PAUSE FOR AT LEAST THE SPECIFIED NUMBER OF MICROSECONDS. Verify that the waitTime implementation does not round small numbers down to zero time. Similarly, verify that the waitTime implementation does not overflow when given large numbers (for example, up to 140,000,000 microseconds for a Platform Flash XCF32P PROM).

The following is an example implementation of the waitTime function that is sufficient for all Xilinx devices except for Virtex-II devices (if a more accurate timing function than the standard sleep function, use it for improved programming performance.)

```
void waitTime( long microseconds )
{
    // Round up to the nearest millisecond
    sleep( ( microseconds + 999L ) / 1000L );
}
```

The following are code examples for implementations of the waitTime function that handle all the device families including the Virtex-II. For systems that can clock TCK at 1 MHz or faster, the waitTime function can be implemented so that it generates TCK pulses equivalent to the requested wait time:

```
void waitTime( long microseconds )
{
    // tckCyclesPerMicrosecond is a predetermined constant for your system
    long tckPulses = microseconds * tckCyclesPerMicrosecond;
    for ( long i = 0; i < tckPulses; ++i )
    {
        pulseTCK();
    }
}
```

For systems that can run TCK significantly slower than 1 MHz, the waitTime implementation needs to satisfy the Virtex-II requirement for the few TCK cycles (<25) that it needs, but the implementation should consider optimizing the wait time for the longer wait periods that other devices require:

```
void waitTime( long microseconds )
{
    if ( microseconds >= 50 )
    {
        // Round up to the nearest millisecond and
        //use standard sleep function
        sleep( ( microseconds + 999L ) / 1000L );
    }
    else  // satisfy Virtex-II TCK clock cycles
    {
        for ( long i = 0; i < microseconds;  ++i )
        {
            pulseTCK();
        }
    }
}
```

For help in debugging the code, a compiler switch called DEBUG_MODE is provided. If DEBUG_MODE is defined, the software reads from an XSVF file (which must be named prom.bit) and prints the debugging messages. A compile switch called XSVFSIM allows the designer to simulate the TAP outputs without a physical connection to the target device. Use the DEBUG_MODE with the XSVFSIM switch to view the simulated TAP signal values.

Appendix C: Dynamically Selecting Target Devices for Configuration

In the default configuration flow, the complete JTAG scan chain is defined in the iMPACT software. Designs are assigned to devices within the JTAG scan chain, and the devices to be configured are selected prior to the creation of the SVF file. The devices selected for configuration are called target devices. iMPACT generates an SVF file that contains a separate set of configuration commands and data for each target device. Target devices are configured sequentially, one device at a time. When a target device gets configured, the non-target devices are put into bypass mode. Each set of SVF commands and data for a target device contains an exact complement of bits corresponding to the bypassed, non-target devices. Thus, the exact assignment of designs and exact selection of target devices must be known in advance, because each SVF is built for a specific scan chain and specific selection of target devices.

The default configuration flow is inefficient for systems that use identical designs on multiple FPGAs or that use multiple combinations of designs for a set of FPGAs. For systems that configure multiple FPGAs with the same design, the SVF must still be created with separate sets of commands and data for each FPGA. That is, the design data is duplicated for each FPGA to be configured. For systems that use multiple combinations of designs across a set of FPGAs, SVF files must exist for each possible combination of design assignments. Again, design data is duplicated within the system. Because a one-to-one correspondence exists between the original SVF file and the corresponding XSVF file used in the embedded environment, the creating of inefficient SVF files equivalently affects the XSVF file storage requirements.

Using Dynamic Targeting to Reduce System Storage Requirements

To improve the data storage efficiency of these particular systems, a special version of the XSVF player is included in the XAPP058 download package. This special version of the XSVF player uses XSVF files built to configure just one device and supports the ability to dynamically target a given XSVF file to configure any compatible device in the scan chain. Only one XSVF file per design is required. In a system that uses identical designs on multiple FPGAs, a single XSVF (design) file can be reused to configure all of the FPGAs. In a system that uses multiple combinations of designs for a set of FPGAs, separate XSVF files corresponding to each design can be dynamically selected and targeted to the FPGAs.

The dynamic targeting feature reduces system storage requirements in the following systems:

- Systems in which FPGAs are configured with identical designs.
- Systems in which a set of FPGAs can be configured with multiple combinations of selected designs.

C-Code Files for the Dynamic Targeting XSVF Player

The files for this special version of the XSVF player are located in the `dynamic_target` directory from the download package. The dynamic_target directory contains two files: `micro_dynamic_target.c` and `micro_dynamic_target.h`. These two files are modified versions of the base `micro.c` and `micro.h` source files from the `src` directory in the download package. The code in the `micro_dynamic_target.c` file is modified to support dynamic selection of the device to be configured within a scan chain. The `micro_dynamic_target.h` file simply contains the declaration of the modified procedural interface that supports this dynamic targeting feature.

Substitute the dynamic_target files for the base `micro.c` and `micro.h` files in `src` directory to build an XSVF player that supports the dynamic targeting feature:

- Copy `dynamic_target\micro_dynamic_target.h` to `src\micro.h`
- Copy `dynamic_target\micro_dynamic_target.c` to `src\micro.c`

Building XSVF Files for Dynamic Targeting

An XSVF file that is used to configure a dynamically selected device at run-time must contain just the set of commands and data to configure a single, compatible device.

To create an XSVF file for dynamic targeting, use iMPACT to:

1. Define a scan chain that contains just the single device.
2. Assign the design file to the device in the scan chain.
3. Select the device as the operation target.
4. Generate the XSVF file that contains the program operation commands and data for the assigned design.

A separate XSVF file must be created for each design used to configure a device. These XSVF files are individually used to configure selected devices in the system.

A Primer on the Dynamic Targeting Feature

The basic commands within an XSVF file are designed to shift instruction and data bits through the JTAG scan chain into a target device. The commands in an XSVF file built for a single-device scan chain effectively shift the instruction and data bits directly into the JTAG ports of the target device. To dynamically retarget a single-device XSVF file to a specific device in a multi-device scan chain, the XSVF player must account for the shift registers of the non-target devices in the scan chain and insert the appropriate bits before or after the target device's instruction or data bit sets.

The IEEE Standard 1149.1 specifies the BYPASS instruction to consist of all one-bits and the BYPASS data register to be exactly one-bit wide. With this information, the exact bit patterns for the bypassed, non-target devices can be calculated. During an instruction shift, one-bits must be shifted into the instruction registers of all the bypassed, non-target devices. During a data shift, an extra data (zero) bit must be shifted into the bypass registers of all non-target devices.

Using the Special XSVF Player to Dynamically Select Target Devices

In the regular XSVF player, a pointer to the beginning of the XSVF data is first set. Then, the start function (xsvfExecute) is called to execute the XSVF data. The same flow applies to the special XSVF player with additional parameters that must be specified to the start function.

The primary function (xsvfExecute) that starts the special dynamic_target XSVF player is enhanced with five additional parameters. These parameters specify the number of leading and trailing instruction and data bits to be inserted before or after the main set of bits from the XSVF commands. An additional parameter is accepted that aligns Virtex configuration data to a 32-bit boundary (see [Ref 3] for additional information on the Virtex 32-bit configuration frame that imposes the 32-bit boundary requirement on the bitstream).

The enhanced xsvfExecute function is declared in the `micro_dynamic_target.h` file as follows:

```
int xsvfExecute(int iHir, int iTir, int iHdr, int iTdr, int iHdrFpga);
```

The parameters are described in Table 7.

Note: The 32-bit alignment issue applies only to the Virtex, Virtex-E, and Spartan-II device families.

Table 7: **XSVF Player Parameters**

Parameter	Name	Description
iHir	Header Instruction Register	The number of (one) bits to shift before the target set of instruction bits. These bits put the non-target devices after the target device into BYPASS mode. The iHir value must be equivalent to the sum of instruction register lengths for devices following the target device in the scan chain.
iTir	Trailer Instruction Register	The number of (one) bits to shift after the target set of instruction bits. These bits put the non-target devices before the target device into BYPASS mode. The iTir value must be equivalent to the sum of instruction register lengths for devices preceding the target device in the scan chain.
iHdr	Header Data Register	The number of (zero) bits to shift before the target set of data bits. These bits are placeholders that fill the BYPASS data registers in the non-target devices after the target device. The iHdr value must be equivalent to the sum of devices following the target device in the scan chain.
iTdr	Trailer Data Register	The number of (zero) bits to shift after the target set of data bits. These bits are placeholders that fill the BYPASS data registers in the non-target devices before the target device. The iTdr value must be equivalent to the sum of devices preceding the target device in the scan chain.
iHdrFpga	Header Data Register for the Virtex FPGA Commands	The number of (zero) bits to shift before the target set of Virtex FPGA data bits. These bits are used to align the configuration bitstream for Virtex devices to a 32-bit boundary. The iHdrFpga value must be equivalent to 32 minus the sum of devices preceding the target device in the scan chain. If no devices precede the target device, the value is zero. If the sum of devices is greater than 32, then the value must be 32 minus the modulo [32] of the sum of devices preceding the target device.

These parameters are equivalent to the HIR, TIR, HDR, and TDR commands in the SVF specification. See the SVF specification for further details:

http://www.asset-intertech.com/support/svf.html

From the given set of parameters, the micro_dynamic_target.c implementation automatically adds the necessary set of complementary bits to the XSVF commands to compensate for the bypassed devices in the scan chain.

Note: If all of the xsvfExecute parameters are equal to zero, then the special XSVF player functionality is equivalent to the base XSVF player that takes an XSVF file created for a fully specified scan chain! Thus, the special XSVF player with the dynamic targeting feature can be used in both the normal (fully-specified XSVF) and special (dynamic targeting) modes.

Dynamic Target Example

To configuring four Virtex 300E devices with identical designs using a single XSVF source file, the original XSVF file must be created using the instructions from "Building XSVF Files for Dynamic Targeting," page 25. Assuming the design for an XCV300E is located in the `design.bit` file, the XSVF file must be created as follows:

1. Define a scan chain in iMPACT with just the single XCV300E device.
2. Assign the `design.bit` file to the single instance of the XCV300E in the scan chain.
3. Select the XCV300E as the operation target.
4. iMPACT generates an XSVF file to program the XCV300E with the **`-fpga`** option.
1. Reset the XSVF program pointers to point to the beginning of the XSVF data.
2. To program device #1, call the xsvfExecute function with the following parameters:

   ```
   xsvfExecute(15, 0, 3, 0, 0)
   ```
3. Reset the XSVF program pointers to point to the beginning of the XSVF data.
4. To program device #2, call the xsvfExecute function with the following parameters:

   ```
   xsvfExecute(10, 5, 2, 1, 31)
   ```
5. Reset the XSVF program pointers to point to the beginning of the XSVF data.
6. To program device #3, call the xsvfExecute function with the following parameters:

   ```
   xsvfExecute(5, 10, 1, 2, 30)
   ```
7. Reset the XSVF program pointers to point to the beginning of the XSVF data.
8. To program device #4, call the xsvfExecute function with the following parameters:

   ```
   xsvfExecute(0, 15, 0, 3, 29)
   ```

Further examples of the code for the four device Virtex scan chain and a four device XC18V00 scan chain can be found in the `dynamic_target` directory of the download package.

An example XSVF player executable that provides this dynamic targeting capability is available under the `playxsvf\Release_D`**T** directory. This executable runs on a Windows 95/98/Me/NT/2000 PC with the Xilinx Parallel Cable III or Parallel Cable IV.

Appendix D: Binary to Intel Hex Translator

This appendix contains C code that can be used to convert XSVF files to Intel Hex format for downloading to an EPROM programmer. Most embedded processor code development systems can output Intel Hex for included binary files, and for those systems the following code is not needed. However, designers can use the following C code if the development system they are using does not have Intel Hex output capability.

Overview

The ISP controller described in this application note allows designers to program and test XC9500/XL CPLDs from information stored in EPROM. This information is saved in a binary XSVF file that contains both device programming instructions as well as the device configuration data. The 8051 microcontroller reads the EPROM (or EPROMs) that contain the XSVF file, converts the binary information to XC9500/XL compatible instructions and data, and outputs the programming information to the XC9500/XL device through a four-wire test access port.

After an XC9500/XL design is converted to XSVF format, the XSVF information is converted to Intel Hex format for downloading to an EPROM programmer. The resulting EPROMs, containing the CPLD programming information, can then be used in this ISP controller design.

```
/*
  This program is released to the public domain. It
prints a file to stdout in Intel HEX 83 format.
*/

#include <stdio.h>

#define RECORD_SIZE0x10/* Size of a record. */
#define BUFFER_SIZE 128

/*** Local Global Variables ***/

static char *line, buffer[BUFFER_SIZE];
static FILE *infile;

/*** Extern Functions Declarations ***/

extern char hex( int c );
extern void puthex( int val, int digits );

/*** Program Main ***/

main( int argc, char *argv[] )
{
  int c=1, address=0;
  int sum, i;
  i=0;
  /*** First argument - Binary input file ***/

  if (!(infile = fopen(argv[++i],"rb"))) {
    fprintf(stderr, "Error on open of file %s\n",argv[i]);
    exit(1);
  }

  /*** Read the file character by character ***/

  while (c != EOF) {
    sum = 0;
    line = buffer;
    for (i=0; i<RECORD_SIZE && (c=getc(infile)) != EOF; i++) {
      *line++ = hex(c>>4);
      *line++ = hex(c);
```

```
      sum += c; /* Checksum each character. */
    }
    if (i) {
      sum += address >> 8;/* Checksum high address byte.*/
      sum += address & 0xff;/* Checksum low address byte.*/
      sum += i;         /* Checksum record byte count.*/
      line = buffer;     /* Now output the line! */
      putchar(':');
      puthex(i,2);     /* Byte count. */
      puthex(address,4);     /* Do address and increment */
      address += i;     /*    by bytes in record.    */
      puthex(0,2);     /* Record type.    */
      for(i*=2;i;i--)     /* Then the actual data.    */
        putchar(*line++);
      puthex(0-sum,2);     /* Checksum is 1 byte 2's comp.*/
      printf("\n");
    }
  }
  printf(":00000001FF\n");/* End record. */
}

/* Return ASCII hex character for binary value. */

char
hex( int c )
{
  if((c &= 0x000f)<10)
    c += '0';
  else
    c += 'A'-10;
  return((char) c);
}

/* Put specified number of digits in ASCII hex. */

void
puthex( int val, int digits )
{
  if (--digits)
    puthex(val>>4,digits);
  putchar(hex(val & 0x0f));
}
```

References

1. XAPP503, *SVF and XSVF File Formats for Xilinx Devices*.
2. XAPP017, Boun*dary Scan in XC4000 Devices*.
3. XAPP139, *Configuration and Readback of Virtex FPGAs Using JTAG Boundary-Scan.*

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/15/01	3.0	Revised Xilinx release.
06/25/04	3.1	Minor changes made.
03/11/05	3.2	Updated for Platform Flash PROMs.
10/01/07	4.0	• Updated template. • Updated document for ISE iMPACT 9.2i support. • Other minor edits and changes made.

Notice of Disclaimer