XC9500 In-System Programming Using an Embedded Microcontroller

XAPP058 January, 1998 (Version 1.2) Application Note

Summary

The XC9500 high performance CPLD family provides 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, eliminating the need to re-tool PC boards. By using an embedded controller to program these CPLDs from an on-board RAM or EPROM, designers can easily upgrade, modify, and test designs, even in the field.

Xilinx Family

XC9500

Introduction

The XC9500 CPLD family combines 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, 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 XC9500 CPLDs 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 the EZTag™ software that automatically reads standard JEDEC device programming files and converts them to SVF format which contains both data and programming instructions for the CPLDs. These files are then converted to a compact binary format (XSVF) and can be stored in the on-board EPROM. The 8051 microcontroller interprets the XSVF information and generates the programming instructions, data, and control signals for the XC9500 CPLD.

By using a simple IEEE 1149.1 (JTAG) interface, XC9500 CPLDs are easily programmed and tested without using expensive hardware. Multiple devices can be daisy-chained, permitting a single 4-wire Test Access Port (TAP) to control any number of XC9500 CPLDs or other JTAG-compatible devices.

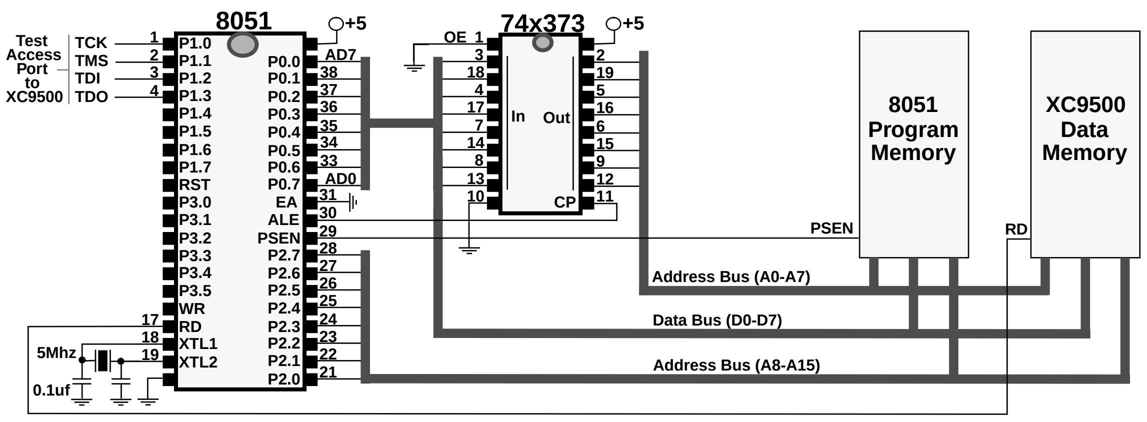

Figure 1: ISP Controller Schematic

Programming XC9500 CPLDs

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. XC9500 CPLDs accept programming and JTAG boundary-scan test instructions in SVF format, via the TAP. The timing for these TAP signals is shown in Figure 5.

The EZTag software automatically converts standard JEDEC programming files into SVF format. However, the SVF format is ASCII which is inefficient for embedded applications due to its memory requirements. Therefore, to minimize the memory requirements, SVF is converted into a more compact (binary) format called XSVF. In this design, an 8051 C-code algorithm interprets the XSVF file and provides the required JTAG TAP stimulus to the CPLD, performing the programming and (optional) test operations which were originally specified in the SVF file.

Note: For a description of the SVF and XSVF commands and file formats, see Appendix A and B.

The flow for creating the programming files that are used with this design, is shown in Figure 2.

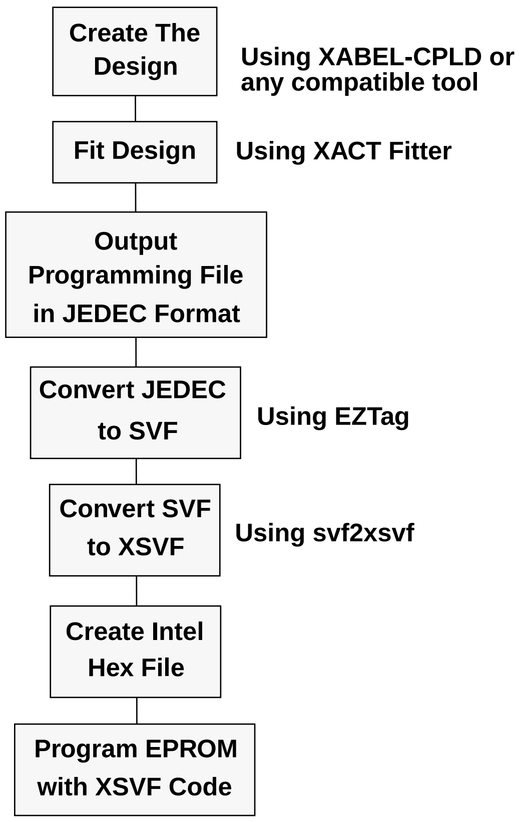

Figure 2: Program Flow

JTAG Instruction Summary

XC9500 devices accept both programming and test instructions via the JTAG TAP. The JTAG commands used for programming and functional test are:

- INTEST - Isolates the device from the system, applies test vectors to the device input pins, and captures the results from the device output pins.
- ISPEN - Enables the ISP function in the XC9500 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.
- 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.

The following instructions are also available in the XC9500 devices but are not used for programing or functional test:

- 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.
- USERCODE - Returns a 32-bit user-programmable code that can be used to store version control information or other user-defined variables.
- HIGHZ - Causes all device pins to float to a high impedance state.

Creating an SVF File Using EZTag

This procedure shows how to create an SVF file; it assumes that the Xilinx XACT version 6.0.0 software, or newer, is being used, which includes the XC9500 fitter and the EZTag software.

1. Create the design using XABEL-CPLD or any compatible third-party design entry tool.
2. Fit the design and save it to a JEDEC output file.
3. Invoke the EZTag software from the XACT command line using the following command:

```
eztag -svf
```

The following message appears:

```
Xilinx (R) EZTAG XC9500-CPLD-6.0.0 - JTAG
Boundary-Scan Download

Copyright (C) Xilinx Inc. 1991-1995. All
Rights Reserved.

----------------------------------------------------------------

SVF GENERATION MODE.
EZTAG ?
```

4. At the **EZTAG ?** prompt type the following command:

   part ***deviceType1*****:*****designName1***
   deviceType2**:*****designName2***
   deviceTypeN**:*****designNameN*** **<CR>**

   deviceType is the name of the BSDL file for that device and ***designName*** is the name of the design to translate into SVF. Multiple *deviceType:designName* pairs are separated by spaces. For example:

   ```
   part xc95108:abc12 xc95144:wxyz
   ```

   This command defines the JTAG device chain, from one to any number of devices. The parts specified in the **part** command should be arranged in order beginning with the first device to receive TDI and ending with the last device to output TDO.
   Note: For any non-XC9500 part in the JTAG chain make certain that the BSDL file for the specified part is available along the XACT path and is called *deviceType*.bsd.

5. Enter any one of the following commands:

- **erase *designName*** — generates an SVF file for the bit sequence needed to erase the specified part.
- **verify *designName* [-j *jedecFileName*]** — generates an SVF file that specifies the bit sequence to read back the device contents and compare it against the contents of the specified JEDEC file.
- **program [-v] *designName* [-j *jedecFileName*]** — generates an SVF file that specifies the bit sequence to program the specified part from a JEDEC file named *designName*.jed (or alternately, the JEDEC file name specified after the "-j"). The program command option add the following functionality:
  -v — Follow up the programming operation with a read-back verification against the contents of the JEDEC file.

6. Exit EZTag by entering the following command:

   ```
   quit
   ```

   NOTE: The SVF file will be named *designName*.svf, and will be created in the current working directory (the directory in which EZTAG is being run). Consecutive operations on the same *designName* file will overwrite the SVF file each time. The SVF file contains all data and commands necessary to perform the specified function.

7. Use the svf2xsvf tool to translate the ASCII SVF file into a more compact binary XSVF format. XSVF files are approximately 80% smaller than SVF files. To perform the conversion, enter the following command at the system prompt:

   svf2xsvf ***svf_file_name xsvf_file_name***

   Note: The svf2xsvf translator supports only SDR, SIR, and RUNTEST instructions in the source SVF file.

The XSVF file now contains both the programming data and instructions, ready for use by the 8051 microcontroller.

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 will automatically convert included binary files to the appropriate format. Public domain file conversion software is also available, as shown in Appendix D.

Software Limitations

EZTAG can generate SVF files only for devices for which JEDEC files can be created. Designers should verify that the development software they are using can create JEDEC files for the specific devices they intend to use.

The current software can only generate SVF files for operations on one part at a time. If there are several parts to be programmed, additional program commands must be executed — one for each part, creating multiple SVF files. In each SVF file, one device will be programmed while the others are held in bypass mode.

Hardware Design

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

Hardware Design Description

The 8051 allows 64K of program and 64K of data space; much more than is needed in this application. However the ability to separate address and data space is used to simplify the addressing scheme.

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 1 shows the estimated EPROM capacity needed to contain both the 8051 code and the XC9500 programming data. The XSVF file sizes are shown for an erase and program operation.

Table 1: XSVF File Sizes

Device Type	XSVF File Size	C-Code	Total
XC9536	5194	7k	12K
XC9572	11674	7k	19K
XC95108	19598	7k	27K
XC95144	12960 (estimated)	7k	20K
XC95216	26390	7k	33K
XC95288	34560 (estimated)	7k	42K

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 standalone ISP controller. However, it is also possible to apply these techniques to microcontrollers that may 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 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 Figure 5.

Using a different microcontroller would require 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 will not need to be modified.

RAM can be used instead of the EPROM in this design. This would allow the XC9500 devices to be programmed and tested remotely via modem, using remote control software written by the user.

Debugging Suggestions

The following suggestions may be helpful in testing this design:

- View the contents of the XSVF file using the xsvf2ascii converter. This will decode the binary file and display the XSVF data and instructions. To run this converter, enter the following command at the system prompt:

  ```
  xsvf2ascii
  ```

- Change the **#define DEBUG_MODE 0** to **#define DEBUG_MODE 1** in the ports.h file to see the calculated values of the TDI and TMS ports on the rising edge of TCK, when the code is compiled. Use this to verify the functionality of the C-code if it is ported to a different microcontroller. (See Appendix C for more information.)
- Use the ASCII text output, generated by xsvf2ascii, to verify that the bit sequence output of the microcontroller is correct.
- 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.

Firmware Design

The flow chart for the C-Code is shown in Figure 3. This code continuously reads the instructions and arguments from the XSVF file contained in the XC9500 program data EPROM and branches in one of three ways based on the three possible XSVF instructions (XRUNTEST, XSIR, XSDR) as described in Appendix B.

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 will stay 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 has been output 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 will be 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 has been output 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 6. 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 (which was originally set in the XRUNTEST instruction).

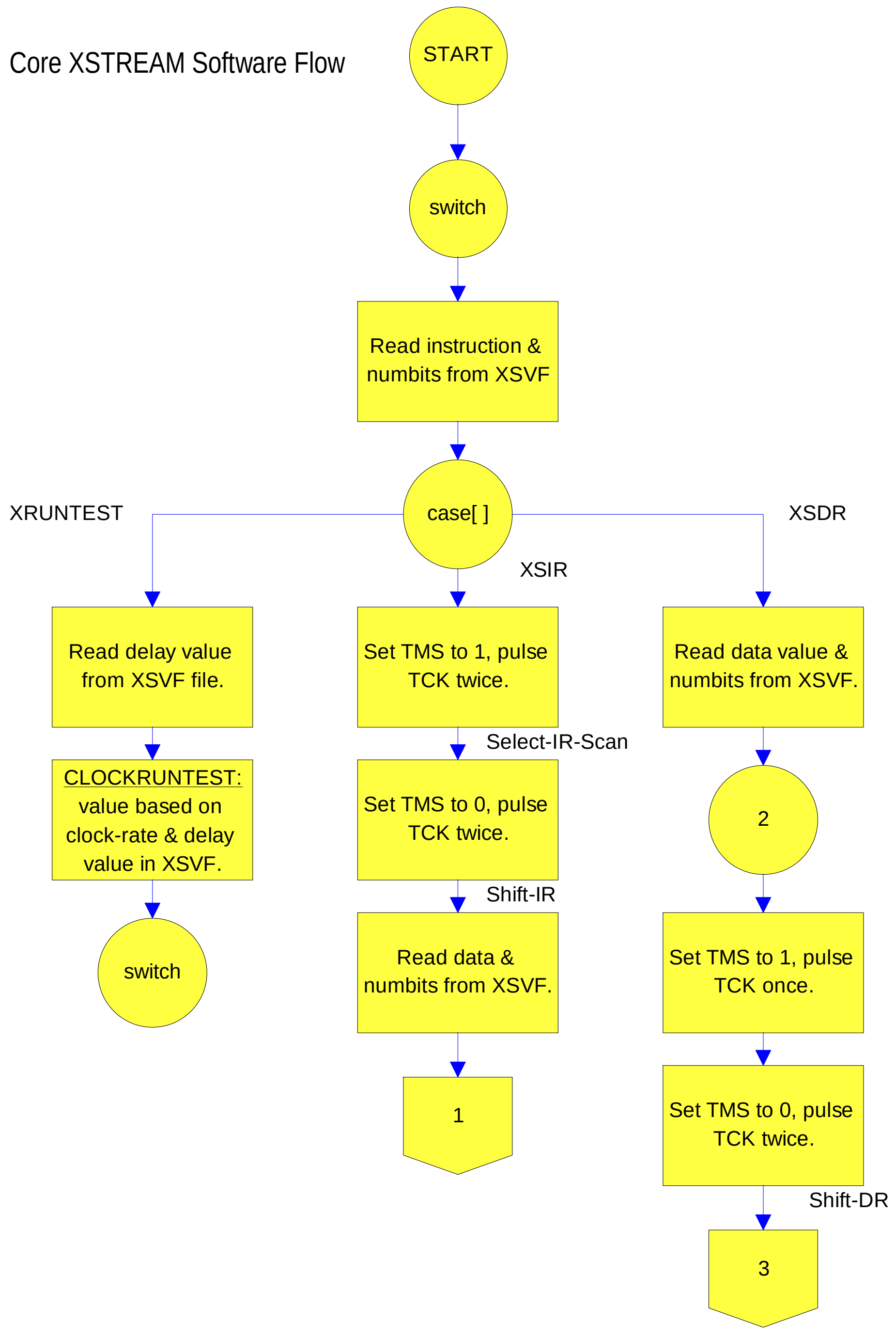
Core XSTREAM Software Flow
START
switch
Read instruction & numbits from XSVF
case[ ]
XRUNTEST
XSIR
XSDR
Read delay value from XSVF file.
CLOCKRUNTEST: value based on clock-rate & delay value in XSVF.
switch
Set TMS to 1, pulse TCK twice.
Select-IR-Scan
Set TMS to 0, pulse TCK twice.
Shift-IR
Read data & numbits from XSVF.
1
Read data value & numbits from XSVF.
2
Set TMS to 1, pulse TCK once.
Set TMS to 0, pulse TCK twice.
Shift-DR
3

Figure 3: Flow Chart for the ISP Controller Code

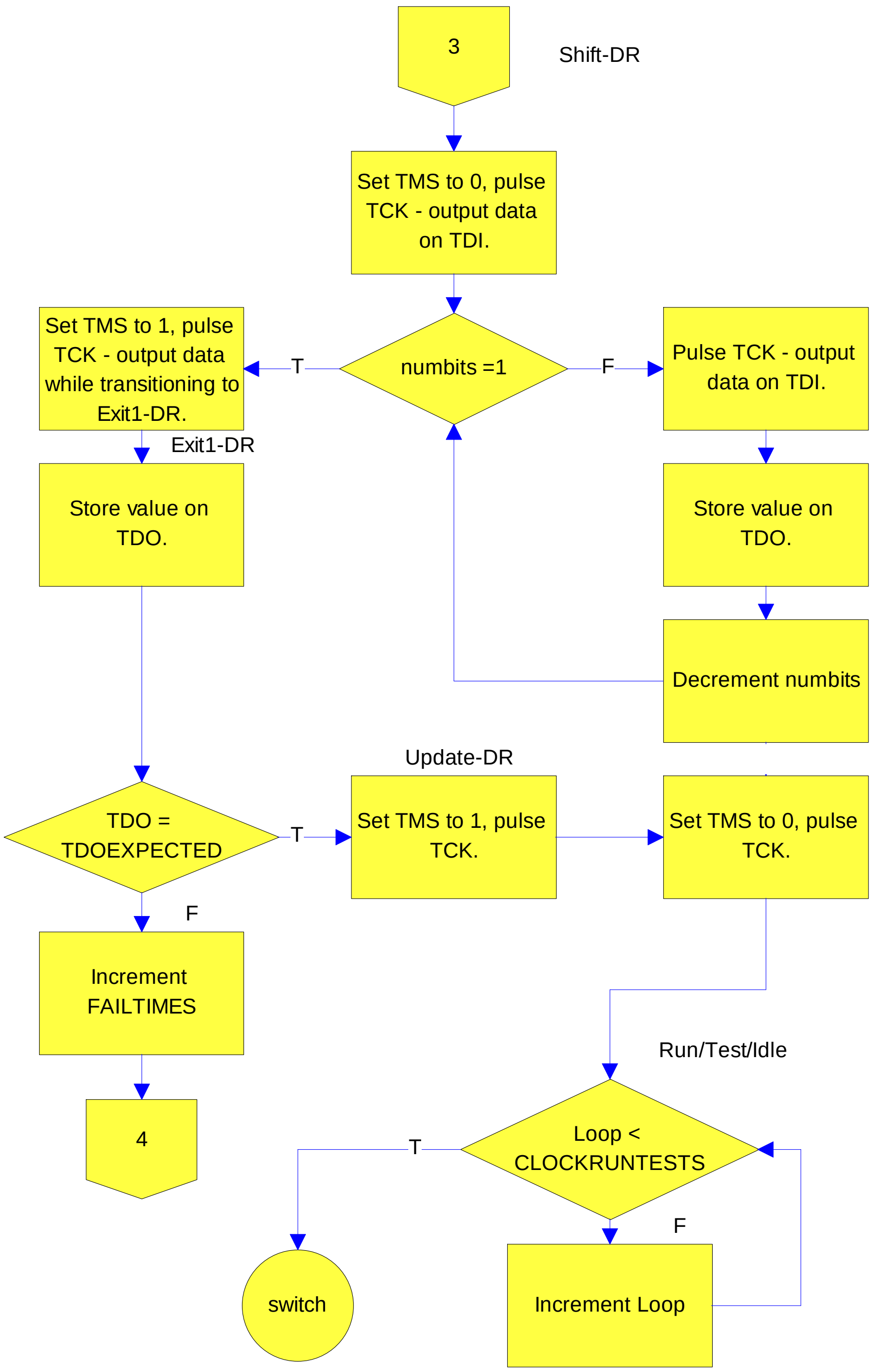

Figure 3: Continued

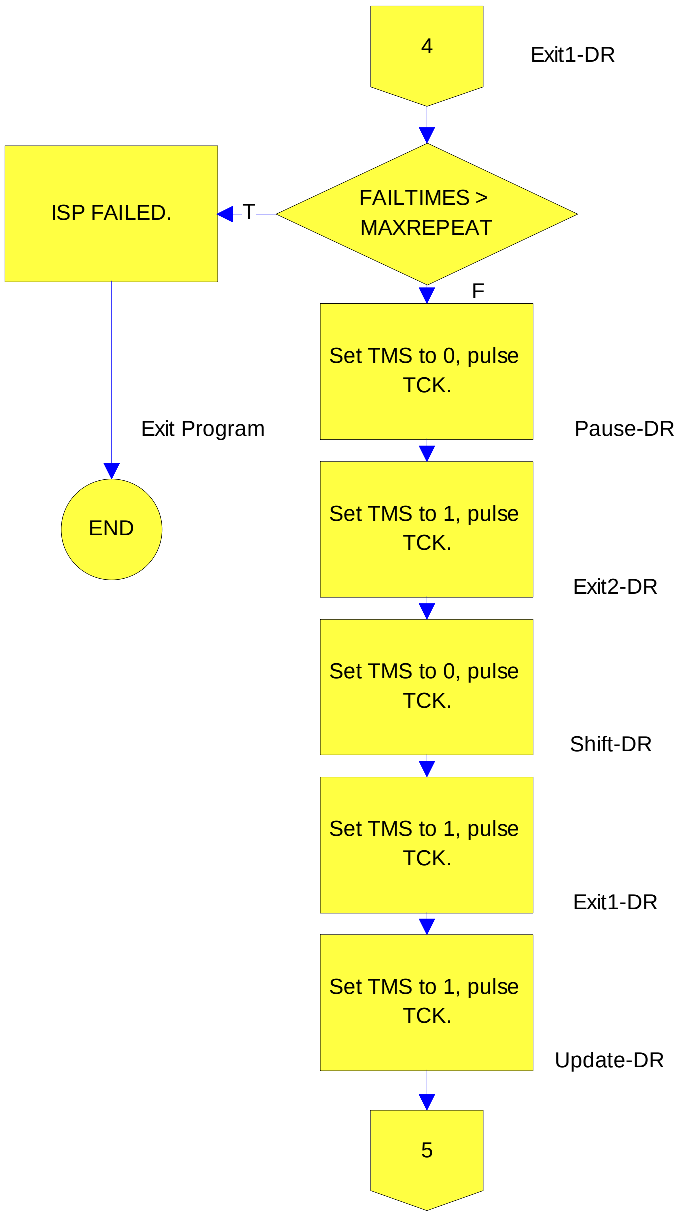

Figure 3: Continued

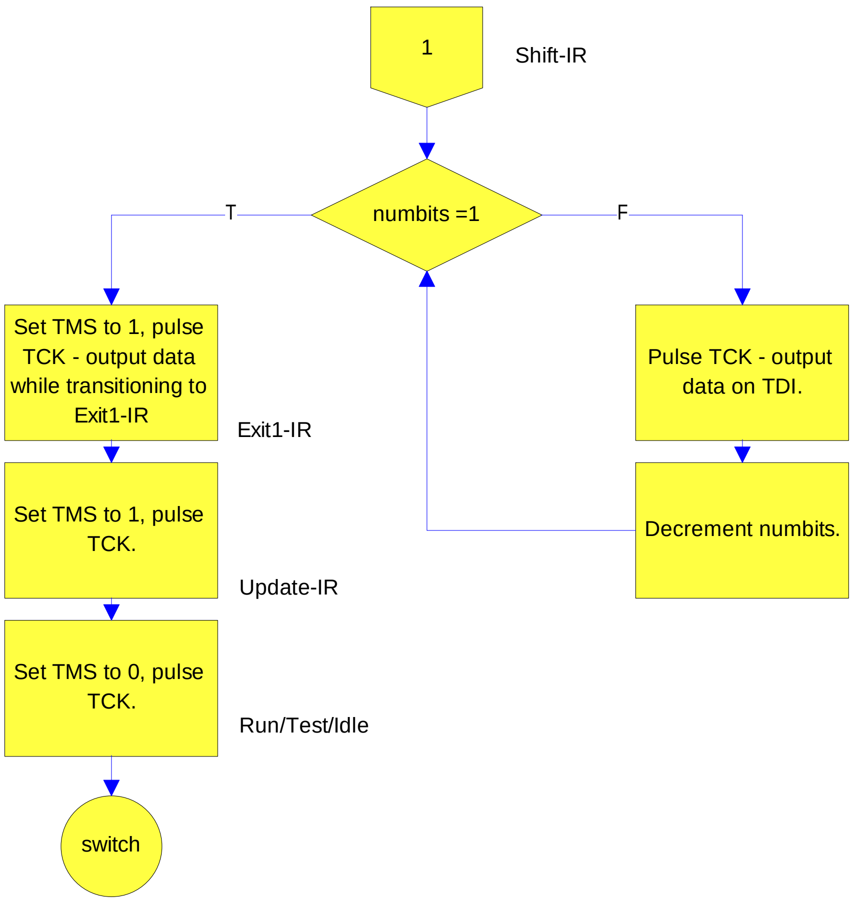

Figure 3: Continued

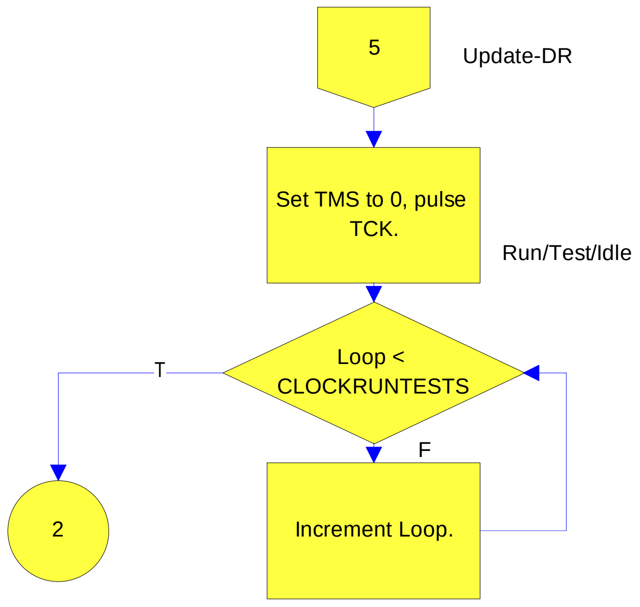

Figure 3: Continued

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 XC9500 CPLD program memory resides in the 8051 data block and is enabled by the $\overline{\text{RD}}$ signal.

Port Map

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

Table 2: 8051 Port 1 Mapping

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

TAP Timing

Figure 5 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.

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

Table 3 shows the timing parameters for the TAP waveforms, shown in Figure 5.

Table 3: 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 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 C. This information is valuable to users who want to modify the C-code for porting to other microcontrollers.

The XSVF file contains all XC9500 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 device. The command sequence for device programming is shown in Figure 4.

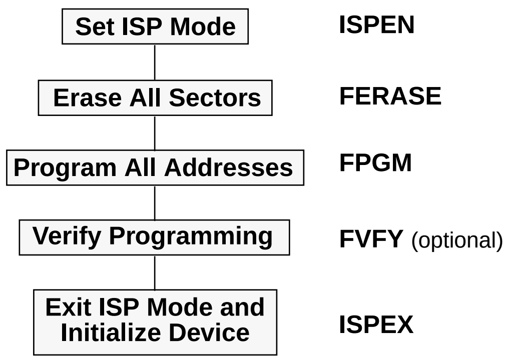

Figure 4: Device Programming Flow

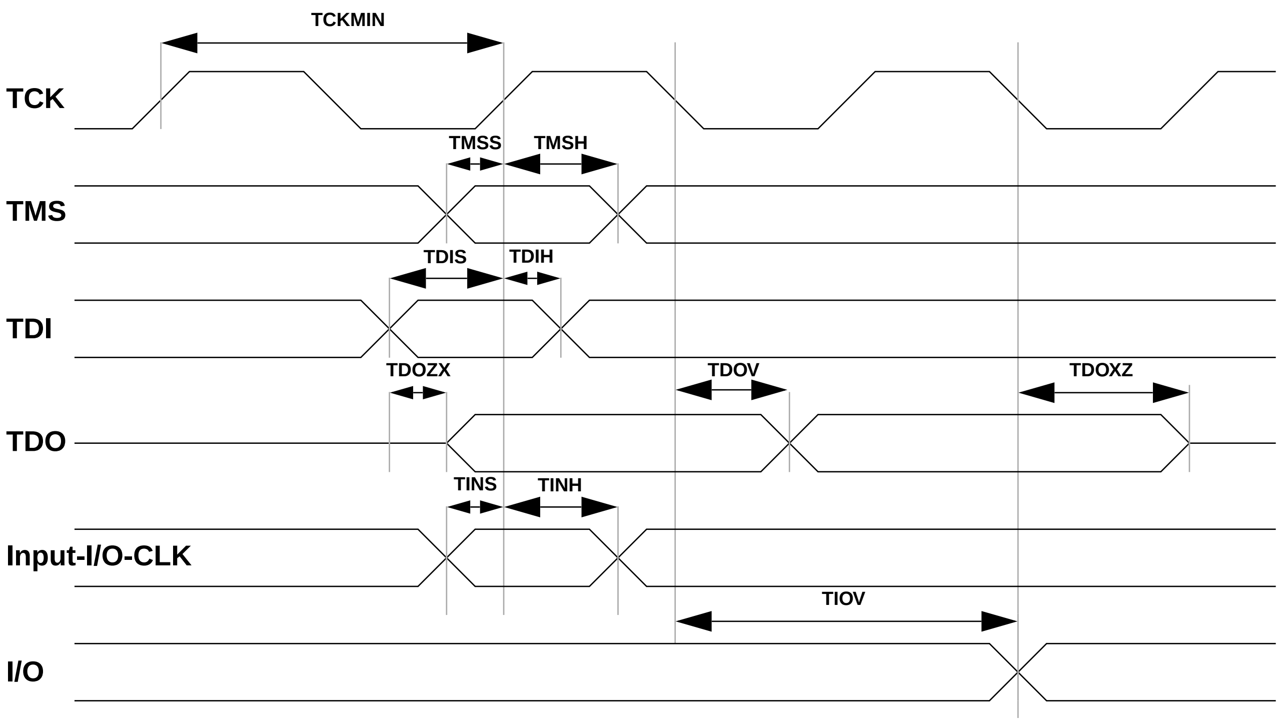

Figure 5: Test Access Port Timing

Exception Handling

Figure 6 shows the state diagram for the internal device programming state machine, as defined by the IEEE 1149.1 standard. 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 a program or erase operation needs to be repeated, which may occur on a small percentage of addresses. If a sector or address needs to be re-programmed 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 waits for the amount of time that was previously specified by XRUNTEST. The effect of this state sequence is to re-apply the previous value rather than apply the new TDI value that was just shifted in.

This "exception handling loop" is attempted no more than 32 times. If the TDO value does not match after 32 attempts, the part is defective and a failure is logged. When the retry operation is successful, the algorithm shifts-in the next XSDR data.

Conclusion

XC9500 CPLDs are easily programmed by an embedded processor. And, because the XC9500 family is 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 programs.

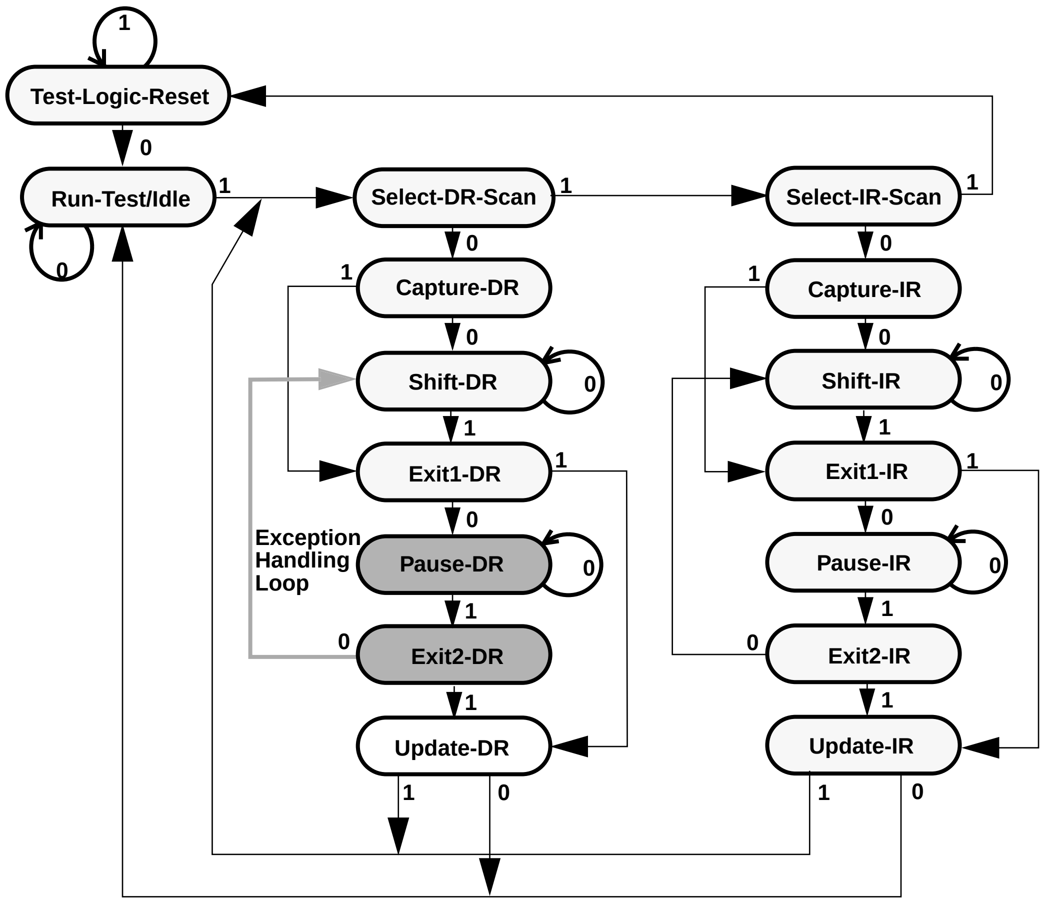

Figure 6: TAP State Machine Flow

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

Appendix A

SVF File Format for the XC9500 Family

SVF Overview

This appendix describes the Serial Vector Format syntax, as it applies to the XC9500 family; only those commands and command options that apply to XC9500 devices are described. An SVF file is the media for exchanging descriptions of high-level IEEE 1149.1 bus operations which consist of scan operations and movements between different stable states on the 1149.1 state diagram (as shown in Figure 6). SVF does not explicitly describe the state of the 1149.1 bus at every Test Clock (TCK).

An SVF file contains a set of ASCII statements. Each statement consists of a command and its associated parameters, terminated by a semicolon. SVF is case sensitive, and comments are indicated by an exclamation point (!).

Scan data within a statement is expressed in hexadecimal and is always enclosed in parenthesis. The scan data cannot specify a data string that is larger than the specified bit length; the Most Significant Bit (MSB) zeros in the hex string are not considered when determining the string length. The bit order for scan data defines the LSB (rightmost bit) as the first bit scanned into the device for TDI and SMASK scan data, and is the first bit scanned out for TDO and MASK data.

SVF Commands

The following SVF Commands are supported by the XC9500 Family:

- SDR (Scan Data Register).
- SIR (Scan Instruction Register).
- RUNTEST.

For each of the following command descriptions:

- The parameters are mandatory.
- Optional parameters are enclosed in brackets ([ ]).
- Variables are shown in *italics*.
- Parenthesis "( )"are used to indicate hexadecimal values.
- A scan operation is defined as the execution of an SIR or SDR command and any associated header or trailer commands.

SDR, SIR

SDR *length* TDI (*tdi*) SMASK (*smask*)
[TDO (*tdo*) MASK (*mask*)];

SIR *length* TDI (*tdi*) TDO SMASK (*smask*);

These commands specify a scan pattern to be applied to the target scan registers. The SDR command (Scan Data Register) specifies a data pattern to be scanned into the target device Data Register. The SIR command (Scan Instruction Register) specifies a data pattern to be scanned into the target device Instruction Register.

Prior to scanning the values specified in these commands, the last defined header command (HDR or HIR) will be added to the beginning of the SDR or SIR data pattern and the last defined trailer command (TDR or TIR) will be appended to the end of the SDR or SIR data pattern.

Parameters:

length — A 32-bit decimal integer specifying the number of bits to be scanned.

[TDI (*tdi*)] — (optional) The value to be scanned into the target, expressed as a hex value. If this parameter is not present, the value of TDI to be scanned into the target device will be the TDI value specified in the previous SDR/SIR statement. If a new scan command is specified, which changes the length of the data pattern with respect to a previous scan, the TDI parameter must be specified, otherwise the default TDI pattern is undetermined and is an error.

[TDO (*tdo*)] — (optional) The test values to be compared against the actual values scanned out of the target device, expressed as a hex string. If this parameter is not present, no comparison will be performed. If no TDO parameter is present, the MASK will not be used.

[MASK (*mask*)] — (optional) The mask to be used when comparing TDO values against the actual values scanned out of the target device, expressed as a hex string. A "0" in a specific bit position indicates a "don't care" for that position. If this parameter is not present, the mask will equal the previously specified MASK value specified for the SIR/SDR statement. If a new scan command is specified which changes the length of the data pattern with respect to a previous scan, the MASK parameter must be specified, otherwise the default MASK pattern is undefined and is an error. If no TDO parameter is present, the MASK will not be used.

[SMASK (*smask*)] — (optional) Specifies which TDI data is "don't care", expressed as a hex string. A "0" in a specific bit position indicates that the TDI data in that bit position is a "don't care". If this parameter is not present, the mask will equal the previously specified SMASK value specified for the SDR/SIR statement. If a new scan command is specified which changes the length of the data pattern with respect to a previous scan, the SMASK parameter must be specified, otherwise the default SMASK pattern used is undefined and is an error. The SMASK will be used even if the TDI parameter is not present.

Example:

```
SDR 27 TDI (008003fe) SMASK (07ffffff)
TDO (00000003) MASK (00000003) ;
```

```
SIR 16 TDO (ABCD);
```

RUNTEST

RUNTEST *run_count* TCK;

This command forces the target 1149.1 bus to the Run-Test/Idle state for a specific number of microseconds, then moves the target device bus to the IDLE state. This is used to control RUNBIST operations in the target device.

Parameters:

run_count — The number of TCK clock periods that the 1149.1 bus will remain in the Run Test/Idle state, expressed as a 32 bit unsigned number.

Example:

```
RUNTEST 1000 TCK;
```

```
! Begin Test Program

TRST OFF;                !disable test reset line
ENDIR IDLE;              !End IR scan in IDLE
HIR
HDR 16 TDI (FFFF) TDO (FFFF) MASK (FFFF); !16 bit DR Header
TIR
TDR
SIR
SDR
STATE
RUNTEST
!End test program
```

Figure 7: Sample SVF File

Appendix B

XSVF File Format and Conversion Utilities

This appendix includes the following reference information:

- **The XSVF Commands** — The instructions that are supported, their arguments, and definitions.
- **The svf2xsvf Utility** — Converts the standard SVF file format to the more compact binary XSVF format.
- **The xsvf2ascii Utility** — Converts the XSVF file format to ascii text for debugging purposes.

XSVF Commands

The following commands describe the 1149.1 operations in a way that is similar to the SVF syntax. The key difference between SVF and XSVF is that the XSVF file format affords better data compression and therefore produces smaller files.

The format of the XSVF file is a one byte instruction followed by a variable number of arguments (as described in the command descriptions below). The binary (hex) value for each instruction is shown in Table 4:

Table 4: Binary Encoding of XSVF Instructions

XSVF Instruction	Binary Encoding (hex)
XCOMPLETE	0x00
XTDOMASK	0x01
XSIR	0x02
XSDR	0x03
XRUNTEST	0x04
XREPEAT	0x07
XSDRSIZE	0x08
XSDRTDO	0x09
XSETSDRMASKS	0x0a
XSDRINC	0x0b

XTDOMASK

```
XTDOMASK value<"length" bits>
```

XTDOMASK sets the TDO mask which masks the value of all TDO values from the SDR instructions. Length is defined by the last XSDRSIZE instruction. XTDOMASK may be used multiple times in the XSVF file if the TDO mask changes for various SDR instructions.

Example:

```
XTDOMASK 0x00000003
```

This example defines that TDOMask is 32 bits long and equals 0x00000003

XREPEAT

```
XREPEAT times<1 byte>
```

Defines the number of times that TDO will be tested against the expected value before the ISP operation will be considered a failure. By default, a device may fail an XSDR instruction 32 times before the ISP operation is terminated as a failure. This instruction is optional.

Example:

```
XREPEAT 0x0f
```

This example sets the command repeat value to 15.

XRUNTEST

```
XRUNTEST time<4 bytes>
```

Defines the amount of time (in microseconds) the device should sit in the Run-Test/Idle state after each visit to the SDR state.

Example:

```
XRUNTEST 0x00000fa0
```

This example specifies an idle time of 4000 microseconds.

XSIR

```
XSIR length<1 byte> TDIValue<"length"
bits>
```

Go to the Shift-IR state and shift in the TDIValue.

Example:

```
XSIR 0x08 0xec
```

XSDR

```
XSDR TDIValue<"length" bits>
```

Go to the Shift-DR state and shift in TDIValue; compare the TDOExpected value from the last XSDR instruction against the TDO value that was shifted out (use the TDOMask which was generated by the last XTDOMASK instruction). Length comes from the XSDRSIZE instruction.

If the TDO value does not match TDOExpected, return to the Run-Test/Idle state again, and wait the amount of time last specified by the XRUNTEST command, then try the SIR instruction again. If TDO is wrong more than the maximum number of times specified by the XREPEAT instruction, then the ISP operation will be determined to have failed.

Example:

```
XSDR 02c003fe
```

XSDRSIZE

```
XSDRSIZE length<4 bytes>
```

Specifies the length of all XSDR/XSDRTDO records that follow.

Example:

```
XSDRSIZE 0x0000001b
```

This example defines the length of the following XSDR/ XSDRTDO arguments to be 27 bits (4 bytes) in length.

XSDRTDO

```
TDIValue<"length" bits>
TDOExpected<"length" bits>
```

Go to the Shift-DR state and shift in TDIValue; compare the TDOExpected value against the TDO value that was shifted out (use the TDOMask which was generated by the last XTDOMASK instruction). Length comes from the XSDRSIZE instruction.

If the TDO value does not match TDOExpected, return to the Run-Test/Idle state again, and wait the amount of time last specified by the XRUNTEST command, then try the SIR instruction again. If TDO is wrong more than the maximum number of times specified by the XREPEAT instruction, then the ISP operation will be determined to have failed.

The TDOExpected Value will be used in all successive XSDR instructions until the next XSDR instruction is given.

Example:

```
XSDRTDO 0x000007fe 0x00000003
```

For this example, go to the Shift-DR state and shift in 0x000007fe. Perform a logical AND on the TDO shifted out and the TDOMASK from the last XTDOMASK instruction and compare this value to 0x00000003.

XSETSDRMASKS

```
XSETSDRMASKS addressMask<"length" bits>
dataMask<"length" bits>
```

Set SDR Address and Data Masks. The address and data mask of future XSDRINC instructions are indicated using the XSETSDRMASKS instructions. The bits that are 1 in addressMask indicate the address bits of the XSDR instruction; those that are 1 in dataMask indicate the data bits of the XSDR instruction. "Length" comes from the value of the last XSDRSize instruction.

Example:

```
XSETSDRMASKS  00800000 000003fc
```

XSDRINC

```
XSDRINC startAddress<"length" bits>
numTimes<1 byte> data[1]<"length2" bits>
...data[numTimes]<"length2" bits>
```

Do successive XSDR instructions. Length is specified by the last XSDRSIZE instruction. Length2 is specified as the number of 1 bits in the dataMask section of the last XSETSDRMASKS instruction.

The startAddress is the first XSDR to be read in. For numTimes iterations, increment the address portion (indicated by the addressMask section of the last XSETSDRMASKS instruction) by 1, and load in the next data portion into the dataMask section.

Note that an XSDRINC <start> 255 data0 data1 ... data255 actually does 256 SDR instruction since the start address also represents an SDR instruction

Example:

```
XSDRINC  004003fe 05 ff ff ff ff ff
```

XCOMPLETE

```
XCOMPLETE
```

End of XSVF file reached.

Example:

```
XCOMPLETE
```

svf2xsvf File Conversion Utility

This executable reads in an SVF file (generated by EZTag) and generates an XSVF file.

Usage:

```
svf2xsvf [-nc] [-r number] <file1>
<file2>
```

file1: SVF input file name.
file2: XSVF output file name.

Options:

-nc — No compression. Don't use the XSETSDRMASKS and XSDRINC instructions.

-r *number* — Set the XREPEAT value to number

xsvf2ascii File Conversion Utility

This executable reads in an XSVF file (generated by svf2xsvf) and outputs the XSVF commands contained in the file. It is useful for debugging.

Usage:

```
xsvf2ascii <file1> <file2>
```

file1: XSVF input file name.
file2: ascii output file name.

Appendix C

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.

For help in debugging the code, a compiler switch called DEBUG_MODE is provided. This switch allows the designer to simulate the TAP outputs in a PC environment. If DEBUG_MODE is defined, the software reads from an XSVF file (which must be named prom.bit) and prints the calculated value of the microcontroller's I/O ports (TDI and TMS) on each rising edge of TCK. Because the TDO value cannot be read during DEBUG_MODE, the software assumes that the TDO value is correct. This function provides a simulation of the TAP signals that can be used to verify the actual operation.

```
r
/*******************************************************/
/* file: lenval.c                                      */
/* abstract:  This file contains routines for using    */
/*            the lenVal data structure.               */
/*******************************************************/
#include "lenval.h"
#include "ports.h" /* for DEBUG_MODE define */

/* return the value represented by this lenval */
long value(lenVal *x)
{
    int i;
    long result=0;          /* result to hold the accumulated result */
    for (i=0;i<x->len;i++)
    {
       result=result<<8;   /* shift the accumulated result */
       result+=x->val[i];  /* get the last byte first */
    }
    return result;
}

/* set x to value; assumes value<512 */
void initLenVal(lenVal *x, long value)
{
    x->len=1;
    x->val[0]=(unsigned char) value;
}
```

```
/* return TRUE iff actual=expected (after masking out some bits using mask */
short EqualLenVal(lenVal *expected, lenVal *actual, lenVal *mask)
{
    int i;
#ifdef DEBUG_MODE
    /* since we can't read TDO, just assume TDO matches whatever is expected */
    return 1;
#endif
    for (i=0;i<expected->len;i++)
    {
       unsigned char byteVal1=expected->val[i];   /* i'th byte of expected */
       unsigned char byteVal2=actual->val[i];     /* i'th byte of actual   */
       byteVal1 &= mask->val[i];                  /* mask out expected     */
       byteVal2 &=mask->val[i];                   /* mask out actual       */
       if (byteVal1!=byteVal2)
           return 0;                              /* values are not equal  */
    }
    return 1;                                      /* values are equal      */
}



/* return the (byte, bit) of lv (reading from left to right) */
short RetBit(lenVal *lv, int byte, int bit)
{
    int i;
    unsigned char ch=lv->val[byte];   /* get the correct byte of data */
    unsigned char mask=128; /* 10000000 */
    for (i=0;i<bit;i++)
       mask=mask>>1;       /* mask the correct bit of the byte       */
    return ((mask & ch) !=0);  /* return 1 if the bit is 1, 0 otherwise */
}

/* set the (byte, bit) of lv equal to val (e.g. SetBit("00000000",byte, 1)
   equals "01000000" */
void SetBit(lenVal *lv, int byte, int bit, short val)
{
    int i;
    unsigned char *ch=&(lv->val[byte]);
    unsigned char OrMask=1, AndMask=255;
    for (i=0;i<7-bit;i++)
       OrMask=OrMask<<1;
    AndMask-=OrMask;
    *ch = *ch & AndMask;  /* 0 out the bit */
    if (val)
       *ch = *ch | OrMask;   /* fill in the bit with the correct value */
}

/* add val1 to val2 and store in resVal;      */
/* assumes val1 and val2  are of equal length */
void addVal(lenVal *resVal, lenVal *val1, lenVal *val2)
{
    unsigned char carry=0;
    short i;
    resVal->len=val1->len;  /* set up length of result */
    /* start at least significant bit and add bytes    */
    for (i=val1->len-1;i>=0;i--)
    {
       unsigned char v1=val1->val[i];  /* i'th byte of val1 */
       unsigned char v2=val2->val[i];  /* i'th byte of val2 */
       /* add the two bytes plus carry from previous addition */
       unsigned char res=v1+v2+carry;
       /* set up carry for next byte */
       if (v1+v2+carry>255)
           carry=1;                    /* carry into next byte */
       else
           carry=0;
       resVal->val[i]=res;             /* set the i'th byte of the result */
    }
}

/* read from XSVF numBytes bytes of data into x */
void readVal(lenVal *x, short numBytes)
{
    int i;
    for (i=0;i<numBytes;i++)
        readByte(&(x->val[i]));  /* read a byte of data into the lenVal */
    x->len=numBytes;             /* set the length of the lenVal        */
}
```

```
/*******************************************************/
/* file: lenval.h                                      */
/* abstract:  This file contains a description of the  */
/*            data structure "lenval".                 */
/*******************************************************/

#ifndef lenval_dot_h
#define lenval_dot_h

/* the lenVal structure is a byte oriented type used to store an */
/* arbitrary length binary value. As an example, the hex value   */
/* 0x0e3d is represented as a lenVal with len=2 (since 2 bytes   */
/* and val[0]=0e and val[1]=3d.  val[2-MAX_LEN] are undefined    */

/* maximum length (in bytes) of value to read in        */
/* this needs to be at least 4, and longer than the     */
/* length of the longest SDR instruction.  If there is, */
/* only 1 device in the chain, MAX_LEN must be at least */
/* ceil(27/8) == 4.  For 6 devices in a chain, MAX_LEN  */
/* must be 5, for 14 devices MAX_LEN must be 6, for 20  */
/* devices MAX_LEN must be 7, etc..                     */
/* You can safely set MAX_LEN to a smaller number if you*/
/* know how many devices will be in your chain.         */
#define MAX_LEN 4


typedef struct var_len_byte
{
    short len;   /* number of chars in this value */
    unsigned char val[MAX_LEN+1];  /* bytes of data */
} lenVal;


/* return the long representation of a lenVal */
extern long value(lenVal *x);

/* set lenVal equal to value */
extern void initLenVal(lenVal *x, long value);

/* check if expected equals actual (taking the mask into account) */
extern short EqualLenVal(lenVal *expected, lenVal *actual, lenVal *mask);

/* add val1+val2 and put the result in resVal */
extern void addVal(lenVal *resVal, lenVal *val1, lenVal *val2);

/* return the (byte, bit) of lv (reading from left to right) */
extern short RetBit(lenVal *lv, int byte, int bit);

/* set the (byte, bit) of lv equal to val (e.g. SetBit("00000000",byte, 1)
   equals "01000000" */
extern void SetBit(lenVal *lv, int byte, int bit, short val);

/* read from XSVF numBytes bytes of data into x */
extern void  readVal(lenVal *x, short numBytes);

#endif
```

```
/************************************************************/
/* file: micro.c                                            */
/* abstract:  This file contains the main function          */
/*            call for reading in a file from a prom        */
/*            and pumping the JTAG ports.                   */
/*                                                          */
/* Notes: There is a compiler switch called DEBUG_MODE.     */
/*        If DEBUG_MODE is defined, the compiler will read  */
/*        the xsvf file from a file called "prom.bit".     */
/*        It will also enable debugging of the code         */
/*        by printing the TDI and TMS values on the         */
/*        rising edge of TCLK.                              */
/************************************************************/

#include "lenval.h"
#include "ports.h"


#define CLOCK_RATE 150    /* set to be the clock rate of the system in kHz */

/* encodings of xsvf instructions */

#define XCOMPLETE        0
#define XTDOMASK         1
#define XSIR             2
#define XSDR             3
#define XRUNTEST         4
#define XREPEAT          7
#define XSDRSIZE         8
#define XSDRTDO          9
#define XSETSDRMASKS     10
#define XSDRINC          11

/* return number of bytes necessary for "num" bits */
#define BYTES(num) \
        (((num%8)==0) ? (num/8) : (num/8+1))

extern void doSDRMasking(lenVal *dataVal, lenVal *nextData,
             lenVal *addressMask, lenVal *dataMask);
extern short loadSDR(int numBits, lenVal *dataVal, lenVal *TDOExpected,
              lenVal *TDOMask);
extern void clockOutLenVal(lenVal *lv,long numBits,lenVal *tdoStore);
extern void gotoIdle();

lenVal TDOMask;    /* last TDOMask received */
lenVal maxRepeat;  /* max times tdo can fail before ISP considered failed */
lenVal runTestTimes; /* value of last XRUNTEST instruction */


#ifdef DEBUG_MODE
#include <stdio.h>
FILE *in;          /* for debugging */
#endif
```

```
/* clock out the bit onto a particular port */
void clockOutBit(short p, short val)
{
    setPort(p,val);  /* change the value of TMS or TDI */
    pulseClock();    /* set TCK to Low->High->Low      */
}

/* clock out numBits from a lenVal;  the least significant bits are */
/* output on the TDI line first; exit into the exit(DR/IR) state.   */
/* if tdoStore!=0, store the TDO bits clocked out into tdoStore.    */
void clockOutLenVal(lenVal *lv,long numBits,lenVal *tdoStore)
{
    int i;
    short j,k;

    /* if tdoStore is not null set it up to store the tdoValue */
    if (tdoStore)
       tdoStore->len=lv->len;

    for (i=0;i<lv->len;i++)
    {
       /* nextByte contains the next byte of lenVal to be shifted out */
       /* into the TDI port                                           */
       unsigned char nextByte=lv->val[lv->len-i-1];
       unsigned char nextReadByte=0;
       unsigned char tdoBit;
       /* on the last bit, set TMS to 1 so that we go to the EXIT DR */
       /* or to the EXIT IR state */
       for (j=0;j<8;j++)
       {
           /* send in 1 byte at a time */
           /* on last bit, exit SHIFT SDR */
           if (numBits==1)
             setPort(TMS,1);

           if (numBits>0)
           {
             tdoBit=readTDOBit();  /* read the TDO port into tdoBit */
             clockOutBit(TDI,nextByte & 0x1);  /* set TDI to last bit */
             nextByte=nextByte>>1;
             numBits--;
             /* first tdoBit of the byte goes to 0x00000001   */
             /* second tdoBit goes to 0x00000010, etc.        */
             /* Shift the TDO bit to the right location below */
             for (k=0;k<j;k++)
                tdoBit=tdoBit<<1;

             /* store the TDO value in the nextReadByte */
             nextReadByte|=tdoBit;
           }
       }
       /* if storing the TDO value, store it in the correct place */
       if (tdoStore)
           tdoStore->val[tdoStore->len-i-1]=nextReadByte;
    }
}
```

```
/* parse the xsvf file and pump the bits */
int main()
{
    lenVal inst; /* instruction */
    lenVal bitLengths; /* hold the length of the arguments to read in */
    lenVal dataVal,TDOExpected;
    lenVal SDRSize,addressMask,dataMask;
    lenVal sdrInstructs;
    long i;

#ifdef DEBUG_MODE
    /* read from the file "prom.bit" instead of a real prom */
    in=fopen("prom.bit","rb");
#endif
    gotoIdle();
    while (1)
    {
       readVal(&inst,1);  /* read 1 byte for the instruction */
       switch (value(&inst))
       {
       case XTDOMASK:
           /* read in new TDOMask */
           readVal(&TDOMask,BYTES(value(&SDRSize)));
           break;
       case XREPEAT:
           /* read in the new XREPEAT value */
           readVal(&maxRepeat,1);
           break;
       case XRUNTEST:
           /* read in the new RUNTEST value */
           readVal(&runTestTimes,4);
           break;
       case XSIR:
           /* load a value into the instruction register */
           clockOutBit(TMS,1);  /* Select-DR-Scan state  */
           clockOutBit(TMS,1);  /* Select-IR-Scan state  */
           clockOutBit(TMS,0);  /* Capture-IR state      */
           clockOutBit(TMS,0);  /* Shift-IR state        */
           readVal(&bitLengths,1); /* get number of bits to shift in */
           /* store instruction to shift in */
           readVal(&dataVal,BYTES(value(&bitLengths)));
           /* send the instruction through the TDI port and end up   */
           /* dumped in the Exit-IR state                            */
           clockOutLenVal(&dataVal,value(&bitLengths),0);
           clockOutBit(TMS,1);  /* Update-IR state       */
           clockOutBit(TMS,0);  /* Run-Test/Idle state   */
           break;
       case XSDRTDO:
           /* get the data value to be shifted in */
           readVal(&dataVal,BYTES(value(&SDRSize)));
           /* store the TDOExpected value    */
           readVal(&TDOExpected,BYTES(value(&SDRSize)));
           /* shift in the data value and verify the TDO value against */
           /* the expected value                                       */
           if (!loadSDR(value(&SDRSize), &dataVal, &TDOExpected, &TDOMask))
           {
```

```
         /* The ISP operation TDOs failed to match expected */
         return 0;
           }
           break;
       case XSDR:
           readVal(&dataVal,BYTES(value(&SDRSize)));
           /* use TDOExpected from last XSDRTDO instruction */
           if (!loadSDR(value(&SDRSize), &dataVal, &TDOExpected, &TDOMask))
             return 0;  /* TDOs failed to match expected */
           break;
       case XSDRINC:
           readVal(&dataVal,BYTES(value(&SDRSize)));
           if (!loadSDR(value(&SDRSize), &dataVal, &TDOExpected, &TDOMask))
             return 0;  /* TDOs failed to match expected */
           readVal(&sdrInstructs,1);
           for (i=0;i<value(&sdrInstructs);i++)
           {
             lenVal nextData;
             int dataLength=8; /* fix to be number of 1's in dataMask */
             readVal(&nextData,BYTES(dataLength));
             doSDRMasking(&dataVal, &nextData, &addressMask, &dataMask);
             if (!loadSDR(value(&SDRSize), &dataVal,
                        &TDOExpected, &TDOMask))
                return 0;  /* TDOs failed to match expected */
           }
           break;
       case XSETSDRMASKS:
           /* read the addressMask */
           readVal(&addressMask,BYTES(value(&SDRSize)));
           /* read the dataMask    */
           readVal(&dataMask,BYTES(value(&SDRSize)));
           break;
       case XCOMPLETE:
           /* return from subroutine */
           return 1;
           break;
        case XSDRSIZE:
           /* set the SDRSize value */
           readVal(&SDRSize,4);
           break;

       }
    }
}



/* determine the next data value from the XSDRINC instruction and store     */
/* it in dataVal.                                                            */
/* Example:  dataVal=0x01ff, nextData=0xab, addressMask=0x0100,              */
/*           dataMask=0x00ff, should set dataVal to 0x02ab                   */
void doSDRMasking(lenVal *dataVal, lenVal *nextData, lenVal *addressMask,
           lenVal *dataMask)
{
    int i,j,count=0;
    /* add the address Mask to dataVal and return as a new dataVal */
    addVal(dataVal, dataVal, addressMask);
    for (i=0;i<dataMask->len;i++)
    {
       /* look through each bit of the dataMask. If the bit is    */
       /* 1, then it is data and we must replace the corresponding*/
       /* bit of dataVal with the appropriate bit of nextData     */
       for (j=0;j<8;j++)
           if (RetBit(dataMask,i,j))  /* this bit is data */
           {
             /* replace the bit of dataVal with a bit from nextData */
             SetBit(dataVal,i,j,RetBit(nextData,count/8,count%8));
             count++;  /* count how many bits have been replaced */
           }
    }
}

/* goto the idle state by setting TMS to 1 for 5 clocks, followed by TMS */
/* equal to 0 */
void gotoIdle()
{
    int i;
    setPort(TMS,1);
    for (i=0;i<5;i++)
       pulseClock();
    setPort(TMS,0);
    pulseClock();
}


/* return 0 iff the TDO doesn't match what is expected */
short loadSDR(int numBits, lenVal *dataVal, lenVal *TDOExpected,
              lenVal *TDOMask)
{
    int failTimes=0;
    while (1)
    {
       lenVal actualTDO;
       int repeat;  /* to do RUNTESTS */
       clockOutBit(TMS,1);  /* Select-DR-Scan state  */
       clockOutBit(TMS,0);  /* Capture-DR state      */
       clockOutBit(TMS,0);  /* Shift-DR state        */
       /* output dataVal onto the TDI ports; store the TDO value returned */
       clockOutLenVal(dataVal,numBits,&actualTDO);
       /* compare the TDO value against the expected TDO value */
       if (EqualLenVal(TDOExpected,&actualTDO,TDOMask))
       {
           /* TDO matched what was expected */
           clockOutBit(TMS,1);  /* Update-DR state    */
           clockOutBit(TMS,0);  /* Run-Test/Idle state*/

           /* wait in Run-Test/Idle state */
           waitTime(value(&runTestTimes));
           break;
       }
       else
       {
            /* TDO did not match the value expected */
            failTimes++;      /* update failure count */
            if (failTimes>value(&maxRepeat))
               return   0;   /* ISP failed */
            clockOutBit(TMS,0); /* Pause-DR state      */
            clockOutBit(TMS,1); /* Exit2-DR state      */
            clockOutBit(TMS,0); /* Shift-DR state      */
            clockOutBit(TMS,1); /* Exit1-DR state      */
            clockOutBit(TMS,1); /* Update-DR state     */
            clockOutBit(TMS,0); /* Run-Test/Idle state */
            /* wait in Run-Test/Idle state */
            waitTime(value(&runTestTimes));
        }
    }
    return 1;
}
```

```
/*******************************************************/
/* file: ports.c                                       */
/* abstract:  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 prom                            */
/*                                                     */
/* Notes: See the notes for micro.c for explanation of */
/*        the compiler switch  "DEBUG_MODE".           */
/*******************************************************/
#include "ports.h"

#ifdef DEBUG_MODE
#include "stdio.h"
extern FILE *in;
#endif

#ifdef DEBUG_MODE
/* if in debugging mode, use variables instead of setting the ports */
short pTCK,pTMS,pTDI;
#endif


/* if in debugging mode, then just set the variables */
void setPort(short p,short val)
{
#ifdef DEBUG_MODE
    if (p==TCK)
        pTCK=val;
    if (p==TMS)
        pTMS=val;
    if (p==TDI)
        pTDI=val;
#endif
}

#ifdef DEBUG_MODE
void printPorts()
{
    printf("%d   %d\n",pTMS,pTDI);
}
#endif

/* toggle tck LHL */
void pulseClock()
{
    setPort(TCK,0);  /* set the TCK port to low  */
    setPort(TCK,1);  /* set the TCK port to high */
#ifdef DEBUG_MODE
    /* if in debugging mode, print the ports on the rising clock edge */
    printPorts();
#endif;
    setPort(TCK,0);  /* set the TCK port to low  */

}
```

```
/* read in a byte of data from the prom */
void readByte(unsigned char *data)
{
#ifdef DEBUG_MODE
    /* pretend reading using a file */
    fscanf(in,"%c",data);
#endif
}

/* read the TDO bit from port */
unsigned char readTDOBit()
{
#ifdef DEBUG_MODE
    return 1;  /* garbage value for now; replace with real port read. */
#endif
}


/* Wait at least the specified number of microsec.                           */
/* Use a timer if possible; otherwise estimate the number of instructions    */
/* necessary to be run based on the microcontroller speed.  For this example */
/* we pulse the TCK port a number of times based on the processor speed.     */
void waitTime(long microsec)
{
    int repeat;
#define CLOCK_RATE 150 /* set to be the clock rate of the system in kHz */
    long clockRunTests=microsec*CLOCK_RATE/1000;
    for (repeat=0;repeat<clockRunTests;repeat++)
                pulseClock();
}
```

```
/*******************************************************/
/* file: ports.h                                       */
/* abstract:  This file contains extern declarations   */
/*            for providing stimulus to the JTAG ports.*/
/*******************************************************/

#ifndef ports_dot_h
#define ports_dot_h

#if 0
#define DEBUG_MODE  /* this line can be enabled in order to read in the xsvf from */
             /* a file called "prom.bits" and output the                   */
             /* TMS and TDI values on the rising edge of                   */
             /* the clock                                                  */
#endif

/* these constants are used to send the appropriate ports to setPort */
/* they should be enumerated types, but some of the microcontroller  */
/* compilers don't like enumerated types */
#define TCK 0
#define TMS 1
#define TDI 2

/* set the port "p" (TCK, TMS, or TDI) to val (0 or 1) */
extern void setPort(short p, short val);

/* read the TDO bit and store it in val */
extern unsigned char readTDOBit();

/* make clock go down->up->down*/
extern void pulseClock();

/* read the next byte of data from the xsvf file */
extern void readByte(unsigned char *data);

#endif
```

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 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 compatible instructions and data, and outputs the programming information to the XC9500 device through a 4-wire test access port.

After an XC9500 design has been converted to XSVF format, the XSVF information is converted to Intel hex format which is downloaded 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));
}
```