Reconfiguring Excalibur Devices Under Processor Control

February 2003, ver. 1.0 Application Note 298

Introduction

The Excalibur™ devices have a powerful embedded processor, which is integrated with the APEX® FPGA. The embedded processor is active, independent of the FPGA configuration, which allows software control of the FPGA contents.

Embedded Systems are rapidly increasing in size and complexity. Design specifications are calling for an increase in feature set and flexiblity. Time-to-market presures and cost constraints are pushing embedded systems to new levels of system integration and innovatation.

The Excalibur™ device family offers designers an embedded platform that can address the demands of today's embedded systems. The unique integration of an embedded processor stripe on a FPGA offers a new level of system integration to address the complexity, feature set, and flexiblity needs of embedded systems designs.

One particular feature of the Excalibur device family that helps with todays demands is the ablity to configure the FPGA under processor control. The embedded processor is active independent of the FPGA configuration, which allows for software control of the FPGA contents.

This application note describes the multiple image reference designs, which provide the fundamental basis for creating applications with Excalibur devices that use multiple image boot options. It discusses the options for booting and running multiple hardware and software images and discusses applications where Excalibur devices add value.

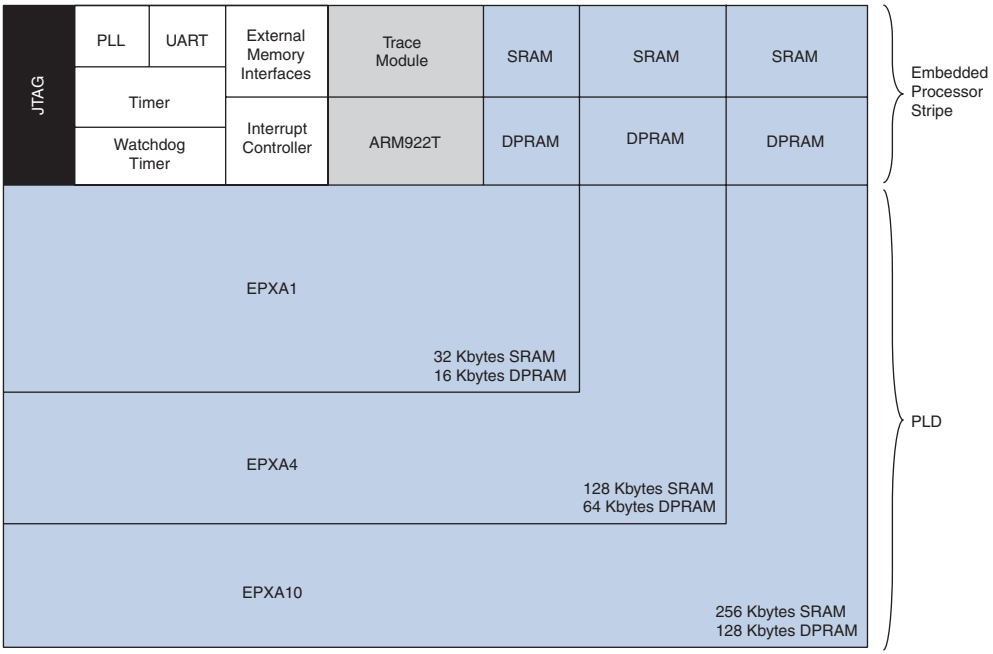
Multiple Image Reference Designs

The multiple image reference designs demonstrate how the embedded processor can select from multiple hardware or software images with which to configure the device.. One design shows you how to load new FPGA hardware into the system, based on an event that occurs in the system. The other design shows you how to select one of multiple combined hardware and software images at boot time.

Overview of Excalibur Devices

The family of Excalibur devices is a complete embedded processor FPGA solution that addresses the needs of today's high-end design requirements. Combining flexibility, integration, and performance, the Excalibur devices consist of an embedded processor within the embedded stripe and the APEX FPGA. Figure 1 shows a block diagram of the Excalibur devices.

Figure 1. Excalibur Devices Block Diagram



The embedded stripe consists of an ARM922T, running at 200 MHz, integrated SRAM, dual-port SRAM (DPRAM), an SDRAM controller, and other embedded peripherals. The embedded stripe uses two high performance advanced microcontroller bus architecture (AMBA™) high-performance buses (AHBs): AHB1 and AHB2, as its communication medium for the peripherals on the embedded stripe.

In addition to the embedded stripe, Excalibur devices have the APEX FPGA. The embedded stripe-FPGA interface consists of two AHB bridges, which allow AHB communication by peripherals in the embedded stripe and AHB peripherals implemented by designers in the FPGA. In addition, the DPRAM interface allows the logic in the FPGA to interface with peripherals in the embedded stripe.

The embedded stripe is a complete processor subsystem that addresses the system requirements of many embedded system designs. The addition of the APEX FPGA adds all the flexibility of programmable logic and greatly extends the feature set that can be implemented with these devices.

The embedded stripe macro cell operates independently of the FPGA. You have software control over the FPGA contents—the processor is functional regardless of the contents of the FPGA. The on-chip intelligence of the embedded processor allows the swapping of hardware and software images, which has many applications for improving design cost, flexibility, and performance.

For more information on the Excalibur devices, refer to the *Excalibur Devices Hardware Reference Manual.*

Reference Design

The mutiple images reference designs show the following two variations of using the embedded stripe to control which function is running on an Excalibur device:

- Selecting one of multiple FPGA hardware images
- Selecting one of multiple combined hardware and software images.

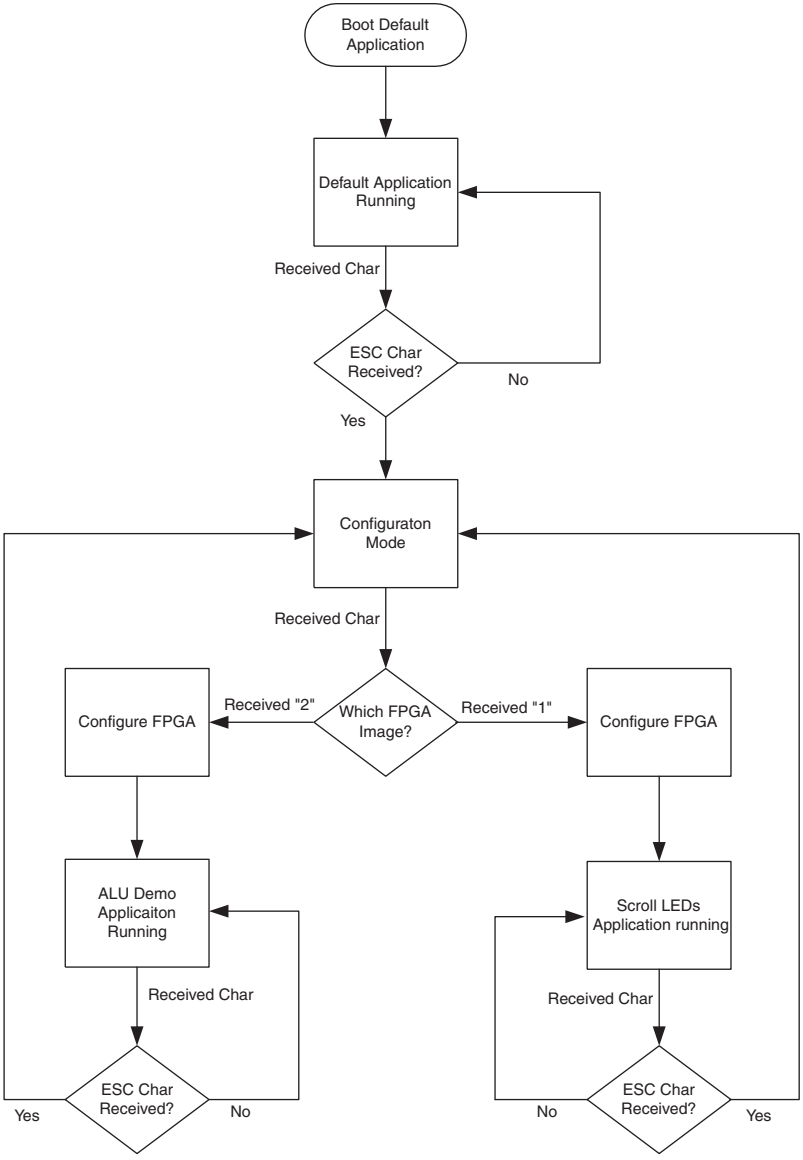
Showing these two basic ways of selecting either different hardware or selecting different hardware and software images are the fundamental basis for achieving some of the flexibility benefits Excalibur devices have to offer.

For more details, see "Example Applications" on page 7.

Multiple FPGA Hardware Images Reference Design

The multiple FPGA hardware images reference design shows an example processor application, which has a configuration mode that is triggered by a system event. When in configuration mode, the application determines which FPGA image is loaded and then exercises it. Figure 2 shows a flow diagram of the multiple FPGA hardware images reference design.

Figure 2. Multiple FPGA Hardware Images Reference Design Flow Diagram



The design boots and loads a default application. The default application continues to run until an event occurs, which signifies that the design should enter configuration mode. In this case, the event is the pressing of the ESC key on the keyboard. The default application enters the configuration mode and determines which FPGA image to load. Once the new FPGA image is loaded, the application exercises the new FPGA image. Pressing ESC, at any time, returns the application to the configuration mode.

This technique allows you to load different hardware images at different times. Designs implemented in Excalibur devices can be partitioned into distinct functional blocks. The on-chip intelligence of the embedded processor allows you to load the functional blocks as needed, while the software application is still running. Loading different hardware images at different times allows you to cut the cost of a design and increases the flexibility of a platform for specification changes.

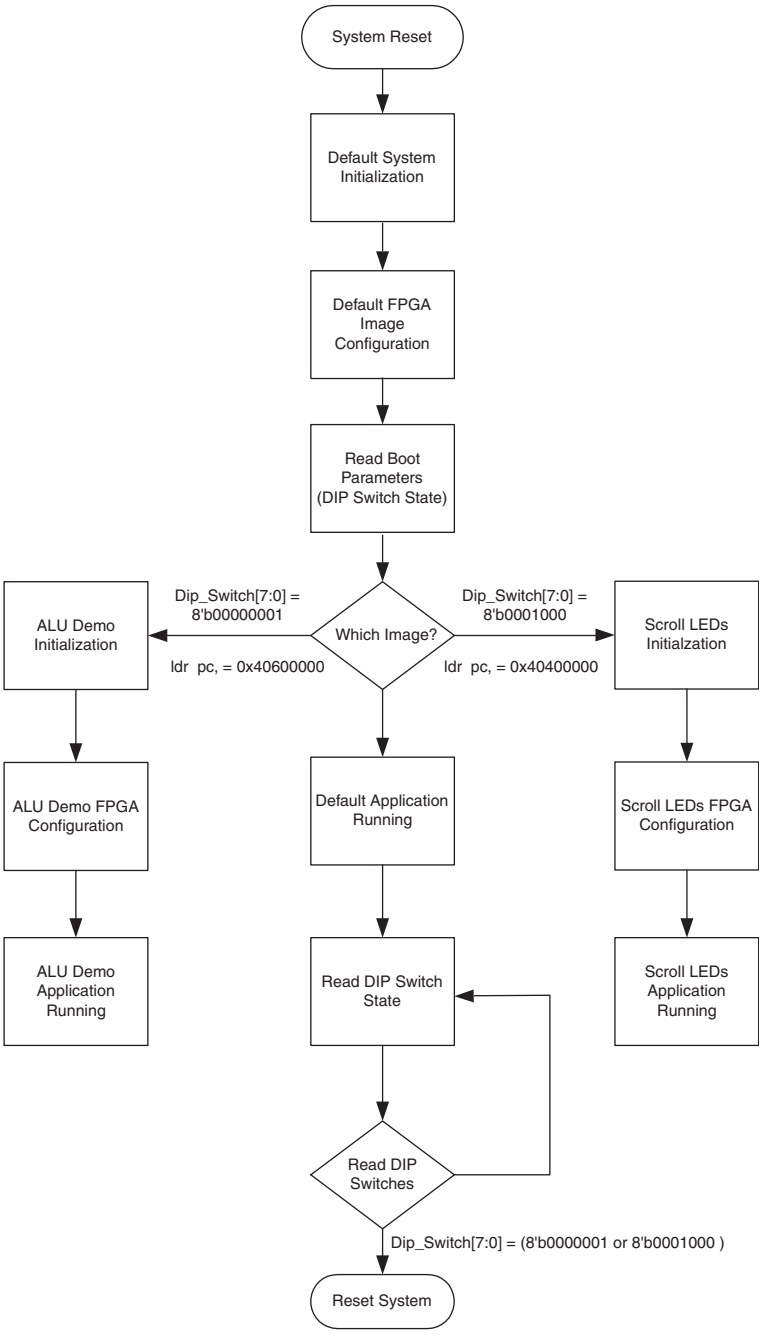
For more information , see "Example Applications" on page 7.

The software that drives the FPGA images must be loaded at boot time. All of the application code for all FPGA images are present in the system, which makes a big application code footprint. In addition, drastic changes to the hardware image in the FPGA can cause the system to malfunction, unless certain constrains are placed on the new image. One solution is to load both the FPGA hardware and the embedded software that is specific to a function, which is described in the following section.

Multiple FPGA Hardware and Software Images

The multiple FPGA hardware and software images reference design shows an example of loading multiple FPGA hardware and software application images. Boot parameters are read during system bring-up, to determine where in the memory the processor begins the application execution. Figure 3 shows a flow diagram of the multiple FPGA hardware and software images reference design.

Figure 3. Multiple FPGA Hardware and Software Images Reference Design Flow Diagram



Initial system bring-up occurs, then the processors read the boot parameters. The state of the dual in-line package (DIP) switches is read through the general purpose input and output (GPIO) port, which determines the location where the processor acesses, to begin the execution of the application. The application that is branched to initializes the system to its needs and also configures the FPGA with the new hardware image. The initialization includes setting up the memory map for the new application, after which the new application is executed.

Reading boot parameters is how typical sophisticated boot loaders, e.g., ARMboot or Redboot, determine where to begin the execution of an application. Images can be run from any non-volatile memory resource that the system has access to. Images can reside in memory resources in the system or reside in remote locations to the system.

For further information, see "Example Applications" on page 7.

This specific implementation requires a reboot to change the image that is running in the system. Many applications cannot tolerate a processor reboot to change the functionality of the application. For some applications either the time to reboot, or losing the processor state while the system is running, is not feasible. An operating system that can load drivers at run time with the **sbi_config** utility allows you to dynamically load FPGA hardware and software drivers.

The sbi_config Utility

The Altera **sbi_config** utility addresses the inadequacies of the previous two designs. The **sbi_config** utility can append configuration information to the hardware image. The configuration information can be interrogated at the time when drivers are loaded. The configuration information contains information on the location of hardware in the address space and specific parameters for configurable peripherals that are in the hardware image. The **sbi_config** utility provides the ultimate flexibility and reuse of code; you can write configurable drivers, which reduce the limitations on differences between FPGA images.

For more information on the **sbi_config** utility, see the Quartus II Help.

Example Applications

This section describes general applications where the previous designs apply to Excalibur devices.

Reconfigurable Computing

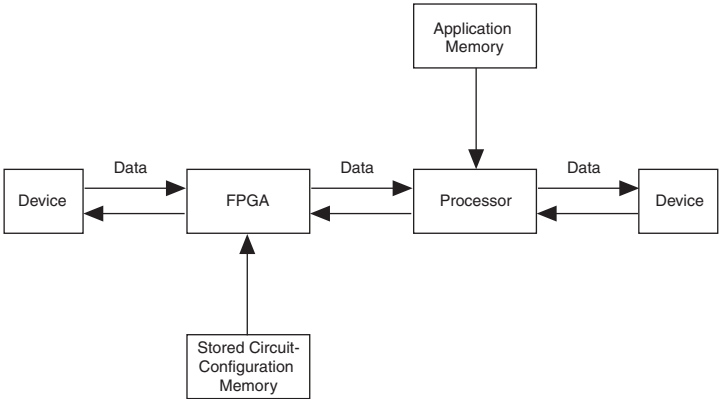
A significant amount of research has gone into reconfigurable computing. One of the goals of reconfigurable computing is to make more efficient use of system resources, by having hardware present only when it is in use.

One example of reconfigurable computing is where a system breaks down a task into a series of smaller tasks, which are computed in a serial fashion. Intermediate results of computations are stored memory resources and used as inputs for the next FPGA image to perform the next computation. The goal of this approach is to reduce the overall system size by loading the FPGA image that performs a specific portion of the computation.

A second example of reconfigurable computing copes with varying standards and dynamically changing interfaces. Where a system handles different interfaces and protocol standards, the ability to shift hardware when it is needed greatly reduces the overall size of the system hardware, e.g., a signal processing system. A system may consist of a number of different filtering algorithms with different word lengths that are done in hardware. A set of FPGA implementations of the different algorithms are stored in a non-volatile memory and shifted in the system as the dynamics of the system changes.

Figure 4 shows the basic stucture of reconfigurable computing systems. Excalibur devices reduce the number of devices required, as some of the functions, such as the processor, memory, and FPGA, are integrated on the device.

Figure 4. Reconfigurable Computing System Structure

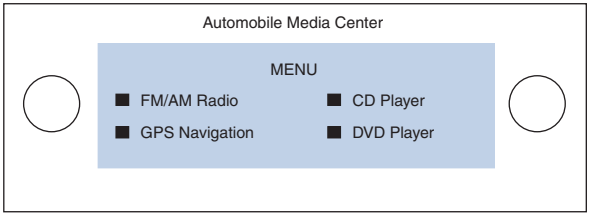


Standard Hardware Platform Deployment

A significant amount of time and resources are spent in the deployment of any system. Development and verification cycles increase as the specifications and the requested feature set increase. More features requested by the specification require additional time and resources. One way to increase time-to-market is to create a single standard platform that has the flexibility to be configured to perform different specific functions. This is an attractive option for products that interface to a common set of peripherals.

For example, an automobile media center (see Figure 5)—one standard hardware platform performs all possible functions of the media center.

Figure 5. Media Center



A system that functions as a DVD player, GPS system, CD player and AM/FM radio can be created. Because the basic interface for these different products can be reused, the system can be easily upgraded to add support for new features. For example, a GPS upgrade can be achieved by downloading a new flash image to the media system with no modifications to the hardware.

Remote Reconfiguration

Performing remote upgrades to an application has many advantages. It applies in the areas of bug fixes and extends to the area of standard platforms.

A system can be distributed out to the field and on system bring-up establish an FTP session with a remote host. The system would have an identifier and pull its new image from the server based on that identifier. Designers can ship systems and make modifications to the internal workings before they arrive.

For an example of remote reconfiguration, see *AN278: Configuring the EPXA1 Development Board with Linux* and *AN213: Excalibur Remote Reconfiguration Design*.

Install the Designs

This section covers the designs' requirements and the directory structure.

Requirements

The designs require the following software:

- Altera Quartus™ II software version 2.2
- ARM Developer Suite (ADS) software version 1.1
- Make utility
- EPXA1 development board

You should have a fundamental knowledge in the following areas:

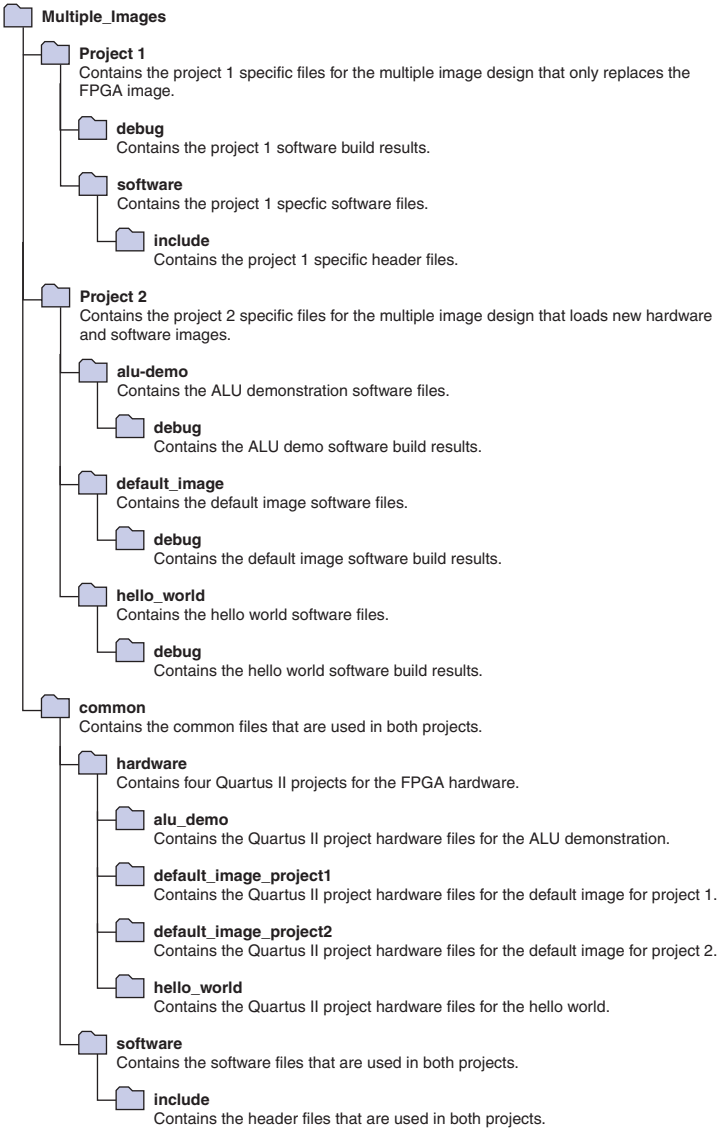
- C and assembly language
- Excalibur design tool flow
- The Quartus II software
- The ADS
- Verilog HDL

Directory Structure

To install the designs, first unzip **Multiple_Images.zip** to your desired location. Right-click on the zip file and choose **Extract to Folder to:.**

Figure 6 shows the directory structure.

Figure 6. Directory Structure



Tables 1 to 7 detail the contents of the directories.

Tables 1 and 2 detail the content of the Project 1 directory. Project 1 has one set of software files that are built to produce one software image. It uses three hardware images, the default image, alu_demo image and hello_world image, which are located in the **\common** directory.

Table 1. Project1 Directory Files

Filename	Description
Build_project.bat	Batch file that builds software.
Clean_build.bat	Batch file that deletes previous compilation results and rebuilds the software.
Makefile.mak	Make file.
Prog_hw.bat	Batch file that programs the EPXA1 development board.

Table 2. Project1\software Directory Files

Filename	Description
Alu_demo.c	Alu_demo application code, which takes the user input and writes from or reads to the arithmetic logic unit (ALU) in the FPGA.
Armc_startup.s	Contains IRQs, sets up stacks, and branches to main.
Config_logic.c	Takes an address of slave binary image (SBI) data and configures the FPGA.
Get_image_num.c	The configuration mode where the new images, which need to be programmed, are selected.
Main().c	Simple default application that looks for an ESC character from the UART.
Sbi_data.s	Pulls in SBI data using the INCBIN directive.
Scroll_leds.c	Application code that drives walking 1s on to the LEDs and prints a messages out of the UART.
Scatter.link	Scatter link file that places: Read_Only @0x0 Sbi_data.o @end of Read_Only Read_Write @ 0x10000000.

Note to Table 2:

(1) The **\project1\software\include** directory contains header files specific to project 1. The **project1\debug** directory is where software build results are stored.

Tables 3 to 7 detail the content of the Project 2 directory. Project 2 has three sets of software files that are built to produce three software image. These software images all link to their own specific hardware images, which are in the **\common\hardware** directory.

Table 3. Project2 Directory Files

Filename	Description
Build_project.bat	Batch file that builds software for the default image, alu_demo, and hello_world.
Clean_build.bat	Batch file that deletes previous compilation results and rebuilds all three images.
Prog_hw.bat	Batch that programs all images to the EPXA1 development board.

Table 4. Project2\alu_demo Directory Files

Filename	Description
Bootloader.s	Initializes the EPXA device and links in SBI data for ALU hardware.
Main.c	Alu_demo application code, which takes the user input and writes to and reads from the ALU in the FPGA.
Scatter.link	Scatter link file that places: Read_Only @0x0 Sbi_data.o @end of Read_Only +40600000 Read_Write @ 0x20000.
Makefile.mak	Make file.

Table 5. Project2\default_image Directory Files

Filename	Description
Bootloader.s	Initializes the EPXA1 device and links in SBI data for default hardware.
Main.c	Default application code that waits for a state change on DIP switches.
Scatter.link	Scatter link file that places: Read_Only @0x0, Sbi_data.o @end of Read_Only +40000000, Read_Write @ 0x10000000.
Makefile.mak	Make file.

Table 6. Project2\hello_world Directory Files

Filename	Description
Bootloader.s	Initializes the EPXA1 device and links in the SBI data for the hello_world hardware.
Hello_world.c	Application code that drives walking 1s on to the LEDs and prints a messages out of the UART.
Scatter.link	Scatter link file that places: Read_Only @0x0, Sbi_data.o @end of Read_Only +40400000, Read_Write @ 0x20000.
Makefile.mak	Make file.

Note to Table 6:
(1) Software build results are stored in the **project2*\debug** directories.

Table 7 describes the contains of the **\common\software** directory.

Table 7. Project2\common \software Directory Files

Filename	Description
Init.s	Used by all project 2 images. Initiazies stacks, turns on instruction caches, and branches to main.
Irq.c	Contains IRQ handler for all software images.
Retarget.c	Used by all applications to set up heap base and stack base in memory.
Uartcomm.c	A driver that is used by all applications for communication with the UART.

Note to Table 7:
(1) The **\common\software\include** directory contains header files that are used for software images.

Run the Designs

The multiple image reference designs ship with **.hex** files that are ready for download to an EPXA1 development board. To run the **.hex** file, you must configure the EPXA1 development board and program the flash memory.

Configure the EPXA1 Development Board

1. Connect the ByteBlaster™ download Cable to the 10 pin header on the development board
2. Connect the serial cable to the RS-232 port 1 (located adjacent to the 10 pin header).
3. Configure the terminal program with the following settings:

- 38400 baud
- 8 data bits
- No Parity
- 1 stop bit

4. Set the following jumpers:

- JSelect = 2-3
- CLKA select = 1-2
- CLKB select = 1-2.

For more information on the EPXA1 Development Board, see the *EPXA1 Development Board Getting Started User Guide*.

Program the Flash Memory

You can run the **prog_hw.bat** files to program flash memory for both projects, if the ByteBlaster driver is installed and the jtagserver is running on the host system.

For more details on programing flash memory, see the Quartus II Help.

Rebuilding the Projects

Both projects contain the **build_project.bat** file, which rebuilds the application code for the respective projects.

Summary

The Excalibur devices provide you with a complete flexible embedded solution with the combination of the embedded stripe and the APEX FPGA. You can boot and run the processor subsystem independent of the FPGA configuration, and software control the FPGA contents. The multiple image reference designs show applications using multiple hardware images and applications that use multiple combined hardware and software images. The two designs provide the fundamental basis for flexible embedded systems using the Excalibur devices in many areas—reconfigurable computing, standard hardware platform development, and remote reconfiguration,

Revision History

Table 8 shows the document revision history.

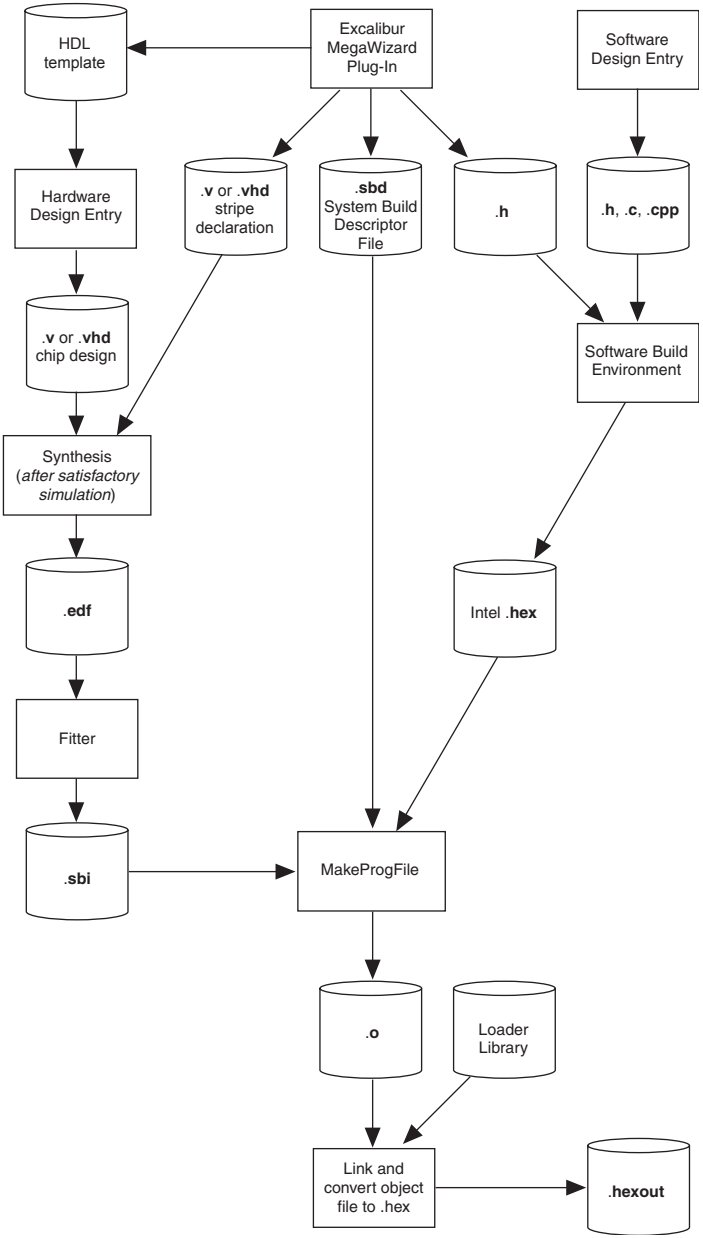
Table 8. Document Revision History

Date	Description
January 2003	Initial release.

Notes:

Figure 4 shows a block diagram of how the a **.hex** file is produced.

Figure 4. Configuration fram Flash Memory



To produce a **.sbi** and a **.hex** file for Excalibur devices, the output HDL and header files from the Excalibur Megawizard® Plug-In are used with design specific HDL and C code. The .**sbi** file is a binary representation of the hardware image; the .**hex** file is the application software for the design. These two images are combined with a System Build Descriptor File (**.sbd**) file to produce an object file, which is linked with the Altera bootloader library file and converted to **.hex** format for flash downloading.

For more details on producing programming files for Excalibur Devices, see *AN299: System Development Tools for Excalibur Devices*.

Notes:

Appendix B— Implementation Details

This appendix describes some of the key utilities that were used to create the reference designs. It also highlights important key areas of the designs, where you should focus attention to understand what is needed to create your own system.

Linking FPGA Hardware Images into Applications

Multiple hardware images accessible at an address visible by the embedded application are essential to produce a design that has multiple FPGA hardware images.

There are many ways to accomplish this task, but the following two solutions are used in the designs:

- ARMASM INCBIN Directive
- bin2elf utility

Alternatively, the application can assume that the **.sbi** file is going to be at a static location in the memory map and not link it in.

ARMASM INCBIN Directive

The ARM assembler supports an assembler directive that allows files to be appended to the file being assembled. The file is placed in the current section where the directive is used. In this design, the INCBIN directive is used on the **sbi_data.s** file to bring the FPGA hardware images. Symbols are exported using the EXPORT directive and are then referenced as an externally defined address from **main()**.

For additional information, see the ADS Assembler Guide.

bin2elf Utility

The Altera bin2elf utility translates plain binary files to executable and linkable file format (**.elf**). This can be used as an alternative to the ARMASM INCBIN Directive. Similar to the INCBIN approach, section and address information can be defined using this utility, which allows the symbols to be referenced elsewhere in the application.

Project 2 Boot Code

Project 2 has a simple example of a simple boot loader, which performs some of the basic functions of a sophisticated boot loader. It follows closely to the boot code shown in *AN 187: Booting Excalibur Devices*. The boot code performs the following functions:

- Reads IDCODE
- Sets up PLLs
- Sets up memory map
- Sets up embedded stripe I/O
- Sets up SDRAM controller
- Copies code SDRAM
- Configures FPGA (using similar techniques describe above)
- Reads boot parameters and branch to specified location

The boot code allows the same file to be used regardless of the location where it is currently being executed. One of the steps performed during the memory map setup is the remapping of the EBI to the execution locations. Take care when remapping, because the application is running out the EBI.

Appendix C—FPGA Configuration

Introduction

This appendix describes the configuration logic in Excalibur devices. An overview on the configuration logic is given, along with steps need to configure the FPGA under processor control.

Configuration Logic Overview

The configuration logic resides on AHB2. It is responsible for transferring configuration data to the FPGA and setting up the system so that the embedded processor can boot. It behaves similarly to the configuration logic specified in the *APEX 20K Programmable Logic Device Family Data Sheet*, but there are two additional features. The Excalibur family allows you to perform the following actions depending on the configuration mode selected:

- Set up embedded stripe registers and on-chip SRAM as part of the configuration bit stream
- Configure or reconfigure the FPGA via the embedded processor.

Boot from Serial Configuration Mode

In boot-from-serial mode, the Excalibur device receives its configuration data via a bit stream, which can contain FPGA data, embedded stripe registers configuration data, and embedded software application code. The configuration logic can receive the bit stream via the APEX passive-serial, passive-parallel configuration schemes. In addition to FPGA configuration, the configuration logic also acts as a master on AHB2 and programs embedded stripe registers and on-chip SRAM. After programming is complete, the embedded processor is brought out of reset and fetches its first instruction from address 0x0. The location for memory or peripherals mapped to address 0x0 is defined in the embedded stripe registers.

Boot from Flash Configuration Mode

In boot-from-flash mode, the embedded processor boots address 0x0, which at power up is mapped to EBI0. The Altera bootloader then initializes the system to what you specified in the Excalibur MegaWizard Plug-In. If the Altera bootloader is not used, the user application code is responsible for system initialization.

In either configuration mode, the FPGA can be reconfigured at anytime during system operation.

For details on Excalibur device configuration toolflow and details on the Altera bootloader, refer to *AN299: System Development Tools for Excalibur Devices*.

For APEX passive serial and passive parallel configuration, refer to *AN116: Configuring SRAM-Based LUT Devices*.

For writing boot code for Excalibur devices, refer to *AN187: Excalibur Solutions—Booting ARM-Based Devices*.

For additional details on configuration logic, refer to the *Excalibur Devices Hardware Reference Manual*.

The .sbi File

The slave-port binary (**.sbi**) file is produced by the Quartus II software and represents the FPGA contents. The Altera bootloader uses this file for FPGA configuration. You can use the file for reconfiguration by user application code. Table 9 shows the **.sbi** file format.

Table 9. SBI File Format

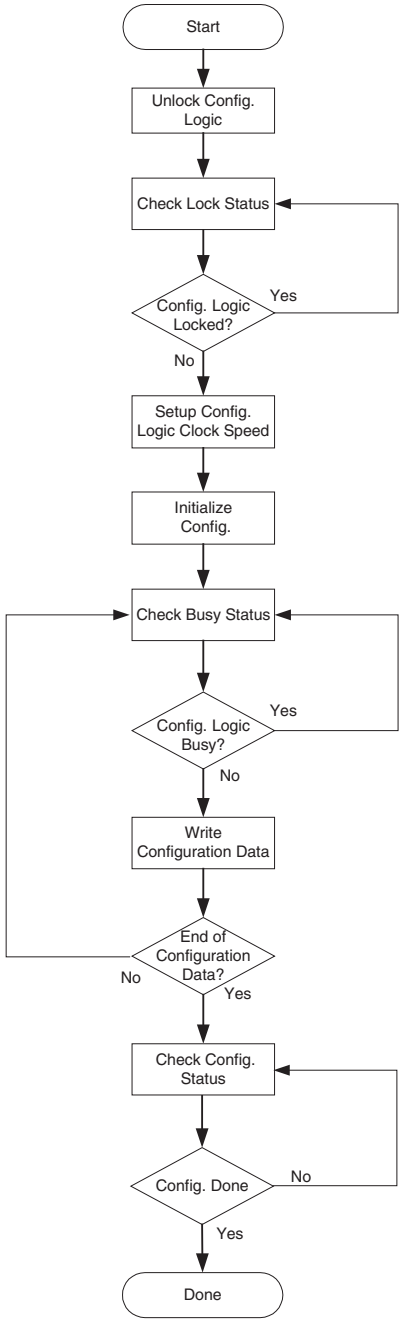
Offset	Size	Data
0H	4	Signature “SBI\0”.
4H	4	IDCODE for target system.
8H	4	Offset to configuration data (*coffset*).
CH	4	Size of configuration data in bytes (*csize*). Must be a multiple of 4.
coffset	*csize*	FPGA configuration data. This is a byte stream to be written to the FPGA slave port.

The **.sbi** file contains a file signature, device IDCODE, offset to the configuration data, the size of the configuration data, and the FPGA configuration data.

FPGA Configuration Steps

The configuration logic has a simple interface to application code. Figure 5 shows the steps to configure or reconfigure the FPGA contents. In addition, example code is in **\project1\software\config_logic.c**.

Figure 5. Configure the FPGA Contents



To configure or reconfigure the FPGA contents, perform the following steps:

1. Unlock the configuration logic unit by writing 0x554E4C4B to the CONFIG_UNLOCK register.
2. Check the lock status, by reading the LK bit of the CONFIG_CONTROL register (1 is unlocked; 0 is locked).
3. Setup the configuration logic clock speed, by writing (*ratio value*) to the *CONFIG_CLOCK* register.
4. Initialize the configuration, by writing 1 to the CO bit of the CONFIG_CONTROL register.
5. Check the busy status, by reading the B bit of the CONFIG_CONTROL register (1 is busy; 0 is not busy).
6. Write the configuration data, by writing the 32-bit SBI data value to the CONFIG_DATA register.
7. Check the configuration status, by reading the CO bit of the CONFIG_CONTROL register (1 is not done; 0 is done).

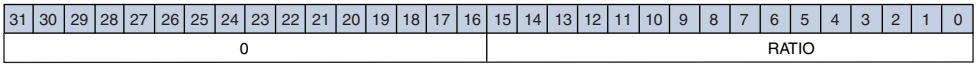
Configuration Logic Registers

Register: CONFIG_CONTROL (configuration control register)
Address: Register base + 140H
Access: Read/write

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0																								ES			E	PC	B	CO	LK

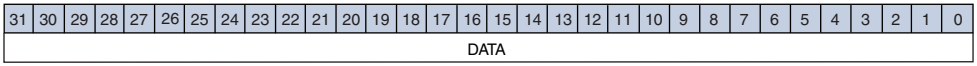
LK	R/W	Lock. When set, writes to this register cause a bus error instead of changing the register contents
CO	R/W	Configure. When set, indicates that the PLD is being configured via the PLD configuration controller. The PLD configuration controller clears this bit when configuration is complete
B	R	Busy. When set, either the data register is not ready to accept data (and inserts wait states if written to) or the PLD controller is resetting in preparation for configuration
PC	R	When set, the PLD is configured and in user mode
E	R	Error. When set, indicates that (re)configuration was not possible for one of the reasons below. – External configuration prevents reconfiguration – The PLD is being configured via JTAG – There was an error in the PLD bitstream – CONFIG_CLOCK value is 0 This bit is latching and is cleared by setting CO while ES is 0
ES	R	Error source. These (non-latching) bits indicate that reconfiguration is not currently possible for one or more reasons. The value presented is the logical or of the following values: 001—PLD being configured via JTAG 010—CONFIG_CLOCK value is zero 100—PLD being configured via external configuration interface
0	R	Reserved for future use. Write as 0 to ensure future compatibility

Register: CONFIG_CLOCK (configuration clock register)
Address: Register base + 144H
Access: Read/write



RATIO	R/W	Half the divide ratio applied to AHB2 clock to generate the PLD download clock. No clock is generated when this has the value 0
0	R	Reserved for future use. Write as 0 to ensure future compatibility

Register: CONFIG_DATA (embedded controller data register)
Address: Register base + 148H
Access: Write



DATA	W	Data written to PLD configuration logic