

Introduction



EXCALIBUR™



Altera® provides users of Excalibur™ systems with a powerful multi-layered simulation environment that can be used to extensively verify system-on-a-programmable-chip (SOPC) designs, as follows:

1. First, the AMBA™ AHB-based peripherals can be verified using the Bus Functional Model (BFM).

The BFM exercises the interface between the user peripherals and the ARM® embedded processor in an Excalibur device. It should be used in the early stages of system design, because it provides a fast solution to verifying the functionality of AHB peripherals.

The BFM is documented in the *Bus Functional Model User Guide*.

2. After verifying the functionality of the peripherals, the next stage in verifying an SOPC design is to test the integrity of the complete system, using either of the following models:
 - The Excalibur stripe simulator (ESS)—which is a fast, functionally-accurate model of the Excalibur embedded stripe. ESS can be used both as a fast hardware simulator and as an instruction set simulator (ISS). However, ESS is not cycle-accurate.
For further details about ESS, see the *Excalibur Stripe Simulator User Guide*.
 - The full stripe simulation model—which allows you to perform cycle-accurate simulations of the Excalibur embedded processor and the other components of the embedded stripe.

This document explains how to use the full stripe simulation model.

Full Stripe Simulation Model Overview

The full stripe simulation model provides a system-level verification environment for Excalibur designs. It is used to monitor the interaction between user software code running on the ARM922T™ processor, interacting with any of the peripherals implemented on the stripe, and any logic designed on the PLD side of the Excalibur device. All of the stripe peripherals are modelled by the full stripe simulation model, including the on-chip SRAM and dual-port SRAM (DPRAM). The model is very powerful, because users can monitor many of the internal signals of the stripe and view their relation to other signals in the design in a fully cycle-accurate manner.

The model makes the following stripe nodes visible to users during simulation:

- ARM922T internal registers
- Embedded stripe registers
- AHB1 bus signals
- AHB2 bus and arbitration signals
- Bridge signals
- Interrupt signals

Excalibur Simulation Flow

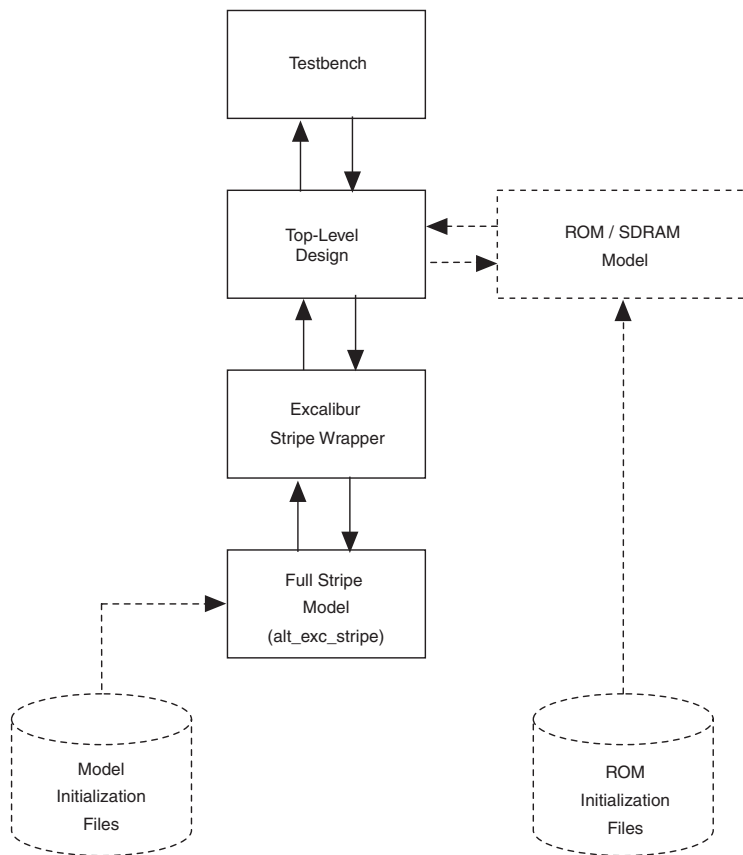


Excalibur systems are designed and configured using the Excalibur MegaWizard® Plug-In Manager. The MegaWizard Plug-In allows you to configure the Excalibur embedded stripe; it generates a VHDL or Verilog HDL wrapper file that instantiates a configured `alt_exc_stripe` entity.

See the *Excalibur Hardware Design Tutorial* for more information on configuring the stripe using the Excalibur MegaWizard Plug-In Manager.

[Figure 1 on page 3](#) shows the simulation flow using the full stripe simulation model

Figure 1. Full Stripe Simulation Flow



After you have configured the stripe using the Excalibur MegaWizard Plug-In, you connect the stripe wrapper to your top-level design as shown in Figure 1. The stripe wrapper instantiates the `alt_exc_stripe` entity that is defined in the full stripe model (`alt_exc_stripe.v` or `alt_exc_stripe.vhd`).

Software code for the full stripe model can be stored in either external memory models or in model initialization files (MIFs), which are used to initialize the on-chip memory of the Excalibur device.

Initializing the Full Stripe Simulation Model

The full stripe simulation model can run software code directly from the on-chip SRAM or DPRAM by using the MIFs. MIFs provide faster simulations than external memory models, because they allow the processor to run directly out of the fast internal SRAM rather than the slower external ROM connected to the EBI bus.

If you are simulating code running from on-chip memory in conjunction with the full stripe model, it is important to include the MIFs listed in [Table 1](#) in the current simulation directory:

Table 1. Memory Initialization Files	
MIF Name	Function
memory.SRAM0 memory.SRAM1	Used to initialize the on-board SRAM. Either or both of these files typically contains the software image for the system.
memory.DPRAM0 memory.DPRAM1	Model the initial contents of the DPRAM blocks of the stripe. These files are typically empty unless they are specifically targeted to contain the software image.
memory.REGS	Describes the states of the internal registers of the system. The file sets up various configuration registers for the PLL, DPRAM, SRAM, EBI, UART and other stripe components.

DPRAM initialization files can be modified to provide initialization data for designs that use the DPRAM to interface to the PLD fabric. Memory values are written to the MIF files in the following format:

```
mm/xxxxxxxx
```

where *mm* is the address and *xxxxxxxx* is the data value to be written to it. An example DPRAM file is shown below:

```
# Memory image of DPRAM0
#
0/EA00002A; 1/EA000006; 2/EA000009; 3/EA00001B;
4/EA00001E; 5/EA000001; 6/EA000014; 7/EA00001F;
8/EAFFFFFE; 9/E92D5FFF; A/EB0000B5; B/E8BD5FFF;
C/E25EF004; D/E92D5FFF; E/E25E0004; F/E5900000;
```



For more information on MIF formats, refer to [Appendix C](#).

MIFs are created by the Quartus II Software Builder when you compile your software project. Alternatively, the **makeprogfile** utility provided with the Quartus II software can be used to generate MIFs for projects that were not built using the Quartus II Software Builder. **makeprogfile** reads the user software image (.hex file) and system builder descriptor (.sbd) stripe configuration file and generates MIFs based on the data in them.

MIFs are created by using the following command line syntax for the **makeprogfile** utility:

```
makeprogfile -m memory <file name>.sbd <file name>.hex
```

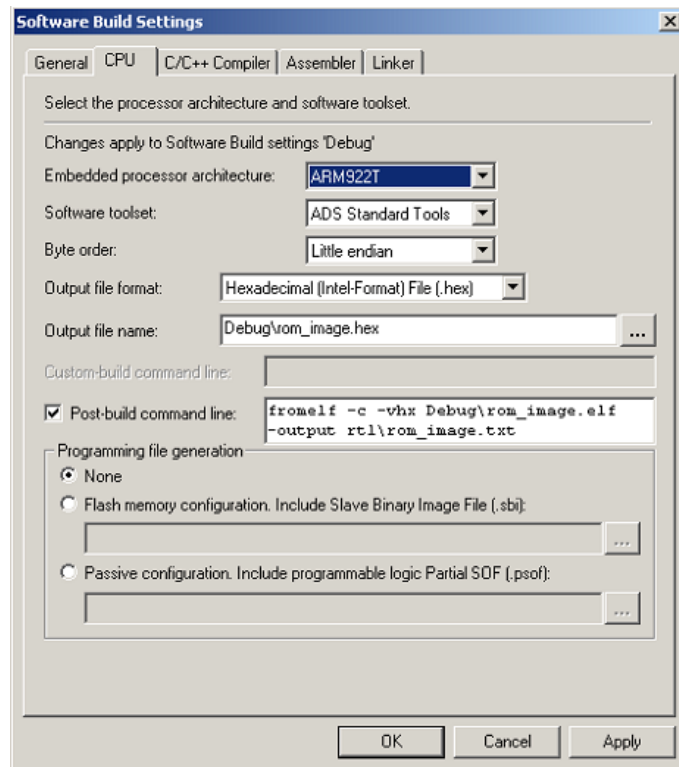
Running Software Images from an External ROM

The **.sbd** file used by **makeprogfile** is generated when you create and configure an Excalibur system using the MegaWizard Plug-In in the Quartus II software. The **.sbd** file describes the exact configuration of the stripe, including the memory map, stripe peripheral settings, and stripe clock settings. The **.hex** file input to **makeprogfile** is the software image generated by your compilation toolset, targeting the ARM922T processor.

Software images to be simulated running on the full stripe model can be stored in an external ROM model instead of MIFs. Although running designs out of a ROM model is slower than running from SRAM using a MIF, it provides a more realistic model of the system.

To simulate a software image running out of ROM, you must connect a ROM model containing the software image to the top-level design. The Quartus II Software Builder can be used to generate a hex file to initialize the ROM. An example configuration of the Software Builder is shown in [Figure 2](#).

Figure 2. Creating an ROM Initialization File with the Software Builder



The ROM initialization file is named **rom_image.txt** in the example shown in [Figure 2](#). This hex file is converted from the **.elf** file created by the software compiler. Designers using the ADS compiler tools can create hex images using the **fromelf** utility, which can be run from the Quartus II Software Builder as shown in [Figure 2](#). An example call to the **fromelf** utility is shown below:

```
fromelf -c -vhx <software image name>.elf -output <ROM initialization file name>
```

The **-vhx** flag in the utility generates a Verilog hex file formatted output. For VHDL users the **-i32** flag can be used to create an Intel hex output file.



For more information on the **fromelf** utility, see the *ARM Developer Suite Version 1.1 Compilers, Linker, and Utilities Guide*.

Compiling and Simulating a Design

Designs can be simulated using the full stripe model after the MIF or ROM initialization files have been generated. The compilation order for RTL simulations is listed below:

1. **alt_exc_stripe.vhd**
2. **<stripe wrapper filename>.vhd**
3. **<any user logic modules>.vhd**
4. **<rom model filename>.vhd** (optional)
5. **<top-level filename>.vhd**
6. **<testbench filename>.vhd**

The following compilation order is used for timing simulations:

1. **apex20ke_atoms.vhd**
2. **apex20ke_components.vhd**
3. **apex20ke_stripe.vhd**
4. **<stripe wrapper filename >.vhd**
5. **<any user logic modules >.vhd**
6. **<rom model filename>.vhd** (optional)
7. **<top-level filename>.vhd**
8. **< testbench filename >.vhd**

MIFs must be placed in the current simulation directory. Verilog designs require the same files as listed above, but with **.v** filename extensions.



For more information on the file location and structure of the full stripe model, refer to [Appendix A](#).

After compiling the necessary files you can load your design into your target simulator and begin simulation. The full stripe model then begins to execute instructions from your external ROM model if one is present or from the MIFs.



Refer to [Appendix B](#) for more information on configuring your simulator to run the full stripe model.

Conclusion

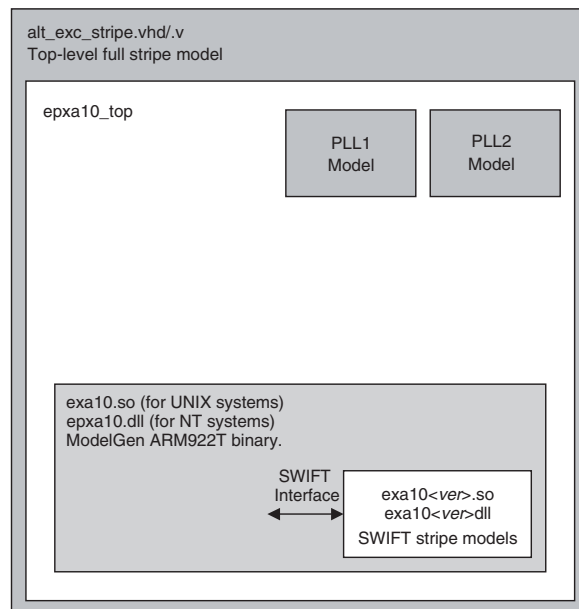
The full stripe model provides a cycle-accurate simulation model that allows you to extensively simulate Excalibur designs. The simulation model is ideal for system-level verification, because it provides a platform for observing the interaction between system software and hardware.



Notes:

Excalibur stripe simulation models are generated by the ModelGen modelling software. The full stripe model generated by ModelGen is described by the `alt_exc_stripe.v` and `alt_exc_stripe.vhd` files, which are included with the Quartus II 2.1 installation. The `alt_exc_stripe` file instantiates a model of the stripe PLLs and a wrapper file for the ModelGen model. The ModelGen wrapper instantiates the binary model of the ARM922T embedded processor and the binary SWIFT model of the remaining stripe components. The structure of the full stripe models is shown in [Figure 1](#).

Figure 1. Full Stripe Simulation Model Structure



The file location structure of the full stripe model is shown below:

```
<Quartus II 2.1 install directory>/eda/sim_lib/excalibur  
  /lpm  
  /stripe_model_hpux  
  /stripe_model_nt  
  /stripe_model_solaris  
  /swift
```

Full stripe simulation models are included for the Windows NT/2000 operating systems as well as for HPUX and Solaris. The **alt_exc_stripe** simulation models are located within the folders for the operating system that you are using. For example, the Windows NT directory structure is shown below.

```
<Quartus II 2.1 install directory>/eda/sim_lib/Excalibur/stripe_model_nt  
  /ModelGen  
    /models  
      /epxa1  
      /epxa4  
      /epxa10  
      /r0  
      /WinNT  
      /mti_modelsim_verilog  
        alt_exc_stripe.v  
        apex20ke_stripe.v  
      /mti_modelsim_vhdl  
        alt_exc_stripe.vhd  
        apex20ke_stripe.vhd
```

The full stripe simulation model supports operation on the simulators and platforms shown in [Table 1](#).

Table 1. Simulators and Platforms Supported by the Full Stripe Simulation Model				
Simulator	Solaris	Win NT /2000	HPUX	
			10.2	11.0
ModelSim VHDL	✓	✓	✓	✓
ModelSim Verilog	✓	✓	✓	✓
NC Sim VHDL	✓		✓	✓
NC Sim Verilog	✓		✓	✓
NC Sim VHDL	✓			
NC Sim Verilog	✓			
VCS Verilog	✓ (1)		✓ (2)	✓ (2)

Notes:

- (1) Supported by Verilog VCS version 6.0.
- (2) Supported by Verilog VCS version 5.2.1.

Environment Variables

In order to compile and simulate the full stripe model correctly, it is important to verify that a number of environment variables are set.

MG_LIB, MG_MODEL_PATH & MG_ROOT

MG_LIB, MG_MODEL, MODEL_PATH, and MG_ROOT specify the location of the ModelGen manager. These variables should be set for all operating systems, irrespective of the simulation tool, as follows:

- For all operating systems, MG_ROOT should be set to:

```
$QUARTUS_ROOTDIR/eda/sim_lib/excalibur/stripe_model_solaris/ModelGen/manager
```

- For Solaris systems, MG_MODEL_PATH should be set to:

```
$QUARTUS_ROOTDIR/eda/sim_lib/excalibur/stripe_model_solaris/ModelGen/models
```

- For HPUX systems, `MG_MODEL_PATH` should be set to:

```
$QUARTUS_ROOTDIR/eda/sim_lib/excalibur/stripe_model_hpux/ModelGen/models
```

- For Windows systems, `MG_MODEL_PATH` should be set to:

```
$QUARTUS_ROOTDIR/eda/sim_lib/excalibur/stripe_model_nt/ModelGen/models
```

- For Solaris systems, `MG_LIB` should be set to:

```
$MG_ROOT/SunOS5/MM
```

- For HPUX systems, `MG_LIB` should be set to:

```
$MG_ROOT/HPUX/MM
```

- For Windows systems, `MG_LIB` should be set to:

```
$MG_ROOT/WinNT/MM
```

LMC_HOME

The `LMC_HOME` variable specifies the location of the Swift model. For all operating systems, the `LMC_HOME` variable should be set to:

```
$QUARTUS_ROOTDIR/eda/sim_lib/excalibur/swift
```

where `$QUARTUS_ROOTDIR` is the Quartus II installation directory.

ARM922T_DISASS & ARM922T_DISASS_OUTPUT

The full stripe model can redirect the assembly instructions currently being executed by the ARM922T processor either to the simulator screen or to a file. Set `ARM922T_DISASS` to `ON` to enable the disassembly feature of the model, which by default redirects the disassembly sequence to the simulator screen. Alternatively, you can output the sequence to a file by setting `ARM922T_DISASS` to `ON` and also setting the environment variable `ARM922T_DISASS_OUTPUT` to *<file location>*. With this combination of variables, the full stripe model outputs a disassembly sequence to the file specified by `ARM922T_DISASS_OUTPUT`.

Using the Full Stripe Model with ModelSim

Designers simulating with ModelSim VHDL must use SE versions of ModelSim, because the full stripe model utilizes FLI calls which are only supported in ModelSim SE.

To use ModelSim as your simulator, add the following location to the system path:

```
%QUARTUS_ROOTDIR%\eda\sim_lib\excalibur\swift\lib\pcnt.lib
```

When simulating Verilog-based Excalibur designs in ModelSim, it is important to specify the location of the ModelGen manager in the **modelsim.ini** or ModelSim project file (**.mpf**) used in your simulation. To specify the location of the model manager, add the following line to the [vsim] section of the **.ini** or **.mpf** file that you are using for your ModelSim project:

```
Veriuser = $MG_LIB/mti_modelsim_verilog/libmgmm.so
```

ModelSim **.ini** or **.mpf** files typically specify a default Veriuser setting. When using the full stripe model, the Veriuser line above should be specified above any other Veriuser settings in the **.ini** or **.mpf** file.

Using the Full Stripe Model with Synopsys VCS Verilog

When using the full stripe model with VCS Verilog, it is important to insert `$(MG_LIB)/synopsys_vcs_verilog` in the `LD_LIBRARY_PATH` environment variable before the Synopsys libraries.

A PLI table must be specified on the VCS command line using the following command:

```
-P pli.tab
```

Example `pli.tab` files are included with Quartus II in `$MG_MODEL_PATH/<device name>r0/synopsys_vcs_verilog`

You must also specify the location of the model manager on the VCS command line by including the following setting on the VCS command line:

```
-LDLFLAGS "-L${MG_LIB}/synopsys_vcs_verilog" -lmgmm
```

Using the Full Stripe Model with Cadence NC Sim

When using the full stripe model with with Cadence NC Sim, you set the `LD_LIBRARY_PATH` environment variable as follows and insert it before the Cadence libraries:

- If you are using NC Sim Verilog, set `LD_LIBRARY_PATH` to `$(MG_LIB)/cadence_nc_verilog`
- If you are using NC Sim VHDL, set `LD_LIBRARY_PATH` to `$(MG_LIB)/cadence_nc_vhdl`

This information is extracted from the Synopsys *Smart Model Library User's Guide*.

A memory initialization file (MIF) contains one or more records. Each record specifies the data to be written to one or more memory locations.

MIF Conventions

The following list shows the conventions and rules that apply to the syntax description for MIF records:

- Braces ({ }) indicate a list of one or more entries.
- Brackets ([]) indicate optional entries.
- Italics indicate variables for which you specify actual values.
- Fields are not case-sensitive.
- More than one record can appear on a line.
- The character "X" or "x" indicates an unknown value, and is illegal except in a data word where the data is expressed in binary, octal, or hexadecimal (not decimal).

MIF Record Syntax


The following syntax applies to MIF records:

```
{address1 [:address2]/base_specifier data_value;} [# comment]
```

where:


address1—is the memory location to which data is to be written, or the beginning address of a range.

:address2—is the end address of a range. Either a colon (:) or a hyphen (-) can be used as a delimiter.

 A slash (/) separates the address specification from the data word.

/base_specifier—is the *base_specifier* argument, which must be one of the following:

- `b Binary
- `o Octal
- `d Decimal
- `h Hexadecimal (the default)

 You can mix different base numbers within a record.

data_value—is the value of the data word to be written to the specified memory locations.

 A semicolon (;) defines the end of each record.

comment—can be included in a record by using #. All following information on the line is treated as a comment.

The examples below typify MIF records.

Example 1

The following example shows how various constructs can be used or combined in an MIF. In this example, the width of the memory location is 8 bits.

```
0:3/0; #Colon separator for address range
4-6/0; #Hyphen separator for address range
\d7/'b10101110; #Address and data can use a different
#numeric base
10/0; 11/'b10000000; #Two records on the same line
12:1e/'HxF; \d31/'hX8; #Information is case-insensitive
20:7FF/4; #Load remaining addresses with 00000100
```

Example 2

When you specify the data value to load into memory, the safest practice is usually to specify values that match the width of the memory location. However, this is not required for the Excalibur simulations. If the data value has fewer bits than the memory location, the model pads the value with leading zeros. If the data value is larger than the memory location, the model rejects the data and issues a warning message.

The following example specifies that the hex value F (binary 1111) is to be loaded into memory location 0 (zero). If the memory location is 9 bits wide, the value entered is 000001111; if the location is 6 bits wide, the value is 001111; and so on.

```
0/F
```

Example 3

Unknown values are most easily specified in binary; often the unknown represents a single bit. In the following example, for an 8-bit memory location the binary value is loaded into 0F exactly as written; the hex value xF is loaded into FA as xxxx1111. For a 9-bit memory location, the binary value is loaded as 01010x0x1 and the hex value as 0xxxx1111.

```
0F / 'b1010x0x1;
FA / xF;
```

Excalibur MIFs

If you generate a SWIFT model within a ModelGen model, all initialization files must have the same root identifier. In addition, file extensions are used to differentiate the five MIFs as follows:

- memory.regs**—register initialization data
- memory.sram0**—SRAM0 initialization data
- memory.sram1**—SRAM1 initialization data
- memory.dpram0**—DPRAM0 initialization data
- memory.dpram1**—DPRAM1 initialization data

Register MIF Format

In the register initialization file, **memory.regs**, the MIF format is extended to define the data value as a 44-bit binary value or, more conveniently, a 12-digit hex value:

```
aaadddddddd
```

where


aaa—is the offset of the register to be loaded within the register region of the memory map


ddddddd—is the data value to be loaded into the register. This value must always be 32 bits, padded with leading zeroes if required.

The *address1* and *:address2* fields of each record in **memory.regs** do not relate to the address of the register being loaded. Initially, the model loads **memory.regs** into an intermediate array; the address fields are the address of each MIF record in that array. The contents of the intermediate array are then parsed in ascending address order to determine the register offsets and data values to be loaded. Thus, *address1* and *:address2* determine the order in which registers are loaded by the model. This is important for some modules such as the SDRAM controller.

RAM MIF Format

All RAM initialization files share the same basic MIF format, in which the *address1* and *address2* fields of each record correspond to the word address within the RAM to be loaded.

 Take care when converting byte addresses to word addresses when you are generating memory initialization data.

 If memory files intended for simulation on a Solaris platform are not first converted using the command

```
dos2unix "memory.xxx"
```

the error **Illegal Entry in memory.xx** file occurs.



101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
<http://www.altera.com>
Applications Hotline:
(800) 800-EPLD
Literature Services:
lit_req@altera.com

Copyright © 2002 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

